

The Materials Research Society Series

Stefan Sandfeld

Materials Data Science

Introduction to Data Mining, Machine
Learning, and Data-Driven Predictions
for Materials Science and Engineering



The Materials Research Society Series

The Materials Research Society Series covers the multidisciplinary field of materials research and technology, publishing across chemistry, physics, biology, and engineering. The Series focuses on premium textbooks, professional books, monographs, references, and other works that serve the broad materials science and engineering community worldwide. Connecting the principles of structure, properties, processing, and performance and employing tools of characterization, computation, and fabrication the Series addresses established, novel, and emerging topics.

Stefan Sandfeld

Materials Data Science

Introduction to Data Mining, Machine Learning, and Data-Driven Predictions for Materials Science and Engineering

Stefan Sandfeld
Forschungszentrum Jülich GmbH
Institute for Advanced Simulation –
Materials Data Science and Informatics
(IAS-9)
52068 Aachen, Germany

ISSN 2730-7360 ISSN 2730-7379 (electronic)
The Materials Research Society Series
ISBN 978-3-031-46564-2 ISBN 978-3-031-46565-9 (eBook)
<https://doi.org/10.1007/978-3-031-46565-9>

© The Materials Research Society 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

For Ruth

Preface

Since the early 2000s, many scientific disciplines, including materials science and materials engineering, have been undergoing a disruptive change. This change is in fact made up of a series of small and large changes, a few of which can be considered real revolutions.

On the one hand, there is the advent of the digitization in the field of experimentation and microscopy, where an increasing number of machines and devices provide direct access to a wealth of digital data and information. In addition, there is a growing awareness—on the part of both manufacturers and scientists—that data from measuring devices should be easily accessible and reusable. Furthermore, simulation methods in computational materials science have reached a very high level of maturity today, making them a source for a wealth of digital data and information.

On the other hand, there are amazing methodological developments in the field of data science and computer science that have effectively “democratized” statistical learning and deep learning approaches through a series of well-designed and mature software libraries. These significantly lower the entry barrier and make the use of methods of, e.g., deep learning possible even for non-computer scientists. When one considers the rapid development of performance and the enormous drop in price of CPU and GPU hardware, it becomes clear that a pivotal point has been reached.

Today, almost everyone can see that machine learning has the potential to change the way we approach our research data in materials science and physics, whether it is small datasets or “big data.” This became clear at the latest after the release of OpenAI’s the “artificial intelligence chatbot *ChatGPT*” by the end of 2022 (which took place during writing of this book). However, there still seems to be a significant gap between what people believe machine learning and artificial intelligence can do for us and what *can* actually already be done in science and engineering.

This book is about data science; it is about exploring the large methodological toolbox of statistics, data visualization, classical machine learning, and deep learning approaches with a special focus on scientific applications. However, this book is not only about solving scientific problems and understanding data, but also about understanding analysis methods—because it is often important for assessing the quality of analysis results that we do not use analysis methods as black boxes.

With this in mind, the book focuses heavily on the derivation and numerical implementation of the majority of all algorithms and methods presented. In addition,

the book teaches the most important basics in mathematics and statistics and assumes that the reader has basic Python programming skills. I believe that a program is worth more than a thousand equations. Therefore, almost every method presented is also presented in form of Python code. The emphasis is not on performance, compactness, or elegant software design, but clearly on the didactic aspects, helping you, the reader, to understand basic concepts and equipping you with the tools to experiment and explore.

Studying data science and machine learning using generic datasets is certainly useful. However, transferring this to the materials science context can be difficult. Therefore, the book provides numerous materials science examples and datasets, most of which are included directly in the text or provided via the book's website supplementary webpage (<https://MDS-book.org>).

Last but not least, I would be very grateful about general feedback, if you found a typo or error, or have a suggestion for including an additional topic in the book. Please see the book's website supplementary webpage (<https://MDS-book.org>) for how to contact me.

About this Book: Goals and Structure

The aim of this book is to provide a readership with a physics or engineering background with the knowledge required to use data science methods (which includes machine learning) to solve problems.

Sometimes we use motivational descriptions that are rather useful to sharpen intuition, but can sometimes be a little less rigorous, e.g., in terms of mathematical proofs. Nevertheless, formal derivations can also be very useful, which is why we show them either in advanced chapters or through step-by-step explanations in the introductory chapters. Last but not least, having to “scrape together” information and explanations from different sources (books, lecture slides, online resources, etc.) can be counterproductive, as many different notations are used. One of the aims of this book is also to introduce consistent notations that are used for all methods and topics. Below we give a brief overview of the structure of the book.

Part I: Introduction and Foundations This part provides a broad overview of many historical and current developments that eventually led to data science and machine learning as we know it today. The consequences for materials science are then presented. An important chapter for practical purposes is the third chapter that covers all the basics about data, mathematics, as well as some computational aspects. At the end of this part, all datasets used in this book are presented.

Part II: A Primer on Probabilities, Distributions, and Statistics The second part is designed as an introduction to all relevant concepts and techniques, starting with combinatorics and probabilities, random variables and probability functions, as well as expectation values and moments. This is followed by the different types of “statistics,” the underlying concepts, and a number of interesting consequences for

data analysis. This part concludes with a larger section on exploratory data analysis and an overview of common distributions in materials science and engineering.

Part III: Classical Machine Learning One of the most important tools in data science is machine learning. In this part, we review basic machine learning concepts, important techniques, and a number of supervised and unsupervised “classical” machine learning methods, which often have their roots in the field of statistical analysis.

Part IV: Artificial Neural Networks and Deep Learning The fourth, big part continues with another kind of machine learning approaches, the artificial neural networks. This part follows the historical developments of neural networks and, building on this, motivates all the models presented. In the second half of this chapter, we move on to deep learning network architectures and introduce state-of-the-art approaches.

Finally, this part contains an overview of linear algebra for machine learning methods. This is not a complete introduction, but rather serves as a refresher for those who have some knowledge of vector and matrix calculus. This part also contains many Python examples. It helps make the book almost completely self-contained.

A “How to MDS-Book” for Getting a Quick Overview

For getting a quick overview of data science and machine learning, the first two chapters could be skipped. Chapter 3 gives a detailed description of the notations and conventions for data and datasets. The reader in a hurry might just glance at Sect. 3.3 and Fig. 3.3, skip the rest, and return to these subsections whenever it is necessary to refresh knowledge of some of the notations. The nomenclature at the beginning of this book should also help to navigate the jungle of variables names, notations, and conventions.

If the reader already has some knowledge of random variables and distributions, the first chapters of the Part II can be skipped and only the chapters on statistics (Chap. 8) and exploratory data analysis (Chap. 9) should be read. These are two chapters with general “data science” subjects that are particularly helpful for understanding the ground on which machine learning is introduced.

If a particular machine learning method is preferred, the reader can choose whether to focus on “classical” (statistical) learning in Part III or on neural networks and deep learning in Part IV, which is written, at least for the most part, such that it can be read independently of Part III. In any case, for a reader without any prior knowledge of machine learning, it is definitely advisable not to skip the introduction Chap. 11, which gives a comprehensive overview of machine learning methods and related subjects.

Acknowledgments

I would like to thank the following people (in alphabetical order) for their great help with this book:

- People who helped create, write or fine-tune examples, short passages of text, exercises or code: Aytekin Demirci, Abril Guzmán, Bashir Kazimi, Binh Duong Nguyen, Kishan Govind, Pavlo Potapenko, Sébastien Bompas, Tarek Iraki
More detailed information also on code contributions are given and updated on the book's supplementary webpage (<https://MDS-book.org>).
- People who reviewed one or more chapters, provided other support during the writing process: Aytekin Demirci, Abril Guzmán, Bashir Kazimi, Binh Duong Nguyen, Hariprasath Ganesan, Jens Bröder, Karina Ruzaeva, Katharina Immel, Hengxu Song, Markus Stricker, Marc Legros, Sébastien Bompas, Tarek Iraki
- Ruth Schwaiger for reviewing chapters and carefully proofreading and editing a significant part of the final draft of the book
- The Forschungszentrum Jülich GmbH for the opportunity to write this book.
- I would like to thank the community at <https://tikz.net> and <https://texample.net/tikz> for providing numerous $\text{\TeX}$ examples for TikZ figures that helped create many of the figures in this book.

Attribution

Some of the graphics used in this book are or contain emoji graphics licensed under CC-BY 4.0: <https://creativecommons.org/licenses/by/4.0/>, Copyright 2019, Twitter, Inc and other contributors.

Contents

Part I Introduction and Foundations

1	A Brief History of Data and Data Science	3
1.1	Where Do All the Numbers Come From?	3
1.2	The Ancient Roots of Data Science	4
1.2.1	Ancient Babylonia (1800–600 BCE)	4
1.2.2	The Age of Classical Antiquity ($\approx$ 750 BCE–600 CE)	6
1.2.3	The Medieval Period (approx. 500–1500 CE)	6
1.2.4	The Renaissance (approx. 1500–1700 CE)	7
1.2.5	OCCAM’s Razor	8
1.3	The First True Data Scientist	9
1.4	The More Recent Roots of Data Science	10
1.5	Data Science and Machine Learning in the Twentieth Century	11
1.6	Summary and Conclusion	12
	References	12
2	From Data Science to Materials Data Science	15
2.1	What Is Data Science and How Is It Related to Machine Learning and Artificial Intelligence?	15
2.1.1	Data Science	15
2.1.2	The Relation to Artificial Intelligence	16
2.1.3	From Machine Learning to Deep Learning	17
2.1.4	Domain Knowledge	17
2.2	Data Science and Machine Learning in Materials Science and Engineering	18
2.2.1	Materials Data Science Without Machine Learning?	18
2.2.2	A Brief Overview of Machine Learning in Materials Science	20
2.2.3	Example: Linkage of Analysis Methods for Data Mining of Microscopy Images	21
2.3	From Data and Information to Knowledge	23
2.3.1	“Data and Information” from the Point of View of Information Science	23
2.3.2	“Data and Information” from the Point of View of Data Science	26

2.4	The Curse of Dimensionality	26
2.5	Summary and Conclusion	28
	References	28
3	What You Should Know About Data, Math, and Computing	31
3.1	General Conventions and Notations	31
3.2	Sets, Tuples, Vectors, and Arrays	32
3.2.1	Sets	32
3.2.2	Tuple	35
3.2.3	Intervals	36
3.2.4	Matrices and Row and Column Vectors	37
3.3	Representation of Data in Statistics and Machine Learning	41
3.3.1	General Problem Formulation	41
3.3.2	Structured Data	43
3.3.3	Tabular Data and the Data Matrix	45
3.3.4	Matrix and Vector Short Forms for Linear Algebra	47
3.4	Summary and Conclusion	51
3.5	Exercises	51
	References	52
4	Materials Science Datasets and Data Generation	53
4.1	Introduction	53
4.2	Dataset MDS-1: Tensile Test with Parameter Uncertainties	53
4.3	Dataset MDS-2: Microstructure Evolution with the Ising Model	57
4.4	Dataset MDS-3: Cahn-Hilliard Model	58
4.5	Dataset MDS-4: Properties of Chemical Elements	60
4.6	Dataset MDS-5: Nanoindentation of a Cu-Cr Composite	60
4.7	Dataset DS-1: The Iris Flower Dataset	62
4.8	Dataset DS-2: The Handwritten Digits Dataset	63
4.9	Online Resource for Obtaining Training Data	64
	References	65
Part II A Primer on Probabilities, Distributions, and Statistics		
5	Combinatorics and Probabilities	69
5.1	Combinatorics	69
5.1.1	Counting with the Additive Principle	70
5.1.2	More Counting with the Multiplication Principle	71
5.1.3	Factorials	73
5.1.4	Permutations—Ordered Sets Without Replacement ...	74
5.1.5	Permutations of n items, k chosen at a time: Generalization of the Multiplication Principle	79
5.1.6	Permutations with Repetition—Ordered Sets/Samples With Replacement	80

5.1.7	Combinations Without Replacement—Unordered Sets/Samples Without Replacement	81
5.1.8	Combinations with Replacement—Unordered Sets/Samples with Replacement	84
5.1.9	Summary: Permutations and Combinations	85
5.2	Probabilities	86
5.2.1	What is Probability?	87
5.2.2	Two Python Examples	90
5.2.3	Outcomes, Events, and the Sample Space	92
5.2.4	Calculating with Events and Probabilities	94
5.2.5	The KOLMOGOROV Axioms	97
5.3	Conditional Probabilities, Product rule, and Bayes' theorem.....	99
5.4	Summary	100
5.5	Exercises	101
	References	102
6	Random Variables and Probability Functions	103
6.1	Random Variables.....	103
6.1.1	Discrete Random Variables.....	105
6.1.2	Continuous Random Variables	106
6.1.3	Assigning Probabilities to Discrete Random Variables	106
6.1.4	Assigning Probabilities to Continuous Random Variables.....	107
6.2	Introduction of Probability Functions.....	108
6.2.1	Why Do We Need Probability Functions?.....	108
6.2.2	Classification of Distribution Types.....	109
6.3	Discrete Probability Distributions.....	110
6.3.1	Probability Mass Functions	110
6.3.2	Cumulative Frequencies	112
6.4	Continuous Probability Distributions	113
6.4.1	Probability Density Functions.....	113
6.4.2	Cumulative Distribution Functions	115
6.4.3	The Difference Between a Function and a Distribution	120
6.4.4	A Brief Summary of the Most Important Properties ...	120
6.5	Multivariate Discrete and Continuous Distribution	121
6.5.1	Generic Formulation of the Joint CDF.....	122
6.5.2	Joint Probability Functions for Discrete Random Variables.....	123
6.5.3	Joint Probability Functions for Continuous Random Variables.....	123
6.5.4	Marginal Probability	125
6.5.5	Conditional Probability	125
6.6	Bivariate Distributions as a Special Case	126
6.6.1	Joint Probability for Two Discrete Random Variables ..	126

6.6.2	Joint Probability for Two Continuous Random Variables	126
6.7	Summary	128
6.8	Exercises	128
7	Expectation, Variance, and Moments	131
7.1	Expected Values of Discrete Random Variables	131
7.1.1	Definition and Examples	131
7.1.2	Calculating with Expectation Values	133
7.2	Variance and Standard Deviation	137
7.2.1	Definition and Examples	137
7.2.2	The Standard Deviation	139
7.2.3	Calculating with Variances	139
7.3	Raw Moments	140
7.3.1	General Formulation	140
7.3.2	The Zero-th Raw Moment	141
7.3.3	The First Raw Moment or the Mean	142
7.4	Central Moments	142
7.4.1	General Formulation	143
7.4.2	Zero-th and First Central Moments	144
7.4.3	The Second Central Moment or the Variance	144
7.5	Standardized Moments	145
7.5.1	General Formulation	145
7.5.2	Aspects of Standardized Moments	146
7.6	Exercises	146
8	Introduction to Statistics	149
8.1	And What Now <i>Is</i> Statistics?	149
8.2	The Sample and the Population	150
8.2.1	The Sample and the Population	150
8.3	Two Flavors: Descriptive and Inferential Statistics	152
8.3.1	What is “Sampling”?	153
8.4	Sampling Strategies	154
8.4.1	Simple Random Sampling	155
8.4.2	Systematic Sampling	156
8.4.3	“Take from Top” Sampling	158
8.4.4	Weighted Sampling	158
8.4.5	Stratified Sampling	159
8.5	The Law of Large and Truly Large Numbers	161
8.5.1	The Law of Large Numbers	161
8.5.2	The Law of Truly Large Numbers	163
8.6	Central Limit Theorem	163
8.7	Relations Between Multivariate Variables: Covariance and Correlation	165
8.7.1	Covariance	165
8.7.2	Covariance Matrix	167

8.7.3	Correlation	172
8.7.4	Correlation Matrix	175
8.8	Exercises	177
	Reference	178
9	Exploratory Data Analysis	179
9.1	The Why, the When, and the How	179
9.2	Two Preliminary Steps	180
9.2.1	Initial Exploration of Data File(s)	180
9.2.2	A Quick Initial Visualization	182
9.3	Descriptive Statistics	183
9.3.1	Simple Descriptions of Univariate Distributions	184
9.3.2	Simple Descriptions of Multivariate Distributions	185
9.3.3	Central Tendency: The Mean, the Median, and the Mode	187
9.3.4	The “Spread” of a Distribution: The Variance	190
9.3.5	Example: Summary Statistics of the Iris Dataset	193
9.3.6	Skewness	193
9.3.7	The Fourth Moment: Kurtosis	195
9.3.8	Summary	195
9.4	Data Visualization	196
9.4.1	Scatter Plots	197
9.4.2	Histograms	198
9.4.3	Box-Whisker Plots	200
9.4.4	Advanced Exploratory Plot Types	202
9.5	Exercises	204
	References	206
10	Commonly Encountered Distributions in Materials Science and Engineering	207
10.1	About the Following Discrete and Continuous Distributions	207
10.2	Discrete Uniform Distribution	208
10.3	Bernoulli Distribution	210
10.4	Binomial Distribution	211
10.5	Geometric Distribution	213
10.6	Poisson Distribution	215
10.7	Normal Distribution	217
10.8	Bivariate Normal Distribution	219
10.9	Multivariate Normal Distribution	222
10.10	The Relation Between Covariance Matrix and Multivariate Normal Distribution	223
10.11	Lognormal Distribution	227
10.12	Exponential Distribution	229
10.13	Logistic Distribution	230
	References	231

Part III Classical Machine Learning

11	Introduction and General Concepts of Machine Learning and Data Science	235
11.1	The Definition(s) of Machine Learning	235
11.2	How and What Do Machines Learn?	239
11.2.1	The Machine Learning Model	239
11.2.2	A First Learning Algorithm: Instance-Based Learning	240
11.3	Introduction of the General Machine Learning Workflow	243
11.4	Data Collection	244
11.5	Data Preprocessing	245
11.5.1	Data Cleaning	245
11.5.2	Outlier Detection	246
11.5.3	Data Transformation	247
11.5.4	Other Transformations	250
11.6	A Taxonomy of Machine Learning Models	252
11.6.1	Overview of Regression	253
11.6.2	Overview of Classification	254
11.6.3	Overview of Clustering	255
11.6.4	Overview of Dimensionality Reduction	256
11.6.5	The Zoo of Names for Input and Output Variables	257
11.7	Error Measures for Numerical Data	257
11.7.1	Introduction to the Problem	258
11.7.2	Mean Absolute Error	259
11.7.3	Mean Squared Error	259
11.7.4	Root Mean Square Error	260
11.7.5	Coefficient of Variation	261
11.7.6	Root Mean Squared Log Error	261
11.8	Similarity Measures for Classification Problems	262
11.8.1	From True Positives to False Negatives	262
11.8.2	The Jaccard Similarity or Intersection Over Union	263
11.8.3	The Dice Coefficient	265
11.8.4	Precision and Recall	266
11.8.5	Categorical Cross-Entropy	266
11.8.6	Example: Binary Segmentation of a TEM Image	267
11.9	Exercises	269
	References	270
12	A First Approach to Machine Learning with Linear Regression	271
12.1	The Roots of Regression Analysis	271
12.1.1	A Historical Perspective	271
12.1.2	Statistics Versus Machine Learning	273
12.2	General Concepts and Important Terminology	275
12.2.1	Interpolation	277
12.2.2	Extrapolation	278
12.2.3	True Functions and Noise	278

12.2.4	The Training Process and the Machine Learning Model.....	279
12.2.5	Hypothesis/Hypothesis Space and the Parameter Space	280
12.2.6	Predictions	281
12.2.7	Inference	281
12.2.8	Under- and Overfitting.....	282
12.2.9	Notations for Input and Output Variables.....	283
12.2.10	Training and Testing: Datasets and Errors Types.....	283
12.3	Simple Linear Regression	287
12.3.1	The Linearity in Linear Regression	288
12.3.2	Hypothesis Space and Parameter Space	289
12.3.3	Estimating Deviations: Errors and Residuals.....	291
12.3.4	The Loss Function and the Loss	292
12.3.5	The Cost Function	293
12.3.6	Cost Function Minimization: Analytical Derivation ...	296
12.3.7	Cost Function Minimization: The Method of Gradient Descent.....	297
12.4	Computational Aspects of Vectorization.....	302
12.4.1	The First Type of Vectorization: Partial Vectorization .	302
12.4.2	A Second Type of Vectorization: Full Vectorization ...	303
12.5	A Worked Example of Simple Linear Regression.....	304
12.5.1	Python Implementation	304
12.5.2	Visualization and Discussion of Training Results.....	306
12.5.3	Validation of the Trained Model	309
12.6	Multiple Linear Regression Models.....	310
12.6.1	Hypothesis Formulation	310
12.6.2	Special Case: Multiple Linear Regression with Two Features	310
12.6.3	Generalization to an Arbitrary Number of Features....	312
12.7	Exercises	313
	References.....	315
13	Advanced Methods and Topics of Regression	317
13.1	Nonlinear Model Behavior with Linear Regression.....	317
13.1.1	Polynomial Regression	318
13.1.2	Other Types of Hypothesis Functions.....	321
13.1.3	Linearization of Nonlinear Hypotheses	322
13.2	Generalized Formulations and Vectorization for Multiple Linear Regression.....	323
13.2.1	Hypothesis and Extended Feature Matrix	324
13.2.2	Derivation of Cost Function and Cost Gradient	325
13.2.3	Closed Form Solution	327
13.2.4	Summary	328
13.3	Generalized Formulation of Linear Regression with Basis Functions	329

13.3.1	Basis Functions	329
13.3.2	Hypothesis with Arbitrary Basis Functions	330
13.3.3	Extended Feature Matrix with Arbitrary Basis Functions	332
13.3.4	Derivation of Cost Function, Cost Gradient, and the Closed Form Solution	332
13.3.5	Numerical Aspects and Discussion	333
13.4	Formulation of Chosen Cases in Terms of Basis Functions	334
13.4.1	Simple Linear Regression	335
13.4.2	Multiple Linear Regression	335
13.4.3	Polynomial Regression	336
13.4.4	Multiple Polynomial Regression	339
13.4.5	Discussion of Polynomial Regression	341
13.5	Semi- and Non-parametric Regression	342
13.5.1	Problem Formulation	342
13.5.2	Regression with Piecewise Polynomials	343
13.5.3	Piecewise Regression Splines	346
13.5.4	Kernel Regression with Gaussian Basis Functions	348
13.5.5	Kernel Regression with FOURIER Basis Functions	353
13.5.6	Other Commonly Used Basis Functions	353
13.6	Further Nonlinear Regression Models	354
13.6.1	Ridge- and LASSO Regression	354
13.6.2	K-Nearest Neighbors Regression	355
13.6.3	Support Vector Machine Regression	355
13.6.4	Artificial Neural Networks	355
13.7	Summary and Conclusion	355
13.8	Exercises	356
	References	357
14	Supervised Classification	359
14.1	Introduction to Supervised Classification	359
14.2	Rule-Based Classification Methods and Decision Trees	361
14.2.1	The Idea Behind Classification Rules	361
14.2.2	The Relation of Rule-Based Methods and Decision Trees	364
14.3	Notions and Concepts for Classification Problems	366
14.3.1	Datasets, Models, and the Hypothesis Class	366
14.4	Nearest Neighbor Classifier	369
14.4.1	Distance Measures	369
14.4.2	The (First- or One-) Nearest Neighbor Method	371
14.4.3	The General k -Nearest Neighbor Method	373
14.4.4	Multi-class Classification	375
14.4.5	Implementation	377
14.5	Gaussian Naive Bayes Model	378
14.5.1	The Class of Naive Bayes Models	379

14.5.2	Maximum-Likelihood Estimation for the Gaussian Naive Bayes Model	380
14.5.3	Algorithm and Python Implementation	381
14.6	Support Vector Machines	384
14.6.1	Linear Support Vector Machine	384
14.6.2	Nonlinear Support Vector Machine	386
14.7	Exercises	388
	References	388
15	Unsupervised Learning	389
15.1	Introduction to Dimensionality Reduction	389
15.1.1	Example	389
15.1.2	The extrinsic and intrinsic dimensionality	390
15.1.3	Notations and Conventions	390
15.2	Principal Component Analysis: Theoretical Background and Derivations	391
15.2.1	A Simple Motivation from Solid Mechanics	392
15.2.2	Further Examples of “Higher”-Dimensional Data Sets	393
15.2.3	The Algorithm of Principal Component Analysis in Brief	395
15.2.4	Preparation: Orthogonal Projection of Data onto a Line	395
15.2.5	Maximizing the Variance along the First Principal Direction	402
15.2.6	The Second Principal Direction	405
15.2.7	The k -th and the Best k Principal Directions	407
15.3	Application Aspects and Examples of PCA	409
15.3.1	Image Data and the Principal Component Analysis	409
15.3.2	Accuracy of the Representation	415
15.3.3	Reconstruction	417
15.4	Further Methods for Dimensionality Reduction	420
15.4.1	The Relatives of Principal Component Analysis	420
15.4.2	Manifold Learning Methods	420
15.5	Clustering	424
15.5.1	k-Means and k-Medians	424
15.5.2	Gaussian Mixture Model	425
15.5.3	Model Selection with the BIC Criterium	428
15.5.4	Density-Based Methods: DBSCAN	429
15.6	Materials Science Examples	430
15.6.1	Dimensionality Reduction for Microstructures from the ISING Model	430
15.6.2	Dimensionality Reduction for Microstructures from the CAHN-HILLIARD Model	432
15.6.3	Clustering of Indentation Data	433
15.7	Exercises	435
	References	435

16	Machine Learning Techniques	437
16.1	Feature Engineering and Feature Importance	437
16.1.1	Feature Engineering	437
16.1.2	Permutation Feature Importance	439
16.2	Data Splitting, Cross-Validation, and Statistical Resampling	440
16.2.1	The Difference Between the Testing and Validation Dataset	441
16.2.2	The Holdout Method—Or the Train/Test Split	442
16.2.3	Leave-One-Out Cross-Validation	446
16.2.4	k -Fold Cross-Validation	447
16.2.5	Repeated k -Fold Cross-Validation	448
16.2.6	Stratified k -Fold Cross-Validation	449
16.2.7	Bootstrap	450
16.3	Baseline Models	452
	References	453

Part IV Artificial Neural Networks and Deep Learning

17	From the Perceptron to Artificial Neural Networks	457
17.1	A First Model of a Neuron	457
17.1.1	The Biological Neuron	457
17.1.2	Introducing... the McCulloch-Pitts Model	459
17.1.3	Boolean Decision Problems	461
17.2	The Rosenblatt Perceptron	463
17.2.1	The Mathematical Model of the Rosenblatt Perceptron	463
17.2.2	Again: Boolean Decision Problems	466
17.2.3	Geometric Interpretation	467
17.2.4	Supervised Learning with the “Perceptron Learning Rule”	469
17.2.5	Geometric Interpretation of the Perceptron Training	474
17.3	The ADALINE Model	476
17.3.1	ADALINE Training Strategy	476
17.3.2	Example: Comparison Between the Perceptron and the ADALINE Model	478
17.3.3	The Relation to Gradient Descent Optimization	479
17.3.4	Limitations of the Perceptron and the ADALINE Model	480
17.4	Increasing the Complexity: Assemblies of Neurons	481
17.4.1	A First Network of Two Computational Neurons	481
17.4.2	The Boolean XOR Decision Problem	484
17.4.3	Python Implementation of the Two-Neuron-Assembly	485
17.4.4	A Fully Connected Network with Input, Output and a Single Hidden Layer	487
17.4.5	“Vectorized” Operations: The Equations for the Network with a Hidden Layer	490

17.4.6	Capabilities of the Networks for Classification and Regression	491
17.5	Summary and Historical Remarks	492
17.6	Exercises	493
	References	494
18	A Gentle Introduction to Deep Learning	497
18.1	Overview of the Historical Developments	497
18.2	Activation Functions	498
18.2.1	Step Function	498
18.2.2	Linear Function	498
18.2.3	Sigmoidal Function	500
18.2.4	The Family of ReLU Functions	500
18.2.5	Hyperbolic Tangent Function	501
18.3	Back-Propagation: Introduction and Example	501
18.3.1	The General Concept	501
18.3.2	A First Example	501
18.3.3	A Complete Python Implementation	506
18.4	General Formulation of Backpropagation	510
18.4.1	Forward Pass	510
18.4.2	Backward Pass	512
18.5	Python Implementation and Example for the Fully Connected Network	516
18.5.1	Python Implementation	517
18.5.2	Application: Identification of Handwritten Digits	521
18.6	Further Concepts and Techniques	523
18.6.1	Mini-Batches	523
18.6.2	Batch Normalization	524
18.6.3	Weight Initialization	525
18.7	Less Is More: The Concept of Dropout	527
18.8	Example: Microstructure Classification and Property Prediction	528
18.9	Exercises	530
	References	532
19	Advanced Deep Learning Architectures and Techniques	533
19.1	Convolutional Neural Networks	533
19.1.1	Motivation of the Architecture	533
19.1.2	Convolutional Layer	534
19.1.3	The Pooling Layer	538
19.1.4	The Receptive Field and Dilated Convolutions	539
19.1.5	Feed Forward Step and Backpropagation	540
19.1.6	CNN Architectures	540
19.2	Deep Learning Techniques	543
19.2.1	Optimization Methods	543
19.2.2	Momentum	544
19.2.3	Early Stopping	544

19.2.4	Transfer Learning	544
19.2.5	Data Augmentation	544
19.3	Two Examples for Deep Learning in Microscopy	545
19.3.1	Semantic Segmentation of TEM Images of Nanoparticles	545
19.3.2	Binary Segmentation Using Synthetic Training Data for Grain Microstructures	546
19.4	Autoencoder: How to Learn with Networks Without Supervision	548
19.4.1	Single-Layer and Multilayer Architectures	548
19.4.2	The Relation to Principal Component Analysis	549
19.4.3	Convolutional Autoencoder	550
19.5	Generative Adversarial Networks: How to Create Data?	550
19.5.1	Model	550
19.5.2	Training	551
19.5.3	Loss Functions for the Min-Max Game	551
19.5.4	Common Problems of GANs	554
19.5.5	Alternative Architectures and Variations	555
19.5.6	Python Implementation	556
19.5.7	Two Examples: Handwritten Digits and Microstructures	560
19.6	Physics-Informed Machine Learning and Beyond	562
19.7	Summary and Conclusion	563
19.8	Exercises	564
	References	565
A	Linear Algebra for Machine Learning	569
A.1	Vector Calculus	570
A.1.1	Definition of Vectors	570
A.1.2	Conventions: Row and Column Vectors	570
A.1.3	Transposition of a Vector	571
A.1.4	Scalar Product of Vectors	571
A.1.5	Orthogonal Vectors	572
A.1.6	Euclidean Norm of a Vector	572
A.1.7	Euclidean Distance	573
A.1.8	Orthogonal Projection of a Vector	573
A.1.9	Gradient of a Scalar Field and a Vector Field	574
A.2	Matrices and Matrix Operations	575
A.2.1	Basic Definition and Creation of Matrices in Python ..	575
A.2.2	Addition of Matrices	576
A.2.3	Multiplication of a Matrix with a Scalar	577
A.2.4	Matrix Product	577
A.2.5	Multiplication of a Matrix with a Vector	579
A.2.6	Hadamard Product of Matrices	580
A.2.7	Transposition of a Matrix	581

A.2.8	Neutral Element for Matrix Addition	582
A.2.9	Neutral Element for Matrix Multiplication	582
A.2.10	Square Matrix	582
A.2.11	Inverse Matrix	583
A.2.12	Symmetric Matrix	584
A.2.13	Triangular Matrices	585
A.2.14	Trace of a Matrix	585
A.2.15	Rank of a Matrix	586
A.2.16	Determinant	587
A.2.17	Orthogonal Matrix	588
A.3	Derived Properties, Further Theorems, and Advanced Definitions	589
A.3.1	Eigenvalues, Eigenvectors, and the Characteristic Polynomial	589
A.3.2	Diagonal Matrices and Diagonalization of Matrices ...	592
A.3.3	Similarity Transformations	594
A.3.4	The Spectral Theorem	594
A.3.5	Eigendecomposition and Singular Value Decomposition	596
A.4	Matrix-Related Operation That Only Exists in Numpy	597
A.4.1	Addition of a Scalar to a Matrix	597
A.4.2	One-Dimensional vs. Two-Dimensional “Vectors”	598
B	Proofs and Derivations	601
B.1	Proofs of the Theorems About Expectation Values	601
B.2	Proofs of Some Theorems About Variances	602
B.3	Simplification of Pearson Moment Coefficient of Skewness	603
B.4	Proofs and Additional Derivations for Distributions	604
B.4.1	Property of the Binomial Coefficient	604
B.4.2	Binomial Distribution: Derivation of Mean and Variance	605
B.4.3	Isocontour Lines of a Bivariate Normal Distribution ..	606
References		608
	List of Python Listings	609
	Index	611

Acronyms

The page number indicates only the first occurrence of the acronym.

ADALINE	Adaptive Linear Neuron.....	476
AE	absolute error	472
AI	artificial intelligence.....	15
AIC	Akaike information criterion.....	429
AL	activation layer	510
ANN	artificial neural network	534
BIC	Bayesian information criterion.....	429
CDF	cumulative distribution function	116
CLT	central limit theorem	163
CV	coefficient of variation	261
CNN	convolutional neural network	533
DL	deep learning.....	15
DoE	design of experiment	28
EDA	exploratory data analysis	152
EBSD	electron backscatter diffraction	115
FEA	finite element analysis.....	44
FEM	finite element method	59
FCL	fully connected layer	510
GAN	generative adversarial network.....	550
GD	gradient descent	504
GMM	Gaussian mixture model	425
GNB	Gaussian naive Bayes	378
IQR	interquartile range	200
SGD	stochastic gradient descent.....	543
iID	independent and identically distributed	137
IoU	intersection over union	264
<i>k</i>-fold CV	<i>k</i> -fold cross-validation	447
KDE	kernel density estimation.....	202
kNN	<i>k</i> -nearest neighbors.....	242
LLN	law of large numbers	161
LOO CV	leave-one-out cross-validation	446
LSQ	least square.....	274

MAE	mean absolute error.....	259
MCP	McCulloch-Pitts	459
MD	molecular dynamics	44
ML	machine learning	15
MLP	multilayer perceptron.....	487
MSD	mean squared deviation	259
MSE	mean squared error	259
OLS	ordinary least squares	327
OHE	one-hot encoding	438
PCA	principal component analysis	169
PC	principal component.....	395
PDF	probability density function.....	109
PDE	partial differential equation	562
PMF	probability mass function	109
PINN	physics-informed neural network.....	563
ReLU	rectified linear unit	500
RMSE	root mean squared error	260
MSLE	root mean squared logarithmic error.....	261
RBF	radial basis function	349
SEM	scanning electron microscopy	546
SVD	singular value decomposition	392
SVM	support vector machine.....	355
SRS	simple random sampling	155
TEM	transmission electron microscopy	21
t-SNE	t-distributed stochastic neighbor embedding.....	421
RV	random variable	134
UMAP	Uniform Manifold Approximation and Projection.....	421
TLU	Threshold Logic Unit.....	459

Part I

Introduction and Foundations

A Brief History of Data and Data Science

1

Equipped with his five senses, man explores the universe around him and calls the adventure Science.

Edwin Hubble (1889–1963),
American astronomer

1.1 Where Do All the Numbers Come From?

Natural science and engineering in general would not be possible without numbers and without the ability to count (although in the discussions of where statistics end and data science begins, it is claimed that data science is different because it does not only work with numerical data, but that is a different story). Counting, and in particular using our fingers as “*the earliest calculating machine*” [5], has been part of humanity for more than 40,000 years. However, the first written records date back to about 10,000 BCE.¹

Beginning around 3,400 BCE, the Sumerians introduced the first known number system, in which numerals were imprinted on clay tablets. They also attached a version of what we would call today *metadata* or *physical units*: small symbols that represented the quantity being counted (e.g., a sheep). Figure 1.1 shows one of the earliest “data visualization” of this number system. Around 2,000 BCE the Babylonians developed a first decimal number system and then already calculated a first approximation of $\pi \approx 3.125$. Around 300 BCE they then also invented the *abacus* as the first calculating tool. While the Roman numerals appeared as early as 1,000 BCE, Europe had to wait another two centuries before the Indo–Arabic

¹ BCE is the abbreviation for *Before Common Era* and CE is the abbreviation for *Common Era*. These are year notations according to the Gregorian calendar. CE and BCE are identical to *Before Christ* (BC) and *Anno Domini* (AD) but do not have their religious connotation.

Fig. 1.1 One of the first data visualization: summary account of silver for the governor written in Sumerian Cuneiform on a clay tablet. (From Shuruppak or Abu Salabikh, Iraq, circa 2,500 BCE. British Museum, London. BM 15826, courtesy Gavin Collins [10])



numeral system (which we still use today) was finally introduced. Of course, there are many more exciting events such as the invention of the number “zero” and the first binary numeral system, but we will now go ahead and take a look at one of the most important uses of numbers in ancient history besides trade and commerce: astronomy and predicting the path of the celestial objects in the sky.

1.2 The Ancient Roots of Data Science

When we talk about the “ancient roots of data science” in this section, we do not mean to imply that, for example, the ancient Babylonians did anything even close to data science. But it is clear that data science would not exist without the historical events presented below. Figure 1.2 provides an overview of the stages in history that have played a crucial role in the development of modern science in general and that have been of particular importance to data science. We begin with the ancient Babylonians, who already demonstrated impressive creativity and inventiveness above.

1.2.1 Ancient Babylonia (1800–600 BCE)

When the Babylonians observed the night sky, they recognized two types of celestial objects. The first type are stars that move around in a circular pattern (it will be seen that they move around Polaris, the North Star) and can “twinkle.” The photo on the left in Fig. 1.3 illustrates these circular orbits. The second type of celestial objects are those that move in a more complex manner. This is known today as *retrograde motion* (cf. Fig. 1.3 right). This complicated motion gave them the name “wandering stars,” which in Greek means “planetes.” The Babylonians counted five such planets: Mercury, Venus, Mars, Jupiter, and Saturn. Together with the Sun and the Moon, this gave seven objects, one for each day of the week, as they introduced at that time. Since the Babylonians could not explain the movement of the planets in any other way, they assumed that the planets must be pulled through the sky by gods. The stars, on the other hand, were fixed on a hemispherical “shell” that

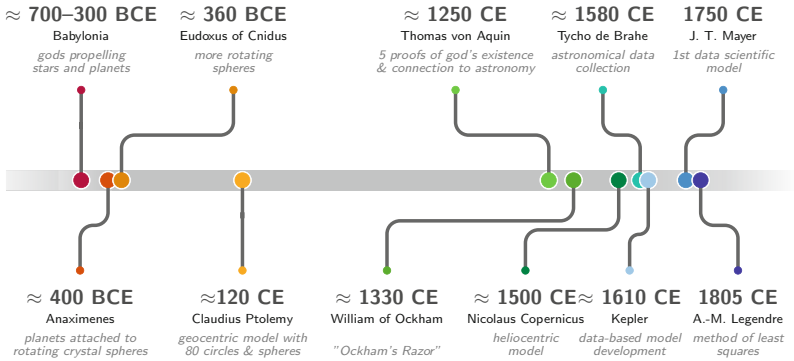


Fig. 1.2 Overview of important historical developments and concepts (up to ≈1800) that contributed toward shaping today's data science



Fig. 1.3 Stars and planets: long-term exposure photograph, made during a complete night, has captured the star trails around the south celestial pole, courtesy Kwon O Chul. The “temporal averages” reveal the apparent motion of stars on circular paths (left). Superposition of snapshots at equidistant time steps (right) reveals the complex forward-backward motion of planets called “retrograde motion” (shown is a photo-montage of the motion of Mars)

constantly rotated to the east. Obviously, they could not properly explain the forces acting between the planets, but their *model* still worked quite well: the goal of the model was to create calendars that could provide information about when it was best to plant, harvest, or go to war [8]. Obviously, any model necessarily contains simplifications and idealizations, and therefore all models are flawed in some of their aspects; the real question, however, is whether the model performs well at the (predictive) task for which it was developed,² and in this respect, the Babylonian model worked perfectly.

² The question of model performance is also a question that we will discuss in the context of machine learning many times.

1.2.2 The Age of Classical Antiquity (≈750 BCE–600 CE)

After the fall of Babylon in 539 BCE, the Babylonian Empire ceased to exist and much of its knowledge was at risk. Fortunately, some of their astronomical knowledge had already reached Greece. In particular, ANAXIMENES (approx. 585–525 BCE) based his work on the Babylonian cosmology. His approach removed the gods from the picture and introduced instead a number of *concentric half-spheres* spanning the flat “earth disk,” with which the stars and planets moved. Thus his celestial mechanics was the first purely physics-based model. To explain why no one could see the spheres, he also invented the invisible, crystal-like element *aether*, which accompanied science for more than a millennium. However, unlike the Babylonians, he could not explain the movement of the planets. The “remedy” was provided by EUDOXUS OF CNIDUS (approx. 408–355 BCE), a student of PLATO. He added more rotating spheres: three for the Sun and for the Moon and four for each of the five visible planets, which could explain the retrograde motion in parts by superposing the motion on two different spheres. In total, this model required 27 spheres. But this was not the last word: ARISTOTLE (384–322 BCE) extended the model to 56 spheres, because this was the only way to explain the brightness variations of the stars. In the context of machine learning, this model would have the taint of *overfitting*.

Unfortunately, even this model could not explain all the observed phenomena, including the fact that the Moon goes through different *lunar phases* of waning and waxing. It took several centuries before the Roman CLAUDIUS PTOLEMY (approx. 100–170 CE) arrived with new ideas (even though some of them had already been mentioned by the ancient Greeks). In his model, the planets were attached to some kind of “celestial wheel,” which in turn was attached to the spheres. In addition, the Earth was placed in a slightly eccentric position so that the planets could move at different velocities. This very complex geocentric model required about 80 circles, spheres, and other geometrical parameters. However, it was extremely accurate and could predict, among other things, the occurrence of eclipses. Figure 1.4a gives an idea of the complex motion of the planets resulting from this model. PTOLEMY also provided astronomical tables and other tools to easily compute the positions of the Moon, Sun, and the other planets. For the next 1,000+ years, this was the state-of-the-art despite the horrendous complexity and accumulation of completely wrong, (un)physical assumptions.

1.2.3 The Medieval Period (approx. 500–1500 CE)

The Middle Ages were a dark time for the European people and for science. Famines, war, and the plague were responsible for greatly reducing the population, including scientists. Invasions and mass migrations created an environment in which science struggled to thrive. THOMAS AQUINAS (1225–1274) is one of the relatively few brilliant minds worth mentioning: he was a modern thinker who tried to

reconcile astronomy and theology. As a first step, he stated five different proofs for the existence of God [8]. This says something about that era, but actually it is more interesting to know that they were all refuted by WILLIAM OF OCCAM. He and the so-called *Occam's razor* play a special role in data science and will be explained in more detail below.

1.2.4 The Renaissance (approx. 1500–1700 CE)

NICOLAUS COPERNICUS (1473–1543) rejected the model of PTOLEMY and could not bring himself to believe that the universe worked that way. He radically changed all of astronomy by placing the Sun at the center of the model of the universe (cf. Fig. 1.4b). This was such a pivotal event that it subsequently became known as the *Copernican Revolution* (although the Greek ARISTARCHOS OF SAMOS had already proposed a heliocentric system around 250 BCE). COPERNICUS' special contribution was to represent his idea in the form of a mathematical description. Even though the predictions of his model were not much more accurate than those of PTOLEMY'S model, his model was much simpler; his model was the first to

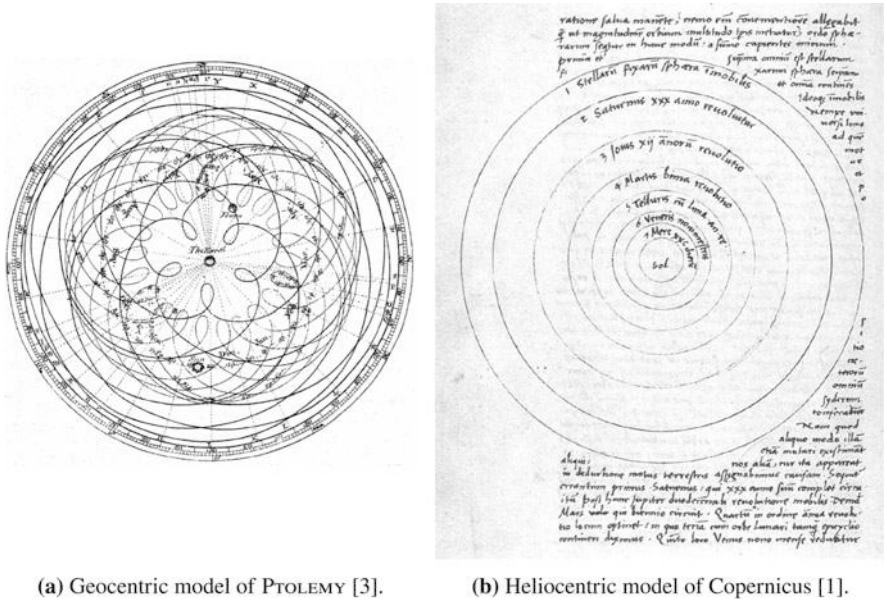


Fig. 1.4 (a) Ptolemy's prediction of the apparent movement of the Sun, Mercury, and Venus as observed from Earth. The inner pentagram-like shape is part of the orbit of Venus. (b) Copernicus' heliocentric model with the Sun in the center from the manuscript "De revolutionibus," Book One, Chapter 10: The Order of the Heavenly Spheres. Both models are comparable in terms of predictive accuracy, but the heliocentric model is significantly simpler

consistently *explain* (and not just *describe*) the retrograde motion of the planets. All this was the foundation for a new epoch of science.

TYCHO DE BRAHE (1546–1601) was a diligent astronomer who spent much of his time collecting data from observing the stars. Coincidentally, he also observed two truly unique events: a nova in 1572 and a comet in 1576. These “outliers” would seem to contradict all that was believed or known about the universe, especially the concept of crystal spheres, since the comet was able to cross the orbit of the planets.

Finally, his later assistant JOHANNES KEPLER (1571–1630) was one of the most interesting personalities in the history of science, with unconventional creativity, spirituality, and accuracy. He admired the simplicity of COPERNICUS’ model and refined it by introducing the trajectories of the planets as ellipses; his three “Kepler laws” are very simple mathematical expressions and are still valid today. Eventually, KEPLER obtained much of the data BRAHE had collected during his lifetime. KEPLER then began to analyze the data from the observed orbit of Mars, and he did so for an impressive 25 years. He compared the data to the predictions of the theoretical models of PTOLEMY and COPERNICUS and found significant deviations. Because of his appreciation of TYCHO BRAHE’s data, this eventually led to a new model in which he introduced the ellipsoid orbits of the planets.

This is a good time to pause our historical tour for a moment and recognize that a defining point was reached with KEPLER. And why? Because after 25 years of thorough data analysis, KEPLER should in some ways be regarded as the first data analyst [7]: never before in human history has there been anyone who analyzed a massive dataset over and over again, just for the sake of analyzing it, but also to reveal structure in the data and identify parameters by comparing the data to mathematical hypothesis equations. KEPLER was motivated by the belief that even if the overall phenomenology of the motion of the stars and planets looks extremely complicated, it is *simple* laws and *simple* principles that have a divine aesthetic and that form the mechanics behind them. This is related to another principle that is important in data science today and by which KEPLER was most likely already influenced: the principle of *Occam’s razor*, which takes us back to the fourteenth century.

1.2.5 OCCAM’S RAZOR

WILLIAM OF OCCAM (also spelled OCKHAM) lived from 1285 to 1347 and was an English Franciscan monk as well as a philosopher and logician. He became famous for his theo(logical) arguments, which he used to refute THOMAS AQUINAS’s “proofs” of God [8] despite his own religious beliefs. His particular style of logical argumentation also brought him some fame. This involved, in particular, a principle known today as OCCAM’s razor. He did not “invent” this principle, but OCCAM used it on several occasions.

Occam’s razor is a principle also known as the *principle of parsimony*. It states that if two theories explain a given phenomenon, we should rather choose the one based on the smallest possible number of “parameters,” i.e., the simplest one. A

popular alternative formulation states that the simplest solution is best or most probable. Although Occam's razor is not a universal rule but rather a principle, many scientists (including statisticians who make BAYESian inferences) are convinced that this principle is something fundamental.

What does this have to do with the complexity of models? It is clear that a very sophisticated model can represent a more complex physical behavior than a very simple model. Take the heliocentric system as an example: the number of parameters is much smaller than those of the PTOLEMYian model. However, if we fitted the model parameters to measured data, the heliocentric model achieves much higher accuracy because more data are available *per parameter*. We will encounter this principle in a number of different situations. In the next chapter, we will encounter similar points when we discuss the *curse of dimensionality*.

Nevertheless, it is only a principle that still needs to be supported by logic and empirical evidence. WILLIAM OF OCCAM, for example, focused more on the meaning of *not making a theory more complicated than necessary*. For KEPLER, however, this principle certainly helped to shape his intuition and to take the bold step of turning the universe almost “inside out.”

1.3 The First True Data Scientist

One could argue whether KEPLER really was the first data scientist. However, there is another person who is certainly considered the first “real” data scientist: a person, who used data in a way that is very close to the approach of today's data scientists. It was the German astronomer JOHANN TOBIAS MAYER (1723–1762)—yet another astronomer! His work was mainly devoted to the study of the Moon and in particular its “libration” (a kind of “waving” behavior of the Moon: the Moon always faces the Earth with the same face, but small fluctuations occur—the libration). He became famous for his *Lunar Tables* and an accurate map of the Moon [14]. To study the libration, he chose a point on the Moon's surface, measured it, and used a trigonometric equation that depended on three unknown parameters α , β , and θ in addition to the measured position. (see [11] for further details).

It would have taken only three measurements on three different days, resulting in three individual equations, to approximate the free parameters. However, MAYER had measurements from a total of 27 days and therefore had to come up with a suitable strategy, as he did not want to throw away valuable data. At the time, there was no theory for dealing with such overdetermined systems of equations [4]. MAYER's idea was to divide the 27 equations into 3 groups of 9 equations. Then he added the equations of each group so that there were only three equations and three unknowns. Now how do you choose which of the 27 equations goes into which group? Since α was the most important parameter for the equation, he decided that in the first group, all the equations had the largest positive value for α . The second group consisted of the nine largest *negative* values of α , and the rest went into the third group. His intuition was to maximize the “contrast” of the coefficient α , which should result in a good estimate. He wrote: “[...] *because each*

of these equations has been formed in the most advantageous manner [...] through the above division into three classes, the difference between the three sums is made as large as possible.” He also provided an empirical evaluation of the accuracy; however, he believed that since there were nine observations for each parameter, the result should also be nine times more accurate, which is obviously false. However, it took several more years for PIERRE-SIMON LAPLACE and later CARL FRIEDRICH GAUSS to derive a correct error estimate.

MAYER’s work is particularly impressive given that the discovery of the *least squares method* was still a long way off in 1750 (it took until 1805 for ADRIEN-MARIE LEGENDRE to introduce this approach). MAYER had the great intuition at that time not to *cherry-pick* the data (e.g., by selecting the “best” three measurements), but to let the data “speak” for themselves by combining a set of equations and thus measurements. In a sense, MAYER realized what is still true for data science and machine learning today: the more data, the better—and the more data, the better the model. EULER worked on a similar problem but could not solve it because he approached the problem with the mind of a theoretician, while MAYER probably benefited from his engineering sense and practical intuition. His data-based method was completely new, was highly regarded by his colleagues, and greatly influenced later work.

1.4 The More Recent Roots of Data Science

The foundations of data science are old and resulted from the desire to predict phenomena and events and to obtain decision support, which has not changed significantly to this day. For this reason, data science is closely linked to the development of modern mathematics and statistics. In the next 250 or 300 years, starting in the second half of the eighteenth century, many exciting developments took place, of which we mention here only some of the most important aspects. A major step was ADRIEN-MARIE LEGENDRE’s conceptualization of the *method of least squares* in 1805, mentioned above. We will encounter his method in numerous places in this text. There were also a number of significant developments in the field of probability theory. One of the most important names was THOMAS BAYES, who worked around the middle of the nineteenth century and is known, among other things, for what is known as BAYES’s theorem. This theorem plays an important role in probabilistic machine learning methods. From the first half of the nineteenth century, probabilities and statistics were developed mainly as tools in the social sciences, e.g., for analyzing death rates or conviction rates for different geographical locations. The late nineteenth century is considered the cornerstone of modern statistics, when scientist such as FRANCIS GALTON benefited from the fact that experiments in the natural sciences were becoming more sophisticated and reproducible and could generate larger amounts of data. Another name we will encounter several times in this text is KARL PEARSON, who lived and worked until the 1930s and is responsible for, among other things, the *standard deviation* and the *correlation coefficient*. Not surprisingly, there are many more names and works

worth mentioning that we have skipped (see, e.g., the overview by Stigler [11] for more details). We now fast-forward a few decades and jump directly to the beginnings of the digital age.

1.5 Data Science and Machine Learning in the Twentieth Century

The early 1960s marked another turning point in science and technology that also gave significant impetus to the development of data science. Around 1960, the transition from vacuum tubes to transistors and integrated circuits in computers occurred. Shortly before 1960, magnetic tape drives were developed, and computer memory chips (RAM) also became available shortly after 1960. The ability to process and store larger amounts of data triggered a whole new scientific discipline centered around “data.”

These new developments began in 1962, when JOHN W. TUKEY [13] published an article entitled “The future of data analysis.” Therein, he defined the notion of data analysis pretty much like what is also called data science today:

[...] procedures for analyzing data, techniques for interpreting the results of such procedures, ways of planning the gathering of data to make its analysis easier [...] and all the machinery and results of (mathematical) statistics which apply to analyzing data.

Also in the mid-1960s, PETER NAUR created the term *datalogy* as the *science of data processes* [12], but it did not catch on. A few years later, in 1974, NAUR, in his book “Concise Survey of Computer Methods” [9], referred to data science as a “science of dealing with data.”

In 1985, the term data science was coined by C.-F. JEFF WU [15] who proposed it as an alternative name for statistics. It is said that in 1992, during the second Japanese-French statistics symposium at the University of Montpellier in France, the name data science was generally accepted, established, and from then on understood as a new discipline. This was also the beginning of a long discussion about whether and to what extent data science is just another name for statistics or whether one does not need statistics at all for data science (see, e.g., the overview in [2])—a discussion that is even today not entirely finished.

Closely related to data science is also the term *data mining*: this term first appeared in 1999 when the article “Mining Data for Nuggets of Knowledge” announced a data mining “short Executive Education course,” organized by JACOB ZAHAVI [6]. There, it was noted that traditional statistical methods work with small datasets, but data mining requires specialized tools for large datasets. Data mining became another term for the process of identifying structure and

patterns in relatively large datasets, and this definition is still valid today, with the availability of increasingly powerful machine learning methods that complement “conventional” analysis methods.

With the advent of advanced machine learning methods and affordable high-performance CPU/GPU computing infrastructures, and with the availability of software libraries that have made sophisticated machine learning and data mining methods accessible to virtually everyone (what is sometimes called the “democratization of AI”), the field of data science gained even more momentum and brought us to where we are today.

1.6 Summary and Conclusion

This concludes the brief and perhaps somewhat unusual introduction of the historical roots and the development of data science. Clearly, data science is a “young” disciplines but at the same time has a long history and a multitude of links to different scientific fields. We will now continue by introducing a number of different concepts from data science and information science while explaining their connections to materials science and engineering.

References

1. N. Copernicus. “De revolutionibus”, p. 9 verso. Source: <https://en.wikipedia.org>. URL https://commons.wikimedia.org/wiki/File:De_Revolutionibus_manuscript_p9b.jpg.
2. D. Donoho. 50 Years of Data Science. *Journal of Computational and Graphical Statistics*, 26(4):745–766, Oct. 2017. DOI <https://doi.org/10.1080/10618600.2017.1384734>.
3. J. Ferguson. Representation of the apparent motion of the sun and planets from the earth. Taken from the “Astronomy” article in the first edition of Encyclopaedia Britannica (1771). Source: <https://en.wikipedia.org>. URL https://en.wikipedia.org/wiki/File:Cassini_apparent.jpg.
4. E. Gray Forbes. The Life and Work of Tobias Mayer (1723–62). *Quarterly Journal of the Royal Astronomical Society*, 8:227, Sept. 1967.
5. G. Ifrah. *The Universal History of Numbers: From Prehistory to the Invention of the Computer*. John Wiley & Sons, Inc., 2000.
6. Mining Data for Nuggets of Knowledge — knowledge.wharton.upenn.edu. URL <https://knowledge.wharton.upenn.edu/article/mining-data-for-nuggets-of-knowledge/>, [Accessed 25-01-2023]
7. A. E. Kossovsky. *Johannes Kepler—The First Data Analyst*, pages 23–24. Springer International Publishing, Cham, 2020. ISBN 978-3-030-51744-1. DOI https://doi.org/10.1007/978-3-030-51744-1_7.
8. J. McFadden. *Life is simple*. Basic Books UK, 2021. ISBN 9781529364958.
9. P. Naur. *Concise Survey of Computer Methods*. Petrocelli, 1975. ISBN 0-88405-314-8.
10. Gavin Collins (photographer). Sumerian account of silver for the governor (background removed). British museum, London. BM 15826. Source: <https://en.wikipedia.org>. URL [https://commons.wikimedia.org/wiki/File:Sumerian_account_of_silver_for_the_governor_\(background_removed\).png](https://commons.wikimedia.org/wiki/File:Sumerian_account_of_silver_for_the_governor_(background_removed).png).
11. S. M. Stigler. *The History of Statistics. The Measurement of Uncertainty before 1900*. The Belknap Press of Harvard University, 1986. ISBN 9780674403413.

12. E. Sveinsdottir and E. Frøkjær. Datalogy — the Copenhagen tradition of computer science. *BIT*, 28(3):450–472, Sept. 1988. DOI <https://doi.org/10.1007/bf01941128>.
13. J. W. Tukey. The Future of Data Analysis. *The Annals of Mathematical Statistics*, 33(1):1–67, 1962. DOI <https://doi.org/10.1214/aoms/1177704711>.
14. E. A. Whitaker. *Mapping and Naming the Moon—A History of Lunar Cartography and Nomenclature*. Cambridge University Press, 2003. ISBN 9780521544146.
15. C.-F. J. Wu. Future directions of statistical research in China: a historical perspective. *Application of Statistics and Management*, 1, 1986.

*Data is not information, information is not knowledge,
knowledge is not understanding, understanding is not wisdom.*

Clifford Stoll (born 1950),

American astronomer, author and teacher

2.1 What Is Data Science and How Is It Related to Machine Learning and Artificial Intelligence?

The previous chapter did not attempt to define what data science is. In the following, we will choose a very intuitive definition and also clarify some common misconceptions about *artificial intelligence* (AI).

2.1.1 Data Science

Most of us would probably agree that data science consists largely of statistics, computer science, machine learning, data processing, and visualization (see, e.g., the overview in [8]). But how does all of this connect, and where does AI come into play? Figure 2.1 shows a visualization of all the different fields that contribute to data science. We can see that data science consists of contributions from many different fields. First of all, data science does not necessarily mean that *machine learning* (ML) or *deep learning* (DL) methods have to be used. These are just some (but admittedly, some of the most successful) methods. However, the roots of data science lie in statistics, which has been accompanied by explorative data analysis for several decades. Both are introduced in the first part of this text.

Last but not least, approaches and tools from computer science and programming are the bread and butter for data science, as the gray box in Fig. 2.1 shows. While it might well be that “programming” as we know and use it today will change

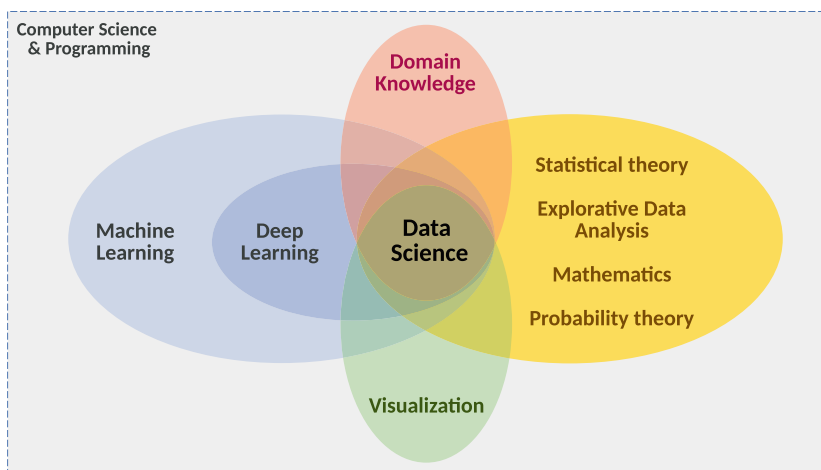


Fig. 2.1 Data science is at the intersection of a number of different disciplines, all more or less loosely connected by topics in computer science and programming (symbolized by the gray background). An important aspect of data science is its connection to domain knowledge, which in this text is the point at which knowledge from physics, materials science, and engineering integrated

completely in the near future—either through systems like ChatGPT [25] or through graphical programming approaches—writing and reusing code and libraries is at present probably the best and most instructive way to solve problems using data science approaches.

What we do *not* see in this sketch is **AI**. Does that mean that **AI** has nothing to do with data science or even **ML**? We answer this question in the following.

2.1.2 The Relation to Artificial Intelligence

The relationship between the three buzzwords *artificial intelligence*, *machine learning*, and *deep learning* probably needs some additional explanation. These terms are everywhere today, very often they are used more or less interchangeably today, and **AI** seems to be quite useful for marketing purposes. However, there is a clear distinction, which will be discussed below.

AI methods are about mimicking human behavior, which requires certain cognitive skills such as reasoning and logic. In addition, it is generally agreed that an important component of **AI** is also the ability to interact with the environment (see, e.g., the seminal text by Russell and Norvig [31]). “Interacting” encompasses perceiving the environment and acting on the environment. **AI** does not automatically imply that the program is intelligent in exactly the same sense as a human.

It is sufficient that it “looks like” it is solving the task in a way that would require intelligent behavior.

By contrast, **ML** focuses heavily on learning algorithms and learning from data examples. How is **ML** related to **AI**? Sometimes one comes across graphical representations that suggest that **ML** is a subset of **AI** (e.g., by showing three circles/sets where **AI** contains **ML**, which in turn contains **DL**, i.e., $\text{AI} \supset \text{ML} \supset \text{DL}$). Sometimes this is even understood in the sense that **ML** is a specialization of **AI**, which is clearly a misinterpretation. **ML** is one of the *means* of **AI** to achieve its goal. Reasoning and logic, while important disciplines of **AI**, are not part of **ML**.

Here are two examples of non-**ML** approaches that would nevertheless contribute to **AI**: one example is a rule-based system in which, for example, a hundred human experts define complex “if-then-else” clauses, enabling a system to make complex decisions that can easily be considered “intelligent.” Another example is genetic algorithms, which perform optimization tasks using algorithms that mimic evolutionary processes.

2.1.3 From Machine Learning to Deep Learning

DL is a further specialization of **ML** as it uses a particular type of **ML** models, namely, *deep neural networks* inspired by the structure and function of the brain. The attribute “deep” results from the fact that such a network contains many layers of artificial neurons, sometimes hundreds and more, as opposed to a *shallow neural network*, which contains only very few of such layers. Again, **DL** is one of the methods that contribute to an artificial intelligence “agent,” but it should not be called **AI**.

In this text, Part **III** is dedicated to the “classical” **ML** methods, which have their roots quite strongly in statistical analysis, while Part **IV** is dedicated to the introduction of **DL** methods that are rather inspired by the biological neurons.

2.1.4 Domain Knowledge

Another key “ingredient” to data science, shown in Fig. 2.1 and often neglected, is domain knowledge. Not only does data science require data; most of which comes from a specific application domain; also, many of the most interesting data science questions originate from a specific problem, such as a scientific or technological or societal problem, where data science is hoped to provide answers and new insights. Although a large number of “off-the-shelf” methods can be used, the really interesting problems are those where the methods need to be tailored to the specifics of the problem, and combining them with domain-specific insights is an ideal (and very fun) way to do this.

2.2 Data Science and Machine Learning in Materials Science and Engineering

Combining data science with domain-specific aspects from materials science leads us to define *materials data science* :

Definition 2.1 (Materials Data Science). Data science is positioned at the interface between machine learning and a large part of statistics and classical data analysis as well as data visualization. All this is additionally connected by methods from computer science and programming. The application of data science approaches to materials science problems or the tailoring of data-based solutions to the specifics of such problems is what *materials data science* is all about.

Such a close coupling between a domain science and data science can be observed in many other fields as well, such as bio data science or geo data science.

When the focus is not only on data analysis and data science aspects, but when also the technological aspects, e.g., compute or storage facilities, play a more important role, then one can rather refer to names such as bioinformatics, materials informatics, or geoinformatics. However, all these are still relatively young disciplines.

How and when did materials science and data science meet? There is no clear point in time when this happened; rather statistical methods and data visualization have always been present in some form, similar to many other engineering and natural science disciplines. However, there are contributions that clearly stand out, for example, by attempting to uncover hidden structures in data or fusing datasets from different sources and performing “meta-studies.”

The Definition of Machine Learning?!



But wait, there is a definition of *materials data science*, which is based on *machine learning*—where is the definition of *machine learning* (or *artificial intelligence*)? For now, we just have to go with our intuition and on what was said in Sect. 2.1.2. A proper definition will be given in Chap. 11.

2.2.1 Materials Data Science Without Machine Learning?

One of the best known examples in materials science are the *material property charts* , which have become an often used way of communicating what the current

state of, e.g., a class of alloys is in terms of their properties and how this new alloys might be superior to others. The material property charts are also known as *Ashby maps* [2, 3], named after MICHAEL ASHBY. These are scatterplots that plot two material properties (e.g., Young’s modulus and density) on two axes and *categorize* each point by specifying the material class (e.g., metals, polymers, ceramics) as the color of the point. In a next step, clusters of points belonging to the same material class are then grouped and visualized as a colored area, as shown in Fig. 2.2. The motivation of such a representation of material properties was described in [3] as follows: “the resulting charts are helpful in many ways. They condense a large body of information into a compact but accessible form; they reveal correlations between material properties that aid in checking and estimating data; and they lend themselves to a performance-optimizing technique, [...], which becomes the basic step of the selection procedure.”

This contains a number of interesting ideas: the charts are not just a representation of the raw data; rather, they attempt to “distill” and visualize certain patterns in the data. Guided by both the engineering knowledge and intuition, ASHBY has chosen the most important pairs of properties, e.g., Young’s modulus vs. density or strength vs. density, because in an engineering design process, the first quantity is usually to be maximized and the second one minimized. In the context of ML this is referred to as *feature engineering*. Additionally, this is supported by double logarithmic scaling of the axes to make details visible at different scales. Sometimes other information was also included, showing, e.g., guideline values for minimum

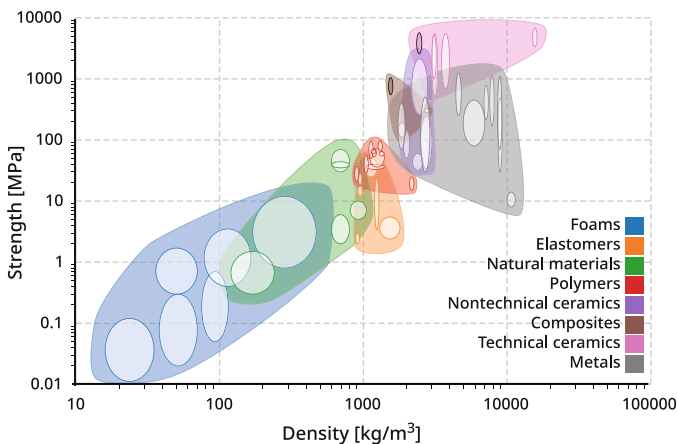


Fig. 2.2 Ashby map showing the location of different material classes in the density-strength space. The light-colored “bubbles” indicate subclasses, e.g., within the gray metals region there are the subclasses of, e.g., pure metals, super-alloys, or amorphous metals. (Adapted from [1] under the CC-BY 4.0 license, see⁴)

⁴ <https://creativecommons.org/licenses/by/4.0/>

weight design. The charts have a clear purpose: to facilitate the materials selection decision process. The Ashby maps are an excellent example of a data science approach that does not require ML.

2.2.2 A Brief Overview of Machine Learning in Materials Science

Nevertheless, ML methods have become increasingly important over the last three decades, and materials scientists have been using neural networks since at least the 1990s. One of the earliest publications is Ghaboussi et al. [16] with the title “Knowledge-based modeling of material behavior with neural networks.” Initially, neural networks were mainly used for simple optimization tasks and for “parameter fitting” or parameter optimization. However, progress was relatively slow, and it took until in the early 2000s for a number of conceptual and technical as well as societal changes to take place.

The awareness that data science can be a means to transform research data into even more valuable “knowledge” also went hand-in-hand with various “digitization aspects” of materials science, such as in the integration of data analysis, modeling, and experimentation [17, 26], and the increasing importance of the FAIR data principles⁵ for research data[40]. This is also reflected in the funding programs of many funding agencies and has led to new initiatives that address research data management and data analysis with the goal of initiating a non-incremental change in the way scientific research is conducted.

One such example is the *Materials Genome Initiative*,⁶ which also increased the awareness about the usefulness of materials informatics, *research data management*, and materials data science.

One of the first publications in this context by K. RAJAN [28] discussed the challenges and opportunities of materials informatics and advocated the goal of *knowledge discovery* by integrating high-quality materials science data, correlation analysis, and validation through established scientific theories and experiments. Not surprisingly, the field of high-throughput computational materials design, in particular, has been very active from the beginning in realizing large databases containing the computed thermodynamic and electronic properties of materials and also analyzing them with statistical and ML methods [9]. Another, still growing application field is machine learning of interatomic potentials for *atomistic simulations* [5]. Interatomic potentials govern the interaction forces between atoms. They are important components of atomistic simulations, and as the potential need to be evaluated for every atom, computational performance is an issue. ML potentials can produce high-quality and chemically accurate datasets faster (by several orders of

⁵ Data is denoted as “FAIR” if it is Findable, Accessible, Interoperable, and Reusable. This is seen as an important prerequisite for making research data machine-readable and machine-actionable, which could also be of great benefit to ML and data science.

⁶ <https://www.nist.gov/mgi>

magnitude) than density functional theory calculations, enabling the simulation of larger systems.

Mesoscale simulation methods and simulations of defects have also benefited from ML-based analyses [27, 32, 33]. The list of interesting computational materials science topics and problems addressed in the spirit of data science is, of course, incomplete and could easily be expanded; for other relevant reviews of materials informatics and materials data science in the context of computational materials science, see, e.g., [14, 29, 36, 39].

The *microscopy* community benefits equally from the developments in the field of data science, ML and DL. “Classical” data mining approaches are able to extract patterns from microscopy images, e.g., [11, 19] or a set of frames of movies acquired during an in situ *transmission electron microscopy* (TEM) experiment, for example, [20, 34, 42]. In the field of microscopy, DL has brought about great progress.

In particular, the development of the so-called *convolutional neural networks* and *generative adversarial networks* (all of which will be introduced in Chap. 19) has initiated an ongoing revolution in the field of image analysis. Applications range from segmentation of objects (such as atoms, defects, or grains, see, e.g., [18, 21, 37]), to image denoising and restoration of, e.g., diffraction patterns (see [4, 12] and references therein) to resolution enhancement [35] and feature tracking. All of this is becoming increasingly accessible to a wider audience [38]. Overviews are given in [15, 41].

2.2.3 Example: Linkage of Analysis Methods for Data Mining of Microscopy Images

An example of a more complex data analysis workflow for microscopy data is shown in Fig. 2.3. A SiC crystal was grown and a wafer was cut. The wafer was then treated with KOH (potassium hydroxide, which is a strong base) so that dislocations that penetrate the surface become visible as dark etch pits.

Interesting questions are as follows: What does the spatial distribution of dislocations look like? What is their character (e.g., are they threading edge, threading screw, or mixed dislocations)? What is their Burgers vector? What is the total number of different types of dislocations? Can the spatial structure be related to the growth conditions of the SiC crystal?

An X-ray topography scan of the entire wafer was performed in individual steps. Subsequently, the individual images of a complete wafer are automatically stitched together using the Python image processing library “openCV” [7]. From this image, smaller image sections were extracted containing the dark etch pits and were automatically classified as “good” and “bad” image sections (the bad ones contain multiple etch pits or other defects).

Etch pits look different depending on the dislocation character and the length of the Burgers vector. To automatically identify the different orientations of the hundreds of thousands of images, dimensionality reduction using *Uniform Manifold Approximation and Projection* (UMAP) [22] was performed. This method transforms

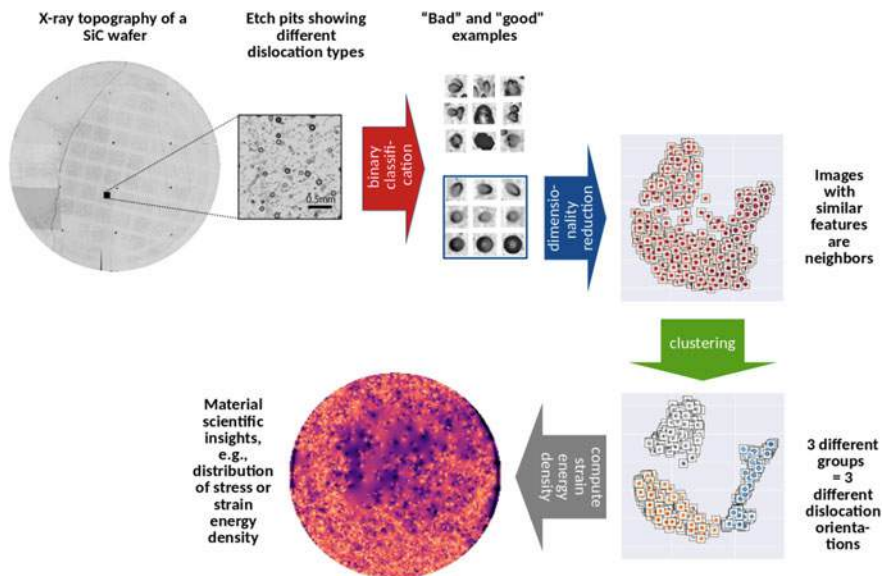


Fig. 2.3 Schematic of a data science workflow for extracting previously inaccessible information from microscopy images. Three different machine learning methods are connected. (Unpublished own work together with P. Wellmann and B. D. Nguyen [24])

the images into a different, two-dimensional space where images with similar “features” (whatever that means exactly!) are close to each other. Note that the two “coordinates” have no physical meaning. Next, the etch pits were divided into three groups (threading screw, edge, and mixed dislocations) by automatic clustering. In the figure, the coloring of the group elements indicates the orientation. This can now either be used to create a high-quality training dataset for *supervised learning* (where for each image the orientation is also known), or it can be used as input for further microstructural analysis. In Fig. 2.3 we show an example of computing the local strain energy density distribution for a given defect structure (taken from [23]).

This example illustrates a number of aspects. First of all, a solid command of classical computer vision methods (here used as preprocessor for stitching and cleaning up the images) is very handy and avoids manual work. Then the combination of different ML methods can be very useful. However, this requires a certain understanding of the individual methods. Last but not least, answering the domain-specific scientific questions to gain new knowledge is the goal and requires at least a basic understanding of the domain or good communication with the domain specialist in order to create this data analysis “pipeline.”

2.3 From Data and Information to Knowledge

In the previous sections, we have already used the terms *data*, *information*, and *knowledge* several times. What is commonly referred to as *data* plays a crucial role in all natural sciences and engineering, and *data science* even contains *data* in its name. Nevertheless, a clear and unambiguous definition is perhaps needed. We now begin with the information science perspective on data and information and then introduce the terms and definitions as they are used in this book.

2.3.1 “Data and Information” from the Point of View of Information Science

In information science, the terms and concepts of *data*, *information*, and *knowledge* play a fundamental role, but even there it is difficult to find a clear answer: for example, a recent overview article [43] lists no less than 130 definitions for these 3 terms. Therefore, for this book, we have chosen a definition that follows the definitions in [10, 30, 43] and best fits our intuitive understanding:

Definition 2.2 (Data). Data are symbols, the properties, or “raw” facts of objects, events, and their environment. They are obtained from observations or measurements made by scientists and other people without (or very limited) contextual interpretation. Data are often quantitative and are presented in numbers, but can also consist of non-numerical symbols or statements.

Data itself, without context, contains no *information*—it is just a collection of facts. For example, a collection of numbers contains no meaning unless there is context or additional information about the purpose of the numbers or the particular ordering of the numbers. So what is required to make data useful? Data must be organized and structured, and it must have context. This brings us to one of the many possible definitions of *information*, which depends on the way how the term “data” is defined:

Definition 2.3 (Information). Information is what is created when data is organized, structured, processed, or analyzed, giving relevance to a particular purpose or context; it is of value to the person who is to use the information. Information is, for example, in the form of answers to questions.

Information can lead to decisions and trigger actions. Information itself, on another level, can again be seen as *data* that needs to be further organized and processed to

obtain information. This makes clear definitions difficult. Therefore, in Example 2.1, a number of examples are shown to elucidate these two concepts in the context of materials science and engineering.

Example 2.1 (Data and Information)



Here are some examples of situations relevant to materials science:

- A spreadsheet of numbers: Here, every number is a raw data value; all data are organized in rows and columns. Statistics summarizes and analyzes data, turning it into information, e.g., you can learn about the mean value of all data, which tells you something about the circumstances under which the data was collected.



- The “data stream” from a video camera: Here you should distinguish between the *raw data*, which could be the electrical signals, and the lightly processed data in the form of files that contain a lot of numbers. Looking at the latter file without knowing that it could be a video stream with a certain compression type makes it almost useless. Adding context, e.g., in the form of metadata (a comment line with a file description or a reference to the file/compression format used), turns it into information, e.g., when a histogram of brightness is computed.



- A text file with three columns of comma-separated numbers contains data—the numbers *and* the comma, which is an important detail. If the scientific context is that these data were recorded during a tensile test, this is already interesting information, but may not be enough to turn the data into information. We may need the additional hint that these are time-force-displacement data given in physical units (e.g., in micro-Newtons and nanometers)—another important distinction: data are just numbers, while information often contains units.



In information science, data and information are often viewed as parts of a hierarchy of at least four different terms or concepts, which are typically visualized in the form of the so-called *DIKW pyramid* (cf. [30] and references therein), shown in Fig. 2.4. There, data and information are accompanied by the two notions of *knowledge* and *wisdom*. The origin of this hierarchy is most probably T.S. Eliot's poem *The Rock* in 1934 [13], which contains the lines:

*Where is the wisdom that we have lost in knowledge?
Where is the knowledge that we have lost in information?*

Not surprisingly, there are also a number of different definitions for the two terms “knowledge” and “wisdom,” which might be even more difficult to harmonize. In general, knowledge is based on information and refers to the “*general understanding and awareness garnered from accumulated information, tempered by experience, enabling new contexts to be envisaged.*” to quote a definitions from [43]. Knowledge enables one to act and make decisions based on information. Finally, the transition to wisdom requires an understanding of the underlying principles, relationships, and mechanisms that enable one to answer “why” questions, and wisdom has a high level of abstraction.

These concepts and notions appear in many published works in the fields of information science/systems and knowledge management and have proven useful for basic classification and evaluation of objects, actions, and tasks in these fields and their meaning and value. We now turn to the question of whether and how this is also relevant to materials science and data science.

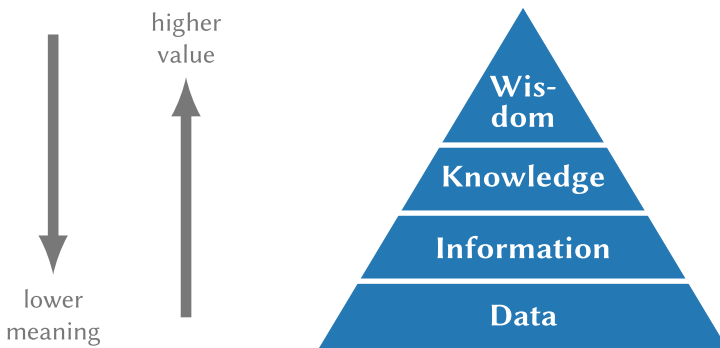


Fig. 2.4 The “DIKW pyramid,” showing the hierarchy of the four information science concepts *Data*, *Information*, *Knowledge*, and *Wisdom*. DIKW (or at least IKW) is believed to have originated from the play “*The Rock*” by T. S. Eliot in 1934. It symbolizes that each term is based on the terms underneath. The top of the pyramid and the final goal is “wisdom”

2.3.2 “Data and Information” from the Point of View of Data Science

From the previous section, it is clear that the information science definition of “data” and “information” differs from our colloquial use of the terms and from the way the terms are understood and used in materials science and engineering and even the field of data science. When we talk about *data*, we usually imply that it is not raw data, but that it is already organized and structured and that the context is clear—even if most people are not aware of the latter. Context is usually defined by describing the circumstances in which the data was created. This includes that we usually know what the data “means,” e.g., in terms of physical quantities and the units used.

2.4 The Curse of Dimensionality

Many problems in science and engineering depend on many different parameters, some of which are known and—worse—some of which may be completely inaccessible. Traditional computer simulations and theoretical models often use idealizations to reduce the problem of interest to a manageable substitute problem. The concept of a *digital twin*, which is a digital representation of an experimental sample or device, instead seeks to capture as many details and parameters as possible. This might look particularly promising in the context of ML approaches, which should be able to handle large datasets and parameter sets with ease and sometimes even be able to make predictions without knowing the underlying physical parameters.

While this is at least partially true, there is a serious problem directly related to the increase of parameters or the number of dimensions in general, known as the curse of dimensionality, an expression coined already in the 1950s by the mathematician R. E. BELLMAN [6]. We will explain this based on a simple case study given in Example 2.2.

Example 2.2 (Parameter Study)



Imagine that we can perform a total of 35 experiments in which we can vary any number of parameters, such as the magnitude of a force, a strain rate, temperature, and so on. Suppose that for each parameter we decide that eight different values are sufficient to capture a representative range of values. How many parameters can you exhaustively explore with the 35 experiments?

Of course, the variation of only one parameter, e.g., the temperature, can be performed. But what about two parameters? Then there would be already $8 \times 8 = 64$ possible combinations (also see Chap. 6) and 35 experiments are not enough. For three parameters we would need $8^3 = 512$ experiments and so on.

In this example, it is clear that the more parameters are investigated, the more data points are needed, and this number grows exponentially. A visualization can be found in Fig. 2.5, where now 35 points are randomly selected. Then their representation in a one-, two-, and three-dimensional (parameter) space is shown. We observe that with increasing dimensionality, the (parameter) spaces become more and more “empty.” In the two-dimensional case, we can still say that there is enough data to interpolate and thus fill the missing data. However, for the three-dimensional case, this might already seem too bold. This is a very simple example of the curse of dimensionality.

The consequences can be severe: on the one hand, we try to include more and more parameter and variables, trying to represent as many details as possible; on the other hand, this reduces the “degree of data coverage,” i.e., discretizing each dimension into p individual intervals and keeping the number data points k constant; in the best case, we would have a coverage of $k/(p^n)$ where n is the dimensionality of the considered space—the “degree of coverage” decreases exponentially with discretization having the greatest impact.

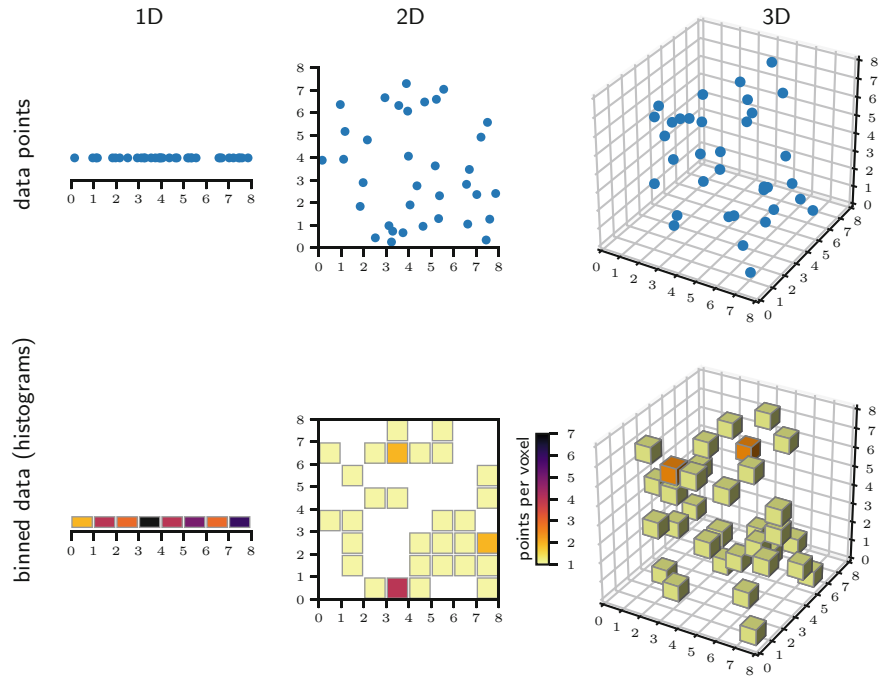


Fig. 2.5 A dataset consisting of 35 records (the blue markers), shown in a one-, two-, and three-dimensional space. The panels at the bottom are “density plots” and the colors correspond to the number of data points in the 1D interval, the 2D pixel, or the 3D voxel similar to a histogram. Only the non-empty pixel/voxel are shown

How can we deal with this problem? In the field of data science and machine learning, there are a number of methods and techniques that help to reformulate the problem. These include methods for dimensionality reduction, which we will cover in Chap. 15. There, the problem at hand is transformed into a lower-dimensional representation that is then (hopefully) easier to be solved. Another approach to mitigating the curse of dimensionality is the *design of experiment* (DoE), which helps to plan in advance how to sample a potentially large parameter space.

2.5 Summary and Conclusion

This chapter has given a very broad overview of a range of different aspects pertaining to data science and machine learning with a particular view on materials science. We also tried to give a first idea about data and in particular how “raw data” can be turned into something even more useful, i.e., information and knowledge. Therefore, the general scene is set now, and in the next chapter, we will take a closer look at “data” from a very practical perspective.

References

1. Material comparison strength vs density (licensed under cc by 4.0, <https://creativecommons.org/licenses/by/4.0/deed.en>. URL https://commons.wikimedia.org/wiki/File:Material-comparison--strength-vs-density_plain.svg.
2. M. F. Ashby. Overview no. 80: On the engineering properties of materials. *Acta metallurgica*, 37(5):1273–1293, 1989.
3. M. F. Ashby. *Materials Selection in Mechanical Design*. Elsevier, 4 edition, 2011. ISBN 978-1-85617-663-7. DOI <https://doi.org/10.1016/c2009-0-25539-5>.
4. L. Banko, P. M. Maffettone, D. Naujoks, D. Olds, and A. Ludwig. Deep learning for visualization and novelty detection in large x-ray diffraction datasets. *npj Computational Materials*, 7(1), July 2021. DOI <https://doi.org/10.1038/s41524-021-00575-9>.
5. J. Behler. Perspective: Machine learning potentials for atomistic simulations. *The Journal of chemical physics*, 145(17):170901, 2016.
6. R. Bellman. *Dynamic Programming*. Dover Publications, 1957. ISBN 9780486428093.
7. G. Bradski. The OpenCV Library. *Dr. Dobbs's Journal of Software Tools*, 2000.
8. L. Cao. Data science. *ACM Computing Surveys*, 50(3):1–42, June 2017. DOI <https://doi.org/10.1145/3076253>.
9. S. Curtarolo, G. L. Hart, M. B. Nardelli, N. Mingo, S. Sanvito, and O. Levy. The high-throughput highway to computational materials design. *Nature materials*, 12(3):191–201, 2013.
10. A. Debons, E. Horne, and S. Cronenweth. *Information Science: An Integrated View*. Professional librarian series. G.K. Hall, 1988.
11. B. L. DeCost, M. D. Hecht, T. Francis, B. A. Webler, Y. N. Picard, and E. A. Holm. Uhcsdb: ultrahigh carbon steel micrograph database: tools for exploring large heterogeneous microstructure datasets. *Integrating Materials and Manufacturing Innovation*, 6:197–205, 2017. DOI <https://doi.org/10.1007/s40192-017-0097-0>.
12. J. M. Ede. Deep learning in electron microscopy. *Machine Learning: Science and Technology*, 2(1):011004, 2021.
13. T. Eliot. *The Rock*. Faber & Faber, London, 1934.

14. K. Frydrych, K. Karimi, M. Pecelerowicz, R. Alvarez, F. J. Dominguez-Gutiérrez, F. Rovaris, and S. Papanikolaou. Materials informatics for mechanical deformation: A review of applications and challenges. *Materials*, 14(19):5764, Oct. 2021. DOI <https://doi.org/10.3390/ma14195764>.
15. M. Ge, F. Su, Z. Zhao, and D. Su. Deep learning analysis on microscopic imaging in materials science. *Materials Today Nano*, 11:100087, Aug. 2020. DOI <https://doi.org/10.1016/j.mtnano.2020.100087>.
16. J. Ghaboussi, J. H. Garrett, and X. Wu. Knowledge-based modeling of material behavior with neural networks. *Journal of Engineering Mechanics-asce*, 117:132–153, 1992.
17. S. R. Kalidindi and M. D. Graef. Materials data science: Current status and future outlook. *Annual Review of Materials Research*, 45(1):171–193, July 2015. DOI <https://doi.org/10.1146/annurev-matsci-070214-020844>.
18. C. Kusche, T. Reclik, M. Freund, T. Al-Samman, U. Kerzel, and S. Korte-Kerzel. Large-area, high-resolution characterisation and classification of damage mechanisms in dual-phase steel using deep learning. *PLOS ONE*, 14(5):e0216493, May 2019. DOI <https://doi.org/10.1371/journal.pone.0216493>.
19. M. I. Latypov, M. Kühbach, I. J. Beyerlein, J.-C. Stinville, L. S. Toth, T. M. Pollock, and S. R. Kalidindi. Application of chord length distributions and principal component analysis for quantification and representation of diverse polycrystalline microstructures. *Materials Characterization*, 145:671–685, 2018. DOI <https://doi.org/10.1016/j.matchar.2018.09.020>.
20. S. Lee, M. J. Duarte, M. Feuerbacher, R. Soler, C. Kirchlechner, C. H. Liebscher, S. H. Oh, and G. Dehm. Dislocation plasticity in fecocrmn high-entropy alloy: quantitative insights from in situ transmission electron microscopy deformation. *Materials Research Letters*, 8(6): 216–224, 2020a.
21. S. Lee, A. Vaid, J. Im, B. Kim, A. Prakash, J. Guénolé, D. Kiener, E. Bitzek, and S. H. Oh. In-situ observation of the initiation of plasticity by nucleation of prismatic dislocation loops. *Nature communications*, 11(1):1–11, 2020b.
22. L. McInnes, J. Healy, and J. Melville. UMAP: Uniform Manifold Approximation and Projection for dimension reduction. *The Journal of Open Source Software*, 3(29):861, 2020. DOI <https://doi.org/10.21105/joss.00861>.
23. B. D. Nguyen, M. Roder, A. Danilewsky, J. Steiner, P. Wellmann, and S. Sandfeld. Automated analysis of X-ray topography of 4H-SiC wafers: Image analysis, numerical computations, and artificial intelligence approaches for locating and characterizing screw dislocations. *Journal of Materials Research* 38:1254–1265, 2023. DOI <https://doi.org/10.1557/s43578-022-00880-z>.
24. B. D. Nguyen, J. Steiner, P. Wellmann, and S. Sandfeld. Combining unsupervised and supervised learning in microscopy enables defect analysis of a full 4H-SiC wafer, 2024. DOI <https://doi.org/10.48550/arXiv.2402.13353>.
25. OpenAI. Gpt-4 technical report, 2023.
26. A. Prakash and S. Sandfeld. Chances and challenges in fusing data science with materials science. *Practical Metallography*, 55(8):493–514, 2018. DOI <https://doi.org/10.3139/147.110539>.
27. M. H. Rafiei, Y. Gu, and J. A. El-Awady. Machine learning of dislocation-induced stress fields and interaction forces. *JOM*, 72(12):4380–4392, Oct. 2020. DOI <https://doi.org/10.1007/s11837-020-04389-w>.
28. K. Rajan. Materials informatics. *Materials Today*, 8(10):38–45, 2005. ISSN 1369-7021. DOI [https://doi.org/10.1016/S1369-7021\(05\)71123-8](https://doi.org/10.1016/S1369-7021(05)71123-8). URL <https://www.sciencedirect.com/science/article/pii/S1369702105711238>.
29. R. Ramprasad, R. Batra, G. Pilania, A. Mannodi-Kanakithodi, and C. Kim. Machine learning in materials informatics: recent applications and prospects. *npj Computational Materials*, 3(1), Dec. 2017. DOI <https://doi.org/10.1038/s41524-017-0056-5>.
30. J. Rowley. The wisdom hierarchy: representations of the dikw hierarchy. *Journal of Information Science*, 33(2):163–180, 2007. DOI <https://doi.org/10.1177/0165551506070706>.
31. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

32. M. Sarvilahti, A. Skaugen, and L. Laurson. Machine learning depinning of dislocation pileups. *APL Materials*, 8(10), 10 2020. ISSN 2166-532X. DOI <https://doi.org/10.1063/5.0020376>. 101109.
33. D. Steinberger, H. Song, and S. Sandfeld. Machine learning-based classification of dislocation microstructures. *Frontiers in Materials*, 6, 2019. ISSN 2296-8016. DOI <https://doi.org/10.3389/fmats.2019.00141>.
34. D. Steinberger, I. Issa, R. Strobl, P. J. Imrich, D. Kiener, and S. Sandfeld. Data-mining of in-situ TEM experiments: Towards understanding nanoscale fracture. *Computational Materials Science*, 216:111830, Jan. 2023. DOI <https://doi.org/10.1016/j.commatsci.2022.111830>.
35. R. Strack. Deep learning advances super-resolution imaging. *Nature Methods*, 15(6):403–403, May 2018. DOI <https://doi.org/10.1038/s41592-018-0028-9>.
36. I. Tanaka, K. Rajan, and C. Wolverton. Data-centric science for materials innovation. *MRS Bulletin*, 43(9):659–663, 2018.
37. P. Trampert, D. Rubinstein, F. Boughorbel, C. Schlinkmann, M. Luschkova, P. Slusallek, T. Dahmen, and S. Sandfeld. Deep neural networks for analysis of microscopy images—synthetic data generation and adaptive sampling. *Crystals*, 11(3), 2021. ISSN 2073-4352. DOI <https://doi.org/10.3390/cryst11030258>. URL <https://www.mdpi.com/2073-4352/11/3/258>.
38. L. von Chamier, R. F. Laine, J. Jukkala, C. Spahn, D. Krentzel, E. Nehme, M. Lerche, S. Hernández-Pérez, P. K. Mattila, E. Karinou, S. Holden, A. C. Solak, A. Krull, T.-O. Buchholz, M. L. Jones, L. A. Royer, C. Leterrier, Y. Shechtman, F. Jug, M. Heilemann, G. Jacquemet, and R. Henriques. Democratising deep learning for microscopy with Zero-CostDL4mic. *Nature Communications*, 12(1), Apr. 2021. DOI <https://doi.org/10.1038/s41467-021-22518-0>.
39. L. Ward and C. Wolverton. Atomistic calculations and materials informatics: A review. *Current Opinion in Solid State and Materials Science*, 21(3):167–176, 2017.
40. M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. ’t Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3(1), Mar. 2016. DOI <https://doi.org/10.1038/sdata.2016.18>.
41. C. Zhang, H. Song, D. Oliveros, A. Fraczekiewicz, M. Legros, and S. Sandfeld. Data-mining of in-situ tem experiments: On the dynamics of dislocations in CoCrFeMnNi alloys. *Acta Materialia*, 241:118394, 2022. DOI <https://doi.org/10.1016/j.actamat.2022.118394>.
42. C. Zhang, H. Song, D. Oliveros, A. Fraczekiewicz, M. Legros, and S. Sandfeld. Data-mining of in-situ tem experiments: On the dynamics of dislocations in cocrfemnni alloys. *Acta Materialia*, 241:118394, 2022.
43. C. Zins. Conceptual approaches for defining data, information, and knowledge. *Journal of the American Society for Information Science and Technology*, 58(4):479–493, 2007. DOI <https://doi.org/10.1002/asi.20508>.

What You Should Know About Data, Math, and Computing 3

It is a capital mistake to theorize before one has data.

Sir Arthur Conan Doyle (1859–1930),
creator of the character Sherlock Holmes

3.1 General Conventions and Notations

Let us start with a brief overview of the most important mathematical conventions and notations used in this text. All used abbreviations and acronyms are listed on page xxv. As we do not expect the reader to move linearly through this text from the beginning to the end, we introduce the used acronyms for each chapter anew.

Concerning mathematical notations, we will make extensive use of scalars, vectors, matrices, and other objects. We mainly stick to conventions that are commonly used in engineering and physics. But since there is not *the* one convention, we state the most important ones here:

Definition 3.1 (Basic Definitions and Notations)

- **Scalars**, i.e., regular numbers, are given by italic Greek or Latin letters, e.g., $a = 1.423$ or $\lambda = 2$.
- **Vectors** are usually written as lowercase, bold, and italic Latin characters, e.g., $\mathbf{a}$ and $\mathbf{x}$, but sometimes we also encounter Greek letter, e.g., as in $\sigma = [0.5, 0.4, 0.9]$.
- **Matrices** are indicated by uppercase, sans-serif, bold Latin characters, e.g., $\mathbf{A}$ or $\mathbf{X}$.

(continued)

- The sets of **real and integer numbers** are denoted by $\mathbb{R}$ and $\mathbb{Z}$, respectively. The latter consists of all positive as well as negative integers including zero. The natural numbers $\mathbb{N}$ are the set of positive integers.
- **Functions** depend on variables, e.g., in $f(x)$ the function is f and x is the variable. To specify fixed parameters or constants p_1, p_2 , we use a semicolon and write, e.g.,

$$f(x; p_1, p_2) = p_1 + p_2 x^2 \quad \text{with} \quad p_1 = 0.5 \text{ and } p_2 = 2.0 .$$

This reads “function f depends on x for given parameters p_1 and p_2 .” An alternative notation is $f_{p_1, p_2}(x)$.

- Different types of **parentheses and brackets** need to be differentiated: $()$ are called *round brackets* or *parentheses*; $[]$ are *square brackets* or just *brackets*, as opposed to *curly brackets* or *braces* $\{\}$; the last type are *angular brackets*, $\langle \rangle$, which are also sometimes called *chevrons*.
- **Python code** is written with a light-gray background, e.g.,

```
print('x =', x) .
```

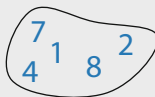
3.2 Sets, Tuples, Vectors, and Arrays

This section introduces definitions of sets, tuples, lists, and arrays. Our definitions are not formal mathematical definitions but are rather definitions of how the terms are commonly used and what the notations are. As many of these terms are used across different disciplines, it is important to settle on “common grounds.” We also show short Python examples as we believe that it is useful for the reader to see the mathematical notation right next to the practical implementation. For those reader who want to acquire some foundation in Python, e.g., [1, 2] are two of many good starting point for that.

3.2.1 Sets

When distinct items are to be collected in a group, the members of this group are called *elements*. To describe the group, the mathematical *set* can be used, which is defined as follows:

Definition 3.2 (Set) A mathematical *set* is an unstructured collection of different elements without repetitions. Elements can be, e.g., numbers, variables, symbols, or also other sets. We indicate a set by curly braces, and elements are separated by commas. For example, the set of the numbers 7, 1, 4, 2, and 8 is written as $\{7, 1, 4, 2, 8\}$.



A set can also be defined by describing the properties of the elements of the set S , e.g.,

$$S = \{n \mid n \in \mathbb{N}, n \text{ is an odd number, and } 1 \leq n \leq 10\} \quad (3.1)$$

which consists of two parts: to the left of $|$ ¹ stands a variable name, which is then used to the right of $|$ to describe properties. Equation (3.1) is pronounced “*The set of all n where n is an odd integer number between 1 and 10*”—in other words, $S = \{1, 3, 5, 7, 9\}$.

The fact that the set is an unstructured collection implies that the ordering of the elements does not matter. The concept of a set also exists in Python. For example, two sets of numbers can be compared by

```
In [1]: set([3, 4, 5]) == set([4, 3, 5, 3])
```

```
Out [1]: True
```

which here shows that both sets are identical despite the double occurrence of the number 3, as a set does not contain duplicate elements (note the phrase “without repetitions” in Definition 3.2). Additionally, the ordering of the elements in a set does not matter (or rather: there *is* no order in a set).

Set Operations

A number of basic mathematical operations on sets exist, which can be visualized using *Venn-diagrams* (closed geometrical shapes that may overlap and contain certain items, used to visualize set operations). Assuming that A and B are sets, then the most important operations between them are:

¹ Sometimes, instead of the vertical line “ $|$,” a colon “ $:$ ” is used in definitions of sets.

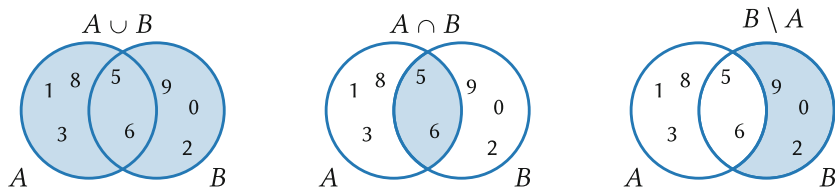


Fig. 3.1 Venn diagrams for three operations between the sets $A = \{5, 3, 1, 8, 6\}$ and $B = \{9, 5, 0, 6, 2\}$ are shown as the two circles in each plot. The shaded area of the left panel shows the union of A and B , the shaded area of the middle panel is their intersection, and the right panel shows the difference $B \setminus A$

- **Union:** The union $A \cup B$ is the set containing all elements of A and B .
- **Intersection:** The intersection $A \cap B$ are those elements that occur both in the set A and in B .
- **Difference:** The difference $A \setminus B$ (or alternatively written as $A - B$) is the set of all elements that are in A but not in B .

A special set is the *empty set* $\{\}$, which contains no elements. The empty set is commonly denoted by $\emptyset$. If for two sets A and B their union is $A \cup B = \emptyset$, then A and B are called *disjoint sets*. A visualization of sets, represented as a closed curve, and some mathematical operations are shown as Venn diagrams in Fig. 3.1. From these visualizations, it can be seen that the union of A and B is $A \cup B = \{1, 8, 3, 5, 6, 9, 0, 2\}$, their intersection is $A \cap B = \{5, 6\}$, and the difference “ B without A ” is $B \setminus A = \{9, 0, 2\}$.

Python already has the most common set operations built in (and as always, there is more than just one way of achieving the desired goal; see the comments in the following code lines):

Create two sets:

```
In [1]: A = {'a', 'bb', 'c'}
        B = {'A', 'bb', 'd', 42}
```

Union of the two sets

```
In [2]: set.union(A, B) # alternative code: A | B
```

```
Out [2]: {42, 'A', 'a', 'bb', 'c', 'd'}
```

Intersection of A and B :

```
In [3]: set.intersection(A, B) # alternative code: A & B
```

```
Out [3]: {'bb'}
```

Difference $A \setminus B$

```
In [4]: set.difference(A, B) # alternative code: A - B
```

```
Out [4]: {'a', 'c'}
```

Difference $B \setminus A$

```
In [5]: set.difference(B, A) # alternative code: B - A
```

```
Out [5]: {42, 'A', 'd'}
```

3.2.2 Tuple

In many situations, however, the ordering of elements is important. In these cases an “ordered list” is useful. In math this is called a tuple:

Definition 3.3 (Tuple) A *tuple* is a collection of elements such as numbers, variables, or symbols that occur in an ordered manner, where repetition matters. This is indicated by parenthesis, and elements are separated by commas. For example, the tuple of the letters m, d, and s is written as (m, d, s) . Sometimes also square brackets are used, e.g., $[4, 5, 5, 1]$.

4	5	5	1

In Python there exist *two* slightly different data types, both of which correspond to the mathematical definition of a tuple from Definition 3.3: the Python `tuple` and the Python `list`. The difference between the two Python data types is that the values of the elements of a `list` can be altered or elements can be deleted while a `tuple` is “immutable,” i.e., the content cannot be changed.

As opposed to the set, Python’s `tuple` and `list` do consider repeated elements, as shown in this example:

```
In [1]: list([1, 3, 5]) == list([1, 3, 3, 5, 5])
```

```
Out [1]: False
```

This line of code compares two lists, returning `False` as they are not the same. The lists have a different number of elements; if they were contained in a `set`, the same comparison would yield `True`. Note that in Python a list is also created by `[1, 3, 4]` and a tuple is created by either `tuple([1, 3, 5])` or just `(1, 3, 5)`.

A number of operations exist for a Python `list`, such as appending or removing elements or counting or sorting elements of a list. The Python `tuple` is quite similar to a Python `list` concerning “read-only”—operations that do not change the content. We will often encounter lists (and less often also tuples) in implementations shown in this text.²

3.2.3 Intervals

Beware of how intervals, i.e., ranges of numbers, are written—some intervals also use square brackets or parentheses and can be confused with tuples. To make the difference clear, we also give the definition of an interval:

Definition 3.4 (Interval) An interval is denoted by $[a, b]$ and contains *all* numbers between the start and end points a and b , including both the start point and the end point. To indicate that the start or end point are not included in the interval, then a parenthesis is used for this point instead of the square bracket.



For example, in the context of real numbers, (a, b) denotes the open interval of all numbers x with $a < x < b$ and $[a, b)$ denotes the half-open interval of all numbers x with $a \leq x < b$. The notation also applies for intervals of integer numbers. From the context it should always be clear if $[a, b]$ denotes a closed interval or a list.

Integer interval A similar notation using square brackets or parentheses also holds for integer numbers while in that case the start and end values are *always* included. Therefore, we prefer a more explicit notation using an ellipsis, e.g., instead of $(i, i + 3)$, we write $(i, \dots, i + 3)$, which denotes the tuple of numbers $(i, i + 1, i + 2, i + 3)$. Alternatively, square brackets can be used. However, as opposed to intervals of real numbers, a mixture of square *and* round brackets would *not* denote a half-open interval and is therefore virtually never used.

In Python an integer interval is also called a *range* and is used, e.g., to specify or *index* elements in a vector or matrix, as introduced below.

² A good place for someone with intermediate Python skills for learning more about many features of Python is the documentation. It can be found online at <https://docs.python.org/>.

Words of advice

- In mathematics, an integer interval includes both the start and end point, while for intervals of real numbers, this is not necessarily the case!
- In mathematics, an interval of real numbers follows a different notation, e.g., in $(2, 4)$, the end points are both excluded. Here, the context is important (i.e., knowing that these should be real values).
- Examples:

It is $(i, i + 3) = (i, \dots, i + 3) = (i, i + 1, i + 2, i + 3)$

and $[-2, 2] = [-2, \dots, 2] = [-2, -1, 0, 1, 2]$

but $[1.0, 5.5) = \{x | 1 \leq x < 5.5\}$.

- Python follows—for various reasons—a different convention: in the Python command `range(i, i+3)`, the end point is excluded by default, and `range(4)` includes the numbers 0, 1, 2, and 3 as Python is “zero-based.”

3.2.4 Matrices and Row and Column Vectors

Is there a relation between tuples or lists and *vectors*? The answer is that they are not related, at least not mathematically spoken, because vectors are the elements of a vector space, which is a completely different story.

However, there is a related meaning in the context of matrices. Matrices play a central role in linear algebra and can be used to represent systems of linear equations (and we will extensively make use of that in the context of optimization in later chapters). Matrices are also used in the spirit of “containers” for (typically) two-dimensional data such as tabular data. But prior to that, we now give a definition of what a *matrix* is.

Definition 3.5 (Matrix) A matrix **A** is an ordering of real-valued elements into m rows and n columns

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad (3.2)$$

(continued)

where an element in an arbitrary row number i (counted from the top) and column number j (counted from the left) is denoted by a_{ij} . The a_{ij} are typically real numbers but can also be integer numbers. The matrix in Eq. (3.2) is called a $m \times n$ matrix.

A number of mathematical operations exist, such as matrix multiplication, addition of two matrices, multiplication of a matrix with a scalar, to name a few. An introduction of the most important rules and details can be found in Appendix A. Also in Python, a number of methods for creating and processing of matrices exist that are compatible with the above-introduced notations. E.g. a 3×4 matrix consisting of zeros can be created using the `numpy` package by

```
In [1]: import numpy
        A = numpy.zeros((3, 4))
        A

Out [1]: array([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.]])
```

The element in the second row and third column can then be accessed by `A[1, 2]` (note that Python indices start at 0, while indices in maths and MATLAB® start at 1). The indexing can be used to set a value of an element (e.g., `A[1, 2] = 1.2`) as well as to retrieve its value (e.g., `print(A[1, 2])`):

```
In [2]: import numpy
        A[1, 2] = 1.2
        A

Out [2]: array([[0., 0., 0., 0.],
               [0., 0., 1.2, 0.],
               [0., 0., 0., 0.]])
```

Row and column vectors appear in numerous situations in *machine learning* (ML). Additionally, the scalar product of two vectors as it is used throughout this text requires the differentiation between row and column vectors.³ We define row and column vectors as certain “sub-matrices” of a matrix as follows:

³ Having to differentiate between row and column vectors is a side effect of the chosen notation. While other, more elegant and general notations exist, it turns out that this is the most accessible approach at least for those who do not have a strong mathematics background in tensor calculus. Additionally, the used notation is compatible with Python implementations.

Definition 3.6 (Row and Column Vectors) Given a matrix $\mathbf{A}$ as in Definition 3.5 a *row vector* is defined as a $1 \times n$ sub-matrix of $\mathbf{A}$. In analogy, a *column vector* is defined as a $m \times 1$ sub-matrix of $\mathbf{A}$. Furthermore:

- a row vector is written in the same way as a list, for example:

$$\mathbf{v} = [a_{21}, a_{22}, \dots, a_{2n}]$$

- a column vector is given by the following expression:

$$\mathbf{w} = \begin{bmatrix} a_{11} \\ a_{21} \\ \dots \\ a_{m1} \end{bmatrix} = [a_{11}, a_{21}, \dots, a_{m1}]^T$$

where the $[\cdot]^T$ in the last expression is the transposition symbol, which exchanges rows with columns. The commas in both expressions are written in analogy to the above definition of a list but are often dropped for brevity.

In other words, a row or column vector consists simply of all elements of a particular row or column of a matrix. Appendix A contains more general information about notation in linear algebra. Python has a straightforward way of denoting a full row or column using either the Ellipsis object (`...`) or the colon (`:`), e.g., to set all elements of row 2 to the value 3, one can write

```
In [1]: A[2, :] = 3 # alternatively: A[2, ...] = 3 or A[2] = 3
        A
Out [1]: array([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [3., 3., 3., 3.]])
```

Similarly, `A[:, 1]` or `A[..., 1]` addresses the elements of the second column. However, care needs to be taken concerning the dimensions of the resulting object, which is a one-dimensional array.

Words of advice

Extracting a row or column from a matrix $\mathbf{A}$ with m rows and n columns by, e.g., $\mathbf{A}[:, 1]$ results in a one-dimensional array, and it *does not* result in a column vector with $m \times 1$ elements (which might be counter-intuitive, at a first glance). This can be seen by looking at the “shape” of the result, i.e., the result’s dimensions, e.g., $\mathbf{A}[:, 1].\text{shape}$ gives m , and $\mathbf{A}[2, :].\text{shape}$ gives n , which indicates one-dimensional arrays with m elements for the first case and n elements for the second case.

Use the function `numpy.reshape` to turn a one-dimensional array into a row or column vector, e.g., the one-dimensional array `c = numpy.array([1, 2, 3])` is turned into a two-dimensional matrix with three rows and one column by `numpy.reshape(c, (3, 1))`.

In the explicit formulation of most *machine learning* (ML) problems, a number of other super- and subscripts are present. As this can make it difficult to directly understand the notations, clear and practical definitions are particularly useful. Therefore, in close analogy to Python or MATLAB[®] syntax, we introduce the following short form for row and column vectors:

Definition 3.7 (Short Notation for the i -th Row or Column Vectors)

Given a matrix $\mathbf{A}$ with the component a_{pq} as in Definition 3.5, we introduce the following short notation for indexing using a colon (:):

- for the i -th row vector: $a_{i,:} = [a_{i1}, a_{i2}, \dots, a_{in}]$,
- for the i -th column vector: $a_{:,i} = [a_{1i}, a_{2i}, \dots, a_{mi}]^T$

In this *mathematical* definition, the indices start at 1. Note, however, that Python indices always start at 0. Sometimes, when mathematical expressions get more complicated, we will surround the above indices by brackets for better readability (e.g., $a_{[:,i]}$), but this will be explicitly mentioned.

With the above definitions and conventions, we are now ready to take the next step and to investigate how data used in the context of statistics and ML is represented.

3.3 Representation of Data in Statistics and Machine Learning

The data that we are interested in for statistical analysis, data mining, or ML can come in various forms: it could be time series data of a scalar quantity (such as measurements of temperature over time), it can be vector-valued field data measured at discrete points such as a strain field measured at points of an equidistant grid, or it could be unstructured data, e.g., represented by a network-like structure.

We begin by introducing some general aspects of the mathematical shape of typical problems with particular view on the data and data structures. Based on this, a number of definitions are given that will form the basis of nearly all of the following chapters.

3.3.1 General Problem Formulation

Regardless of the particular data type at hand, many of the physics and data science problems considered in later chapters can be reduced to a simple albeit abstract mathematical form:

$$\mathbf{y} = f(\mathbf{x}) \quad (3.3)$$

$$\Leftrightarrow (y_1, \dots, y_m) = f(x_1, \dots, x_n), \quad (3.4)$$

i.e., some input that is *somehow* related by a function to some output. An example for a function with three input variables and one output is $y = f(x_1, x_2, x_3)$, and an example for a function with a single input and two output variables would be $(y_1, y_2) = f(x)$.

Words of advice



Note that in Eq. (3.4), we use the typical notation from mathematical analysis where it is not distinguished between row and column vectors. Furthermore, Eq. (3.3) has, strictly speaking, a single function argument (the vector $\mathbf{x}$), but Eq. (3.4) has n of them. Most of the time, the meaning is clear from the context. However, when it comes to implementing such equations in a computer program, we have to be careful and will be as explicit as possible.

In Eq. (3.3), the variable $\mathbf{x}$ is an *input variable* possibly consisting of more than one components x_i . The *output* variable $\mathbf{y}$ also may consist of a number of components y_j . Both, $\mathbf{x}$ and $\mathbf{y}$, may have different number of elements, depending on the respective problem under consideration.

The Names of Variables ...



In mathematics, the x_i are also called *independent variables*, which suggests that they cannot be obtained as the result of any function of other variables. However, in statistics and data analysis, there is no requirement of “independency of input variables” (in fact, often input variables *are* correlated), and hence, we typically avoid the mathematics notion of “independent variables.” In statistics or ML, a number of different names other than “input variables” are given to these two variables; an overview is given in Sect. 11.6.5.

The function f can, in principle, take any arbitrary mathematical form, it can be a linear or a nonlinear function, or it also might “hide” an ordinary differential equation in which case the input consists of the initial value, while the output consists of the resulting value after time integration. Furthermore, f need not be a mathematical function; it even might be a *black box model* where, e.g., a *lookup table* returns for each input the corresponding output. An example explaining the relation of the input and output data is given in Example 3.1.

Example 3.1 (Problem formulation with input and output variables) ✂

Why is a general mathematical function $\mathbf{y} = f(\mathbf{x})$ representative for many problems that we encounter in materials science, engineering, and physics?

We often ask questions such as “*What is the temperature at this particular point?*” or “*What is the property of the material after a particular heat treatment of the specimen?*” In both cases there are input variables and output variables. For example, in the first case, the input variable is a coordinate in space, (x_1, x_2, x_3) , while the output variable is a temperature T . In the second example, the input could be the characteristics of the materials microstructure after a certain heat treatment, while the output variable is the resulting property.

If we know how to compute $f(\mathbf{x})$ for any of the two cases, then we are able to *predict* the temperature or the resulting material property!

Equation (3.3) serves as a template for almost all problems that we consider throughout this text. To be able to understand and systematically explore the relation between input data and output data, given by the function f , we will introduce common terminologies and the relevant data structures in the following.

3.3.2 Structured Data

Throughout this text we will be able to represent most of the data in a structured format as tabular or “array” data. Nonetheless, notations can become complicated when we need to differentiate between different spatial points, vector-valued components, and data with different purposes (often indicated by various sub- or superscripts). Therefore, we start from the very basics by giving definitions and introduce notions that will be used to describe all kinds of data and related aspects: the *data point*, the *data record*, and the *dataset*.

Definition 3.8 (Data point $\mathcal{P}$) A *data point* is a single observation consisting of the input variables $\mathbf{x} = (x_1, \dots, x_n)$ and the output variables $\mathbf{y} = (y_1, \dots, y_m)$ where the data point $\mathcal{P}$ is given as a tuple of row vectors by

$$\mathcal{P} = (\mathbf{x}, \mathbf{y}) = ([x_1, \dots, x_n], [y_1, \dots, y_m]). \quad (3.5)$$

If there is only one independent or dependent variable ($n = 1$ or/and $m = 1$), then $\mathbf{x}$ and/or $\mathbf{y}$ reduce to the scalar variables x and/or y .

In the context of experimentally acquired data, a data point is also called *measurement*; in statistics and ML also the names *instances* or *examples* are used. Here is a simple Python example for defining a data point `P1`:

```
In [1]: import numpy as np
x = np.array([[2, 4, 6, 11]]) # directly write as 2D array
y = np.array([3, -1, 4, 0]).reshape(1, 4) # or: 1 row, 4 columns
P1 = (x, y)
P1
```

```
Out [1]: (array([[ 2,  4,  6, 11]]), array([[ 3, -1,  4,  0]]))
```

We use the script font letter $\mathcal{P}$ to indicate that a data point is not just a vector of two elements $\mathbf{x}$ and $\mathbf{y}$ but in fact a “compound” of $\mathbf{x}$ and $\mathbf{y}$ (a tuple $(\mathbf{x}, \mathbf{y})$). Usually, we can assume that each data point is a measurement of the same quantities such that each data point has the same number of elements resulting in a *structured dataset*. In other words, each int (or instance) number i is given by $\mathcal{P}^{(i)}$ and has the structure

$$\mathcal{P}^{(i)} = \left([x_1^{(i)}, \dots, x_n^{(i)}], [y_1^{(i)}, \dots, y_m^{(i)}] \right). \quad (3.6)$$

Note that in this equation, the data point consists of n input values (e.g., $n = 3$ spatial coordinates) and m output values (e.g., the $m = 6$ independent values of a strain tensor), and in general it is not necessary that $n = m$; see Example 3.3.

We can now define the *dataset*, a notion that we so far have used without properly having defined it:

Definition 3.9 (Dataset $\mathcal{D}$) A dataset $\mathcal{D}$ is the collection of all n data points $\mathcal{P}$ that belong to a statistical, computational, or real experiment. Assuming that the ordering matters, the dataset is the tuple

$$\mathcal{D} = \left(\mathcal{P}^{(1)}, \dots, \mathcal{P}^{(n)} \right). \quad (3.7)$$

In continuation of the above Python code snippet, here is an example that details the definition of the dataset:

```
In [2]: P2 = ... # need to be defined in analogy to P1
        D = (P1, P2)
        D
```

```
Out [2]: ((array([[2, 4, 6, 1]]), array([[ 3, -1,  4,  0]])),
          (array([[3, 7, 0, 2]]), array([[ 5, -2,  4,  1]])))
```

Assuming that all data points have the same structure as in Eq. (3.6), a dataset can easily be stored in a table or represented as a matrix of data. This is also motivated by the previous Python example where we can already see that there are two “blocks” of data, and each new data point adds a new row.

What if the data is not structured?



There may exist situations where the number of input features or outputs is changing, e.g., if in a *molecular dynamics* (MD) simulation with a grand canonical ensemble the number of atoms changes and atomic positions are recorded, if the experimental measurement points are rearranged, or if in a *finite element analysis* (FEA) the mesh is refined and the number of element and thereby the number of, e.g., displacement values changes. In these cases an averaging or “resampling strategy” (e. g., interpolation at regularly spaced points) can be used to represent the data in a data matrix. In all other cases, it even might become necessary to change, e.g., details of the planned experiment.

Example 3.2 (Data Points and Dataset)



Assume a metallic rod that has a length $l = 1$ m and on which in equidistant points $n = 5$ temperature sensors are attached. Their location is $x_i = l \cdot (i - 1)/(n - 1)$ or $\mathbf{x} = [0, 0.25, 0.5, 0.75, 1 \text{ m}]$, which is the input variable. Each temperature sensor has as output a single value, the temperature T_i . Hence, the output variable is $\mathbf{y} = [T_1, \dots, T_5]$. The data point consists of the five input and output values. Taking altogether four measurements (e.g., after a certain time) then results in a dataset that consists of 4 points $\mathcal{P}^{(1)}, \dots, \mathcal{P}^{(4)}$, each of which having ten items.

3.3.3 Tabular Data and the Data Matrix

After having introduced what a dataset is from a rather mathematical point of view, we now turn to the more technical aspect of how to represent or to store such structured data. The most common approach is to contain the data in form of a table or array where each row contains, e.g., a single measurement, the *data record*, which is sometimes also called *instance* or *example*. Writing the data records in rows on top of each other results in tabular data shown in Fig. 3.2. The shown dataset consists of m records in rows, e.g., obtained from measurements. It has n input variables and p output variables in columns, which result in the $m \times (n + p)$ *data matrix*. The input variables X_1, X_2 , etc. are also called the *features* of the dataset. Often, the number of outputs is much smaller than the number of input variables, and in many cases, there is even only one output. In some ML methods (the so-called unsupervised learning methods), the output data is even missing entirely. Example 3.3 illustrates these notations and conventions with an example of how to create the data table for an experiment.

	Input				Output			
	X_1	X_2	$\cdots$	X_n	Y_1	Y_2	$\cdots$	Y_p
Record No. 1 ($\mathcal{P}^{(1)}$):	x_{11}	x_{12}	$\cdots$	x_{1n}	y_{11}	y_{12}	$\cdots$	y_{1p}
Record No. 2 ($\mathcal{P}^{(2)}$):	x_{21}	x_{22}	$\cdots$	x_{2n}	y_{21}	y_{22}	$\cdots$	y_{2p}
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
Record No. m ($\mathcal{P}^{(m)}$):	x_{m1}	x_{m2}	$\cdots$	x_{mn}	y_{m1}	y_{m2}	$\cdots$	y_{mp}

Fig. 3.2 Tabular data consisting of n input variables (or features) and p output variables. Altogether m data records are contained in this dataset where record number i is given by the data point $\mathcal{P}^{(i)}$. The blue matrix of input data is denoted by $\mathbf{X}$, and the orange matrix of output data is denoted by $\mathbf{Y}$

Example 3.3 (Representation of a Measurement Dataset (Cooling of a Specimen))

An experiment investigates how the temperature during a cooling down process of a specimen changes as a function of time. Besides temperature, also the air pressure is recorded as there might be a correlation between pressure and temperature. We obtain a measurement every 10 s over a period of 1 min. For the first half of the measurement period, the pressure values are 1010 hPa, at $t \geq 30$ s and the values are 1013 hPa. The temperature drops linearly from 180 °C at a rate of 1 °C s⁻¹.

Questions: (i) What are the independent and dependent variables? (ii) Write down all components of the dataset in the same way as shown in Fig. 3.2. (iii) What is the data record number 2? (iv) What is the first feature? (v) What is the input record number 3?

Answers: (i) Time and pressure are the independent variables X_1 and X_2 , and temperature is the dependent variable Y_1 (or just Y). (ii) The table has seven rows including the starting point in time:

	Input/Features		Output
	X_1	X_2	Y
Record No.	time in s	pressure in hPa	temperature in °C
1:	0	1010	180
2:	10	1010	170
3:	20	1010	160
4:	30	1010	150
5:	40	1013	140
6:	50	1013	130
7:	60	1013	120

(iii) Record number 2 is the data point $\mathcal{P}^{(2)} = [10, 1010, 170]$; (iv) the first feature is time: $X_1 = [0, 10, \dots, 60]^T$; (v) the input record number 3 is the row vector $\mathbf{x}_3 = [20, 1010]$.

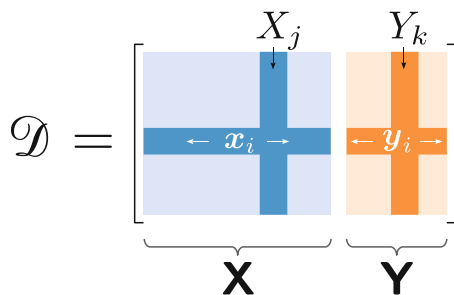


Fig. 3.3 Sketch of the data matrix and the different types of variables. Each of the boldface $\mathbf{x}_i$ denotes a possibly multidimensional data record (a row vector), the non-boldface X_j and Y_k denote several records or measurements of a scalar input or output variable (column vectors). $\mathbf{X}$ and $\mathbf{Y}$ are sub-matrices containing all input and output data, respectively

3.3.4 Matrix and Vector Short Forms for Linear Algebra

Based on the dataset shown in Fig. 3.2, we now introduce and summarize variables and notations that will be frequently used throughout this text. These short forms for special vectors and sub-matrices are in particular useful for concise linear algebra formulations such as vector-matrix products, which will turn out to be helpful for formulations as well as numerical implementations in statistics and ML. A sketch of these quantities can be found in Fig. 3.3.

First, we take a look at the individual feature X_i and output variable Y_j . Both of them are concerned with scalar quantities (e.g., X_1 could be the first Cartesian coordinate of an object and Y_1 its temperature). However, we almost never operate with just a single measurement or data point. Instead, we usually have several measurements because, e.g., we record the position of an object as a function of time as one of the input variables X_i . This is why any feature X_i is in fact represented by a column vector of values $[x_{1i}, x_{2i}, \dots]$ (cf. Example 3.3, question (iv)).

If we, on the other hand, take a look at the m -th data record, then this is given by the vector $\mathbf{x}_m = [x_{m1}, x_{m2}, \dots]$, e.g., this could be coordinates of an object or point or even coefficients of quaternions (which are used to describe texture in crystal plasticity). This is in fact a vector value that also explains the boldface $\mathbf{x}_m$ as opposed to the non-boldface X_j (cf. Example 3.3, question (v)).

Words of advice



Beware of the notations: $\mathbf{x}_i$ is a row vector (the i -th record), but X_j is a column vector (all records of the j -th scalar feature). Even though the boldface $\mathbf{x}_i$ /non-boldface X_j convention might seem unusual, this notation is consistent with the commonly used one in probability theory and statistics, and it represents the fact that any value of X_j is a scalar, while $\mathbf{x}_i$ can be the three-dimensional coordinate of a point, i.e., a vectorial quantity.

Given these column and row names, we can now summarize this in terms of vector notation. The data of a feature variable is specified by:

Definition 3.10 (All records of a specific input or output variable) The variable of the k -th feature or input variable is X_k . The data is given by the column vector

$$X_k = [x_{1k}, \dots, x_{mk}]^T = x_{:,k} . \quad (3.8)$$

In analogy, all data of the r -th output variable is given by the column vector

$$Y_r = [y_{1r}, \dots, y_{mr}]^T = y_{:,r} . \quad (3.9)$$

The colon in $x_{:,k}$ and $y_{:,k}$ denotes all rows, similar to numpy notation. In case that there is only one feature, we just write X or Y .

As before, please refer to Figs. 3.2 and 3.3 for a visualization of these conventions. The i -th data record or data point consists of input and output values:

Definition 3.11 (Specific data record or data point) The record number i consists of the input data $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{in}]$ and the output data $\mathbf{y}_i = [y_{i1}, \dots, y_{ip}]$. It is written in a compact form as

$$\mathcal{P}^{(i)} = (\mathbf{x}_i, \mathbf{y}_i) . \quad (3.10)$$

Often it is useful to directly refer to the matrix of *all* input or of *all* output values. This allows for a very concise formulation in terms of matrix-vector calculus and is also beneficial for the compute time. Here is the definition of the *input data matrix* or the *feature matrix*, as we will call it:

Definition 3.12 (Feature Matrix or Input Data Matrix:) The *feature matrix* $\mathbf{X}$ contains the data of all n features and is a $m \times n$ matrix given by

$$\mathbf{X} = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \quad \text{or:} \quad \mathbf{X} = \begin{bmatrix} | & & | \\ X_1 & \cdots & X_n \\ | & & | \end{bmatrix} . \quad (3.11)$$

The feature matrix is in the statistics literature also called *design matrix*. The *output data matrix* is defined in full analogy:

Definition 3.13 (Output Data Matrix) The *output data matrix*, $\mathbf{Y}$ contains the data of all p output variables and is a $m \times p$ matrix given by

$$\mathbf{Y} = \begin{bmatrix} y_{11} & \cdots & y_{1p} \\ \vdots & \ddots & \vdots \\ y_{m1} & \cdots & y_{mp} \end{bmatrix} \quad \text{or:} \quad \mathbf{Y} = \begin{bmatrix} | & & | \\ Y_1 & \cdots & Y_p \\ | & & | \end{bmatrix}. \quad (3.12)$$

And finally, the whole dataset is represented by the *data matrix* $\mathcal{D}$:

Definition 3.14 (Data Matrix) The data matrix contains the whole dataset and is given by the list of all data points or equivalently by the concatenation of the feature matrix and the output data matrix:

$$\mathcal{D} = [\mathcal{P}^{(1)} \cdots \mathcal{P}^{(n)}]^\top = [\mathbf{X}, \mathbf{Y}], \quad (3.13)$$

which has as many rows as there are data records, and the number of columns is the sum of the number of input features and the number of output values.

A Python example for how to use tabular data is given in Example 3.4, which is the continuation of Example 3.3.

Example 3.4 (Representation of a Measurement Dataset (Part II)) ✂

[This is the continuation of Example 3.3.] In this example we show how we can use numpy to create the different variables and matrices. We start by creating the feature vectors and the target vector as one-dimensional arrays:

```
In [1]: import numpy as np
X1 = np.array([0, 10, 20, 30, 40, 50, 60], dtype=float)
X2 = np.array([1010, 1010, 1010, 1010, 1013, 1013, 1013], dtype=float)
Y = np.array([180, 170, 160, 150, 140, 130, 120], dtype=float)
```

(continued)

We convert them to column vectors, i.e., two-dimensional arrays with only one column. The argument of `reshape` is the new shape of the array. “-1” can occur only once: then the value is automatically determined from the total number of elements and the other dimensions.

```
In [2]: X1 = X1.reshape(-1, 1) # The argument of `reshape` is the dimension of
        X2 = X2.reshape(-1, 1) # the reshaped array. `1` = 1 column
        Y = Y.reshape(-1, 1)  # `-1`: as many rows as necessary
        X1
```

```
Out [2]: array([[ 0.],
               [10.],
               [20.],
               [30.],
               [40.],
               [50.],
               [60.]])
```

Next, we “assemble” the feature matrix $\mathbf{X}$ and the data matrix $\mathcal{D}$:

```
In [3]: X = np.hstack((X1, X2)) # concatenate X1 and X2 horizontally
        D = np.hstack((X, Y))   # concatenate X and Y horizontally
        D
```

```
Out [3]: array([[ 0., 1010., 180.],
               [ 10., 1010., 170.],
               [ 20., 1010., 160.],
               [ 30., 1010., 150.],
               [ 40., 1013., 140.],
               [ 50., 1013., 130.],
               [ 60., 1013., 120.]])
```

And finally, we use our variables for two queries (the questions (iii) and (v) from Example 3.3):

```
In [4]: print("P(2) =", D[1]) # second record
        print("x_3 =", X[2])  # third record of feature matrix
```

```
Out [4]: P(2) = [ 10. 1010. 170.]
        x_3 = [ 20. 1010.]
```

Note that Python indexing starts at zero. Therefore, `D[1]` gives in fact the *second* record.

3.4 Summary and Conclusion

This chapter is the technical foundation for the rest of this book. An important goal was to motivate, clearly define, and explain all used notions and concepts that are required to describe what a dataset is and to explain how we can interact with it using Python or mathematical descriptions. We saw that tabular data is among the most important data “structures” and introduced a number of notations that will be used throughout the whole book. The following box summarizes some of the most important aspects:

Things to Remember



1. Given an arbitrary matrix A that consists of the elements a_{ij} . Then ...
 - the i -th row is given by $a_{i,:} = [a_{i1}, a_{i2}, \dots]$.
 - the j -th column is given by $a_{:,j} = [a_{1j}, a_{2j}, \dots]^T$.
2. The *feature matrix* $\mathbf{X}$ consists of the input data in a “table-like” structure where each row is a data record and each column is a *feature*:
 - the n -th feature is $X_n = [x_{1n}, x_{2n}, \dots, x_{mn}]^T$ where m is the number of records.
 - the k -th input data record is $\mathbf{x}_k = [x_{k1}, x_{k2}, \dots, x_{kn}]$ where n is the number of features.
3. The *output Matrix* follows the same conventions as the feature matrix and is represented by $\mathbf{Y}$.
4. The data matrix $\mathcal{D}$ consists of the feature matrix $\mathbf{X}$ and the output matrix $\mathbf{Y}$ such that $\mathcal{D} = [\mathbf{X}, \mathbf{Y}]$.

3.5 Exercises

3.1 (Sets) Write a Python script that calculates the three set operations shown in Fig. 3.1.

3.2 (Lists) Given are two lists, the first of which is $a = [4, 8, 1, -3, 5]$ and the second $b = [4, 3, 2, 1]$. Use Python to show that the two lists are not identical. Next, print the elements that the two lists have in common as well as those elements that are only in a and only in b .

3.3 (Intervals) Use numpy’s function `linspace` to create the interval of real numbers given by $\{x | 1 < x \leq 2.2 \text{ AND } x = 1 + 0.2n \text{ AND } n \in \mathbb{N}\}$.

Table 3.1 The whole dataset consisting of feature matrix **X** and output matrix **Y**

Input/Features X				Output Y	
X_1	X_2	X_3	X_4	Y_1	Y_2
3	7	9	4	1	-2
4	0	3	2	7	7
6	4	2	0	10	0
9	0	6	6	18	15

3.4 (Row and Column Vectors) Create a matrix *A* that consists of four rows and three columns and contains only 1 as element values. Then set all elements of column number 2 to 2, and set element in row 1 and column 2 to 4. Show the first row of the resulting matrix. Compute the sum of all elements of *A*.

3.5 (Calculating with Input and Output Matrices) Given is a data matrix feature **X** and output **Y** in Table 3.1. Also given is a vector-valued function *f*

$$f : \mathbf{X} \mapsto \mathbf{Y} \quad \Leftrightarrow \quad f \left(\begin{bmatrix} X_1 \\ \vdots \\ X_4 \end{bmatrix} \right) = \begin{bmatrix} 2X_1 - X_3 + X_4 \\ X_1 - 2X_2 + x_3 \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$$

Note that, in this case, the function also could be written as matrix multiplication of **X** with a coefficient matrix. The following questions about **X**, **Y** and *f* help to understand the structure of the used variables, matrices, and the mathematical and computational operations:

1. How many records, feature variables, and outputs does the dataset have? Which is the value of the third record of the second feature?
2. Write down **X** as numpy array and print the number of rows and columns.
3. How can a numpy function and array operations be used to compute **Y**?

References

1. C. Hill. *Learning Scientific Programming with Python*. Cambridge University Press, 2nd edition. ISBN 978-1108745918.
2. Q. Kong, T. Siauw, and A. M. Bayen. *Python Programming and Numerical Methods*. Elsevier, 2021. ISBN 978-0-12-819549-9. DOI <https://doi.org/10.1016/c2018-0-04165-1>.

Materials Science Datasets and Data Generation

4

4.1 Introduction

Machine learning (ML) and data science depend heavily on data. We need data to explain mathematical relations, to investigate statistical properties, and, last but not least, to train our ML models.

While every data scientist should know a few of the famous machine learning datasets, such as the “Iris flower dataset” and the “MNIST dataset of handwritten digits” (introduced below as DS-1 and DS-2), it is necessary to also have datasets that have a direct relevance and interpretation to our scientific field of work—materials science and the neighboring disciplines. For such datasets, we already have some intuition about the meaning of certain aspects or variables. These are the datasets introduced as MDS-1 up to MDS-5.

We intentionally introduce these datasets at this early stage, even before we have introduced how to statistically characterize and describe datasets in general. As a consequence, there are very few “summary statistics” and similar statistical descriptions, because in one of the next chapters, we will use the datasets to introduce what a summary statistic is. We will also describe and explore different aspects of the datasets at various points in this textbook. For additional information on the details of the datasets and the datasets themselves, the reader is referred to the supplementary webpage (<https://MDS-book.org>). A summary of the datasets and how they are used in this book can be found in Table 4.1. Additionally, in a number of places throughout the text, the datasets are used in the text for explanations.

4.2 Dataset MDS-1: Tensile Test with Parameter Uncertainties

A tensile test is one of the most commonly used methods in materials science to determine the effective mechanical material response of a whole specimen. Although the details of the underlying grain, defect, or phase microstructures of

Table 4.1 Overview of the datasets and the usage or purpose

Dataset	Used in		Purpose or Aspect Used
	page	section	
MDS-1 (Tensile Test)	240	11.2.2	instance-based learning
	246	11.5.2	outlier detection
	271	12	regression
	356	13.8	regression (exercises)
	452	16.4	baseline models
	493	17.6	neural networks (exercises)
MDS-2 (Ising Model) and MDS-2 light (Ising Model, small)			
	208	10.2	distribution types
	430	15.6.1	dimensionality reduction (PCA, t-SNE)
	528	18.8	classification, regression (DL)
	533	19.1	network architecture, counting weights
MDS-3 (Phase Microstructure Evolution with the Cahn-Hilliard Model)			
	432	15.6.2	dimensionality reduction (PCA, t-SNE)
	449	16.2.6	cross-validation
MDS-4: Chemical Elements	204	9.5	visualization (EDA)
	435	15.7	PCA, t-SNE, cluster analysis
	437	16.1.1	feature importance
MDS-5: Nanoindentation	202	9.4.4	advanced data visualization (EDA)
	433	15.6.3	cluster analysis (GMM)
DS-1: Iris Flowers	180	9.2.1	initial exploration (EDA)
	202	9.4.4	plot with histograms (EDA)
	175	8.7.4	correlation matrix
	366	14.3	classification
	391	15.2	dimensionality reduction (PCA)
DS-2: Handwritten Digits (MNIST) and DS-2 light (Alpaydin Digits)			
	389	15.1	dimensionality reduction ...
	409	15.3	... and reconstruction with PCA
	435	15.7	dim. reduction (exercises)
	521	18.5.2	classification (DL)

a sample are critical to its mechanical behavior, their effects are strongly averaged in typical macroscopic samples of centimeter size. The material parameters obtained from such tensile tests contain uncertainties that are usually expressed in terms of a mean value and the standard deviation.

In the following, a synthetic dataset of temperature-dependent stress vs. strain data is created. For this purpose, the model equations described in Section 4.2 of [11] are used to provide a phenomenological description of the relation between various material parameters or constants and the stress-strain values. The empirical model was created for structural steel with a nominal yield stress below 450 MPa.

It consists of three terms, the first of which describing the temperature-dependent modulus of elasticity E :

$$E(T) = E_0 \cdot \left(\exp \left[-\frac{1}{2} \left(\frac{\Delta T}{e_3} \right)^{e_1} - \frac{1}{2} \left(\frac{\Delta T}{e_4} \right)^{e_2} \right] \right) \quad \text{with } \Delta T = T - T_0 \quad (4.1)$$

All parameter values used and assumed standard deviations are given in Table 4.2. The second equation defines the temperature dependency of the yield stress F_y :

$$F_y(T) = F_{y0} \left(r_5 + (1 - r_5) \cdot \exp \left[-\frac{1}{2} \left(\frac{\Delta T}{r_3} \right)^{r_1} - \frac{1}{2} \left(\frac{\Delta T}{r_4} \right)^{r_2} \right] \right) \quad (4.2)$$

With these two relations, the true stress σ can now be written as a function of true strain ε and temperature T :

$$\sigma = \begin{cases} E(T) \cdot \varepsilon & \text{for } \varepsilon < \varepsilon_y \\ F_y(T) + (k_3 - k_4 F_{y0}) \exp \left[-\left(\frac{T}{k_2} \right)^{k_1} \right] (\varepsilon - \varepsilon_y(T))^n & \text{for } \varepsilon \geq \varepsilon_y \end{cases} \quad (4.3)$$

where

$$\varepsilon_y = \frac{F_y(T)}{E(T)} \quad (4.4)$$

is the temperature-dependent yield strain.

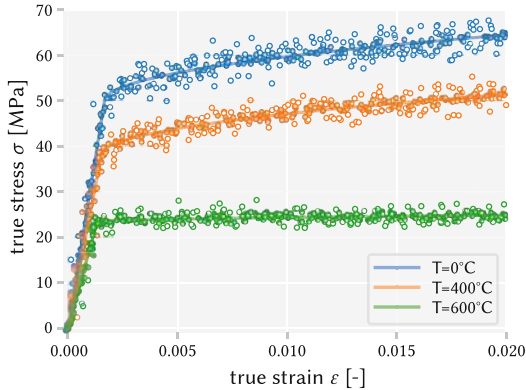
Using the mean values of all material and model parameters given in Table 4.2, we obtain the average stress-strain response shown as solid lines in Fig. 4.1. Each data point was obtained by drawing random values from normal distributions given by the mean and standard deviation in Table 4.2, resulting in varying degrees of noise in the curves. This represents the (assumed) uncertainty with which parameters used were determined or measured. Of course, this is an academic example, but it is particularly useful for the following reasons:

- There is a mathematical function that describes the “true” stress-strain curve (i.e., the data obtained from the mean values)—and it is known to us.
- The resulting data can be used as a dataset with only one input variable (i.e., stress) and one output variable (i.e., strain) for a fixed temperature. It can also be used with two input variables, e.g., stress and temperature, and one output variable.
- The dataset can be divided into different parts that can be used separately. For a simple regression model, for example, one would start with the linear part, while

Table 4.2 Parameters used to generate the synthetic stress-strain-temperature data. The mean value and standard deviation define the range from which a parameter value is randomly selected

Material & model parameters			Description
variable	mean value	std. dev.	
E_0	206 GPa	7 GPa	Young’s modulus at ambient temperature
F_{y0}	345 MPa	12 MPa	Yield stress at ambient temperature
T_0	20 °C	0 °C	Ambient temperature
e_1	3.768	–	Material dependent model parameter for
e_2	1.0	–	Young’s modulus $E(T)$
e_3	1639 °C	–	
e_4	1650 °C	–	
r_1	7.514	–	Material dependent model parameter for
r_2	1.0	–	Yield stress $F_y(T)$
r_3	588 °C	–	
r_4	676 °C	–	
r_5	0.09	–	
k_1	7.82	–	Material dependent model parameter for
k_2	540 °C	–	True stress σ
k_3	1006 MPa	–	
k_4	0.759	–	
n	0.503	0.015	Strain hardening exponent

Fig. 4.1 The solid lines show the average stress-strain response, while the markers show the stress-strain values, when the material or model parameter is sampled from a normal distribution given by the standard deviations in Table 4.2



- for more complicated models, the whole dataset consisting of both linear and nonlinear parts, could be used.
- A number of characteristics of the dataset have a physical meaning, e.g., Young’s modulus, yield stress, etc.

In the following, this dataset will be referred to as the “MDS-1 (Tensile Test)” dataset. The Python code for producing the data as well as the data itself can be obtained from the supplementary webpage (<https://MDS-book.org>).

4.3 Dataset MDS-2: Microstructure Evolution with the Ising Model

The Ising model is a model of statistical physics originally developed by W. LENZ [8] to represent aspects of ferromagnetism, namely, the evolution of the ferromagnetic domain walls. ERNST ISING, to whom the model owes its name, was the first to solve it mathematically for a one-dimensional situation [5]. However, the two-dimensional case leads to much more complex behavior; in particular, it can describe a disorder-order phase transition. The Ising model in general has been studied by theoretical physicists for decades. It is also used as a simulation model in a number of different use cases in physics and materials science.

The following simulation was performed in the context of predicting the structure-property relation using ML approaches by NGUYEN et al. [9]. The dataset is obtained from a two-dimensional simulation model of an $N \times N$ lattice (Fig. 4.2). A magnetic dipole is stored on each of the lattice points. Neighboring dipoles interact with each other. The energy of a simplified system is given by the Hamiltonian

$$H = -J \sum_{\langle i, j \rangle} \sigma_i \sigma_j, \quad (4.5)$$

where we assumed that no external magnetic field is applied and that the interaction strength $J = 1$ between all neighboring dipoles is the same. The indices i and j include all lattice sites, with numbers ranging from 1 to N^2 , and with the additional constraint that in the sum only the nearest neighbors (indicated by $\langle i, j \rangle$) are considered. $\sigma \in \{-1, 1\}$ is the sign of the magnetic dipole at a given site.

To simulate such a system with periodic boundary conditions, we use the Metropolis Monte Carlo algorithm. First, we randomly initialize the spin (up or down, i.e., $+1$ and -1) of all dipoles. Then, we randomly select a site $s(i, j)$ with $i, j \in 1, \dots, N$, flip the corresponding dipole, and calculate the energy of the new

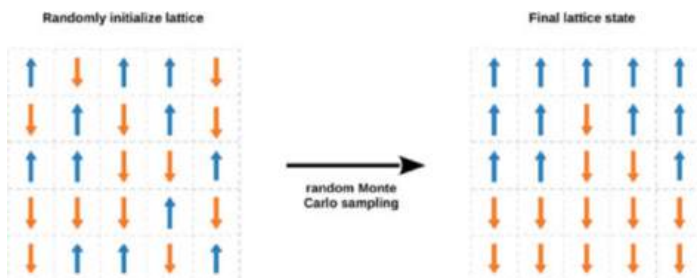
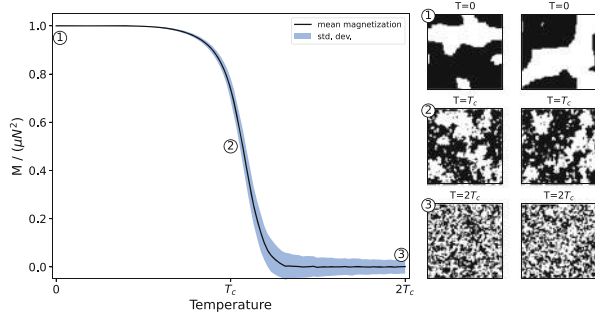


Fig. 4.2 Sketch of two states of an $N \times N$ lattice during a simulation with the Ising model. The left panel shows the initial state, where the spin directions (small arrows) are chosen randomly. After a series of Monte Carlo update steps at a fixed temperature T , the domain walls between the regions of the two spins become visible

Fig. 4.3 Ising model: the left panel shows the evolution of the effective magnetization obtained from the corresponding microstructure as a function of temperature. Examples of microstructures for three different temperatures are shown on the right. The image size is 64×64 pixel



configuration. If the energy of the new configuration is smaller than the previous one, we keep the new configuration; if the energy is greater than the previous one, we only keep it with a probability

$$p = \exp\left(-\frac{\Delta E}{k_b T}\right), \quad (4.6)$$

where T is the prescribed temperature of the system, ΔE the energy difference between the previous and the current state of the system, and k_b is the Boltzmann constant, scaled to unity for simplicity. We repeat those steps N^3 times and then terminate the simulations. The reasoning behind this stopping criterion is the following: after N^3 steps, every site of the $N \times N$ lattice has been visited approximately N times, so that the information had enough time to pass through the whole lattice. Without such a stopping criterion, the two “phases” would eventually merge completely and only one phase would occupy the entire area.

Each of the spin configurations is saved as a black and white image, in which a value of 0 and 255 denotes a negative and a positive spin, respectively. The images are generated from within the range of $[0, 2T_c]$, where the CURIE temperature T_c (named after the physicist MARIA SKŁODOWSKA-CURIE) is given as $T_c = 2J/(k_B \ln(1 + \sqrt{2}))$. With the above scaling of $k_B = 1$ and $J = 1$, we obtain $T_c \approx 2.26$. Figure 4.3 shows examples of resulting microstructures.

The dataset used for data analysis in this text consists of two variants. The first variant, MDS-2 (Ising model), consists of images of 64×64 pixel, given as PNG files. Each file is also accompanied by the information regarding the temperature at which the simulation was performed. A variant of this dataset, MDS-2 light (Ising model, small), consists of a smaller number images that additionally are only 16×16 pixel in size.

4.4 Dataset MDS-3: Cahn-Hilliard Model

Binary mixtures, such as alloys of two metals, may coarsen into two phases, a mechanism that is described by the *spinodal decomposition*. This phenomenon may

occur if an initially homogeneous phase becomes thermodynamically unstable. In this case, small fluctuations quickly grow, as described by [2].

In contrast to the dataset MDS-2 (Ising model), there is no randomness in the evolution of this system; it is completely governed by continuum equations, i.e., the Cahn-Hilliard *equation*, which governs the evolution of the concentration of a phase:

$$\frac{\partial c}{\partial t} = M_c \nabla^2 \frac{\delta E}{\delta c}, \quad (4.7)$$

with the free energy E and a mobility coefficient of the interface M_c . The free energy density ψ consists of terms for the potential, gradient, and elastic energy density:

$$\psi = \psi^{\text{bulk}} + \psi^{\text{grad}} + \psi^{\text{el}} \quad (4.8)$$

where $\psi^{\text{bulk}} = c_0 c^2 (1 - c)^2$, $\psi^{\text{grad}} = \frac{1}{2} k_c |\nabla c|^2$ and $\psi^{\text{el}} = \frac{1}{2} \boldsymbol{\sigma} : \boldsymbol{\epsilon}^{\text{el}}$. The two constants c_0 and k_c are the density scale and the gradient energy density, respectively. $\boldsymbol{\sigma}$ is the stress tensor and $\boldsymbol{\epsilon}^{\text{el}}$ the elastic strain tensor. The energy functional is then

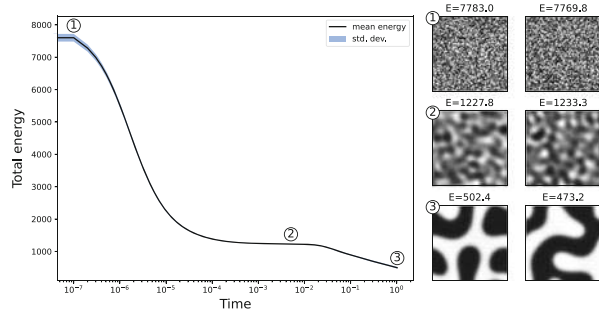
$$E = \int_{\Omega} \psi \, d\Omega = \int_{\Omega} c_0 c^2 (1 - c)^2 + \frac{1}{2} k_c |\nabla c|^2 + \frac{1}{2} \boldsymbol{\sigma} : \boldsymbol{\epsilon}^{\text{el}} \, d\Omega \quad (4.9)$$

The elastic contribution is determined by fulfilling the mechanical equilibrium equation, $\nabla \cdot \boldsymbol{\sigma} = \nabla \cdot \mathbb{C} : \boldsymbol{\epsilon}^{\text{el}} = \mathbf{0}$ where $\mathbb{C}$ is the fourth-order stiffness tensor.

Finite element method (FEM) simulations with the above equations and with periodic boundary conditions were carried out to create the dataset. The width and the height of the domain are both assumed to be 20 μm . The elastic constants are assumed to be $C_{11} = 198 \text{ GPa}$, $C_{12} = 138 \text{ GPa}$ and $C_{44} = 97 \text{ GPa}$. The density scale is $c_0 = 50 \cdot 10^{-6} \text{ J}\mu\text{m}^{-3}$, and the gradient energy density is $k_c = 10 \text{ J}\mu\text{m}^{-1}$. Examples of typical microstructures along with the corresponding total energy as a function of time are shown in Fig. 4.4. There, the “kink” at 2. is caused by the microstructure undergoing rapid changes and coarsening (note the logarithmic scaling of the time axis).

The simulation is performed for a duration of 20 s with adaptive time stepping. Initial values are sampled from a uniform random distribution with values between 0 and 1, which is the only reason for the random-looking microstructures since the evolution equations are deterministic. The dataset consists of the images from the entire simulation process. Each image contains the phase microstructure encoded as grayscale values. In addition, there is also a text file containing the time and total energy value for each image. The image data is exported at every step of the first part and every 10th step of the second part of the simulation. Therefore, the generated datasets contain about 60,000 images. For further details and discussions, see the publications by NGUYEN et al. [9].

Fig. 4.4 Examples of microstructures obtained from the Cahn-Hilliard model. Three different snapshots in time are shown together with the corresponding values of the total energy



4.5 Dataset MDS-4: Properties of Chemical Elements

This is a small dataset in which Ferreira et al. [3] collected the four periodic properties (i.e., atomic radius, electron affinity, ionization energy, and the electronegativity) of a total of 38 chemical elements (22 metals and 16 non-metals) from a number of available sources. The data is shown in Table 4.3. When analyzing the data, one tries to find out whether two of these properties are related or whether it is possible to predict if a material is a metal or not, based on some given properties. The data can be simply copied either from Table 4.3 or from the supplementary webpage (<https://MDS-book.org>).

4.6 Dataset MDS-5: Nanoindentation of a Cu-Cr Composite

This dataset was obtained during nanoindentation of different Cu-Cr composites containing 25 wt% Cr and 60 wt% Cr corresponding to 29.95 at% and 64.40 at% Cr, respectively, as well as pure Cu and Cr as reference samples. The aim of this study was to investigate the extent to which the properties of the two materials can be distinguished based on indentation experiments. Another question was how many different “groups” of properties, i.e., elastic modulus and hardness, can be detected. For example, whether there are only two distinct elastic property sets corresponding to the two pure metals (because Cu-rich regions should have a different hardness or elastic modulus than Cr-rich regions) or whether there are more than two such property groups. The latter would indicate that there are additional effects, e.g., interphases and phase boundaries, contributing to the mechanical behavior.

The investigation pertaining to this dataset was published by ZHANG et al. in [12]; some of the results and the datasets are shown here, with permission of the authors. Figure 4.5 shows typical microstructures of the composites investigated. An overview of the dataset is given in Fig. 4.6, showing the full dataset as experimentally acquired including “outlier.” The outliers were eliminated, resulting in the dataset shown in the right panel of Fig. 4.6. For each of the four materials, the goal was to identify patterns and structures in the data distribution, which can then be explained on the basis of microstructural aspects. The dataset itself consists

Table 4.3 Four periodic properties and group for a selection of chemical elements after [3]

Element	Atomic radius (pm)	Electron affinity (kJ mol ⁻¹)	Ionization energy (kJ mol ⁻¹)	Electro-negativity	Group
Metals					
Li	1.82	59.63	520.22	0.98	1
Na	2.27	52.87	495.85	0.93	1
K	2.75	48.39	418.81	0.82	1
Rb	3.03	46.88	403.03	0.82	1
Cs	3.43	45.51	375.71	0.79	1
Fr	3.48	44.38	392.96	0.70	1
Be	1.53	−66.00	899.51	1.57	1
Mg	1.73	−67.00	737.75	1.31	2
Ca	2.31	2.37	589.83	1.00	2
Sr	2.49	4.63	549.47	0.95	2
Ba	2.68	13.95	502.85	0.89	2
Ra	2.83	9.65	509.29	0.90	2
Al	1.84	41.76	577.54	1.61	13
Ga	1.87	41.49	578.85	1.81	13
In	1.93	28.90	558.30	1.78	13
Tl	1.96	36.38	589.35	1.80	13
Ge	2.11	118.94	762.18	2.01	14
Sn	2.17	107.30	708.58	1.96	14
Pb	2.02	35.12	715.60	1.80	14
Sb	2.06	100.92	830.58	2.05	15
Bi	2.07	90.92	702.94	1.90	15
Po	1.97	183.30	811.83	2.00	16
Nonmetals					
H	1.10	72.77	1312.05	2.20	1
B	1.92	26.99	800.64	2.04	13
C	1.70	121.78	1086.45	2.55	14
Si	2.10	134.07	786.52	1.90	14
N	1.55	−7	1402.33	3.04	15
P	1.80	72.04	1011.81	2.19	15
As	1.85	77.57	944.46	2.18	15
O	1.52	140.98	1313.94	3.44	16
S	1.80	200.41	999.59	2.58	16
Se	1.90	194.97	940.96	2.55	16
Te	2.06	190.16	869.29	2.10	16
F	1.47	328.17	1681.05	3.98	17
Cl	1.75	348.58	1251.19	3.16	17
Br	1.85	324.54	1139.86	2.96	17
I	1.98	295.15	1008.39	2.66	17
At	2.02	270.20	1037.00	2.20	17

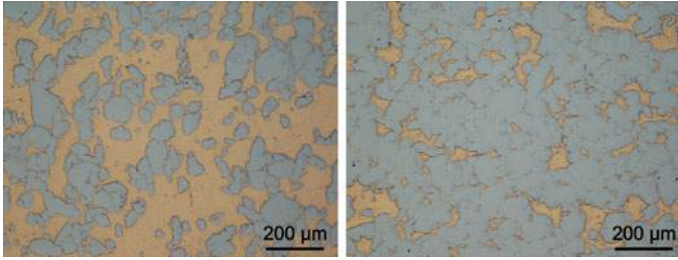


Fig. 4.5 Cu-Cr composites produced by field-assisted sintering technique with 25 wt% Cr shown in the left panel, and 60 wt% Cr shown in the right panel

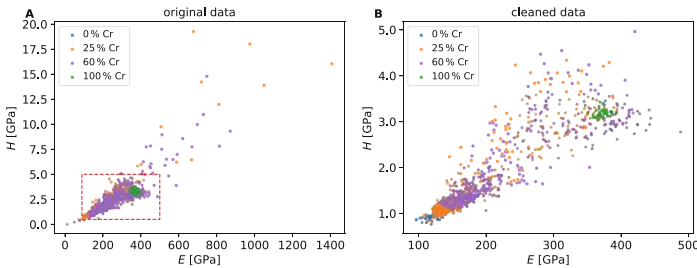


Fig. 4.6 The experimentally obtained distributions of Young's modulus E and hardness H for 0 wt%, 25 wt%, 60 wt% and 100 wt% Cr content. (Left panel: the experimental data in its original form. Right panel: cleaned and preprocessed dataset used in the ML analyses (the rectangle in the left panel indicates the region shown in the right panel))

of tabular data in which the hardness and the elastic modulus recorded for different indentation depths are presented. The original dataset as shown in the left panel of Fig. 4.6) as well as the already preprocessed dataset with outliers removed (as shown in the right panel of Fig. 4.6) can both be downloaded from the supplementary webpage (<https://MDS-book.org>) or obtained from zenodo [13].

4.7 Dataset DS-1: The Iris Flower Dataset

The “Iris Flower Dataset,” sometimes called “Fisher’s Iris Dataset” [4], is one of the most famous datasets used as an introductory example for ML. It consists of the measured width and length of petal and sepal of iris flowers (the features) together with the corresponding name of the species (the label or target). The aim is to find the name of the species based on the length measurements.

Some exemplary data is shown in the table in Fig. 4.7a. There, additional numbers (the class IDs) have been assigned to the three species names. This mapping of text labels (e.g., “Iris-setosa”) to integer numbers makes it much easier for ML models, since such models are based on numbers anyways.

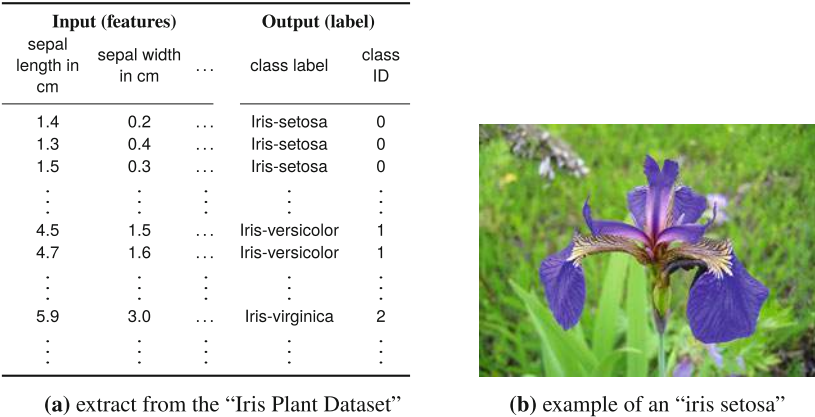


Fig. 4.7 The Iris dataset: the table shows an excerpt from the dataset with the class ID as an additional column. The photograph shows an *iris setosa*. The smaller violet “leaves” at the top are the sepals and the larger ones the petals. (a) Extract from the “Iris Plant Dataset”. (b) Example of an “iris setosa”

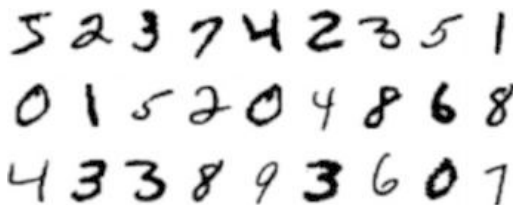
The dataset consists of 4 features, 50 records, and a single output variable, i.e., the name of the flower. To turn this textual label into a number, we assign an integer number to each flower species (the class IDs 0, 1, and 2), as shown in Fig. 4.7a. The dataset can be obtained, e.g., through the Python package `scikit-learn`.¹ It is also available through the supplementary webpage (<https://MDS-book.org>).

4.8 Dataset DS-2: The Handwritten Digits Dataset

Identifying the correct number displayed in an image with handwritten digits has long been one of the benchmark tests in ML. ML methods required benchmark data to compare the performance of the different methods. Therefore, the *MNIST database of handwritten digits* [6] was introduced in the 1990s by Lecun et al. [7] in their seminal paper introducing a revolutionary convolutional neural network for handwritten digit recognition. Since then this has become one of the standard dataset for studying and benchmarking different ML approaches. The data itself consists of 70,000 training and test images with labels. The images are 28×28 pixels in size, and each contains a handwritten number from the range 0 to 9. For more details, such as how the dataset was acquired and preprocessed, see Lecun et al. [7]. Figure 4.8 shows a number of examples from this datasets. For some deep learning (DL) tasks, the dataset is already quite large. Therefore, sometimes a second dataset is used, consisting of only 1797 examples, and the images are 8×8 pixels. The dataset was

¹ The Python ML package `scikit-learn` [10] can be found at <https://scikit-learn.org>

Fig. 4.8 The first 27 images in the MNIST database, which contains a total of 70,000 examples of handwritten digits. Each grayscale image has a resolution of 28×28 pixels



created by E. ALPAYDIN [1] and can also be obtained either from scikit-learn or from the supplementary webpage (<https://MDS-book.org>).

The MNIST dataset is referred to as DS-2 (MNIST), while the dataset of ALPAYDIN is referred to as DS-2-light (Alpaydin Digits).

4.9 Online Resource for Obtaining Training Data

What could you do to find new and exciting datasets for your ML experiments? There is a wide range of different datasets available on the Internet. However, not all datasets are well curated or suitable for ML, and only a small fraction of them are from the field of materials science. Below we list some starting points for searching for datasets. First, there are the repositories that contain generic ML data:

- The *UC Irvine Machine Learning Repository* at <https://archive.ics.uci.edu/> contains a large variety of documented datasets, including some of the “classical” datasets. Datasets have a DOI and are referenceable.
- <https://dasl.datadescription.com/> is another repository where you can search different categories of datasets.
- A huge repository of annotated images mainly for DL can be found at <https://www.image-net.org/>.
- Another interesting platform is <https://paperswithcode.com/>. When a particular ML method has been published, the code and sometimes also the data from it can often be retrieved there. The website also contains a section of a dataset collection that can be browsed and searched.
- An alternative to the Iris dataset is the “Palmer Penguins” dataset, currently hosted at <https://github.com/allisonhorst/palmerpenguins>.

If you are looking for (materials) science datasets, a little more time is usually required to find suitable data:

- zenodo.org is a good starting point for published datasets. But these are not uploaded with the machine learner in mind, so you need some patience and a bit of browsing. Unfortunately, many people who upload their data have no particular need for documentation.

- Many scientific journals require or at least allow that along with a publication, the data is also published, either in the form of a separate data publication or in the form of supplementary material. This can sometimes be helpful, e.g., the journal *npj Computational Materials* (<https://www.nature.com/npjcompumats/>) is one of the journals where authors are more likely to publish their data and code than not.

References

1. E. Alpaydin and C. Kaynak. Optical Recognition of Handwritten Digits. UCI Machine Learning Repository, 1998. DOI <https://doi.org/10.24432/C50P49>.
2. J. W. Cahn. On spinodal decomposition. *Acta metallurgica*, 9(9):795–801, 1961. DOI [https://doi.org/10.1016/0001-6160\(61\)90182-1](https://doi.org/10.1016/0001-6160(61)90182-1).
3. J. E. V. Ferreira, M. T. S. Pinheiro, W. R. S. dos Santos, and R. da Silva Maia. Graphical representation of chemical periodicity of main elements through boxplot. *Educación Química*, 27(3):209–216, July 2016. DOI <https://doi.org/10.1016/j.eq.2016.04.007>.
4. R. A. Fischer. The Use of multiple Measurements in Taxonomic Problems. *Annals of Eugenics*, 7(2):179–188, Sept. 1936. DOI <https://doi.org/10.1111/j.1469-1809.1936.tb02137.x>.
5. E. Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, Feb 1925. ISSN 0044-3328. DOI <https://doi.org/10.1007/BF02980577>.
6. Y. LeCun, C. Cortes, and C. J. Burges. The mnist database of handwritten digits. URL <http://yann.lecun.com/exdb/mnist/>.
7. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. DOI <https://doi.org/10.1109/5.726791>.
8. W. Lenz. Beitrag zum verständnis der magnetischen erscheinungen in festen körpern. *Z. Phys.*, 21:613–615, 1920.
9. B. D. Nguyen, P. Potapenko, A. Demirci, K. Govind, S. Bompas, and S. Sandfeld. Efficient surrogate models for materials science simulations: Machine learning-based prediction of microstructure properties. *Machine Learning with Applications*, 2024. DOI <https://doi.org/10.1016/j.mlwa.2024.100544>.
10. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
11. M. Seif, J. Main, J. Weigand, F. Sadek, L. Choe, C. Zhang, J. Gross, W. Luecke, and D. McColskey. Temperature-dependent material modeling for structural steels: Formulation and application. Technical report, Apr. 2016. DOI <https://doi.org/10.6028/nist.tn.1907>.
12. C. Zhang, C. Bos, S. Sandfeld, and R. Schwaiger. Unsupervised learning of nanoindentation data to infer microstructural details of complex materials, 2023. DOI <https://doi.org/10.48550/arXiv.2309.06613>.
13. C. Zhang, C. Bos, S. Sandfeld, and R. Schwaiger. Nanoindentation Datasets of Copper-Chromium Composites, 2023. DOI <https://doi.org/10.5281/zenodo.8336072>.

Part II

A Primer on Probabilities, Distributions, and Statistics



“The true logic of this world lies in the calculus of probabilities.”

James Clerk Maxwell (1831–1879),

Scottish mathematician and scientist

Probability deals with predicting the likelihood of future events, while statistics involves the analysis of the frequency of past events. Without these two fields, it would neither be possible to characterize nor to understand datasets or certain machine learning algorithms (e.g., Bayesian methods strongly rely on *conditional probabilities*). To *count events*, to create combinations of events, and to compute the probability of occurrence of particular events are important prerequisites for probability and statistics. Therefore, in Sect. 5.1, we start by introducing the foundations of *combinatorics* together with the most important mathematical formulations. We will then continue with deriving and computing *probabilities* (Sect. 5.2) using discrete and continuous random variables (Sect. 6.1).

5.1 Combinatorics

Combinatorics is a field of mathematics and the foundation for many considerations in probability theory. It is mainly concerned with counting and arrangement of things (e.g., numbers, tokens, coins, and all other kinds of objects). The best-known example is probably that of a coin toss: assuming that you are tossing a coin five times, how many different outcomes (i.e., sequences of head or tail) are possible and what are they? Throwing two dice is another example —which are the summed values that can occur and how many times *do* they occur? A famous example from physics and materials science comes from the context of BOLTZMANN’s statistical thermodynamics. But first, we need to introduce some of our terminology:

Definition 5.1 (Experiment, Event, and Sampling).

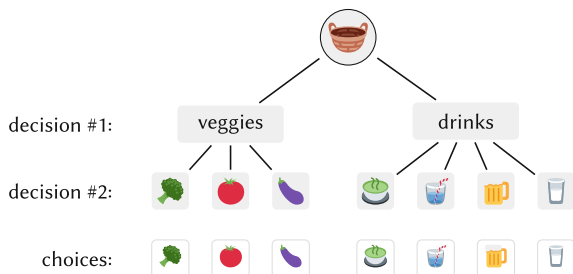
- A random **experiment** can have several outcomes. For example, blindly choosing one token out of three (a red, a blue, and a green one) is an experiment. Another experiment is to initially choose one token, keep it, and choose another one (out of the remaining two), which leaves us with two tokens in our hand.
- An **event** is the possible outcome of a random experiment. For example, in the first example, choosing a red ball is an event, as is choosing a blue one, etc. In the second example, picking a red and a green ball also represents *one* event.
- The procedure of **sampling** (or *to draw a sample*) refers to picking an element from a set of elements. Most of the time, *random sampling* is performed, when elements are chosen (or “drawn”) at random.

We will encounter these terms again, when we introduce probabilities and will still have to refine them a bit. But for now, the above definitions are sufficient to get started. Counting things or events nearly always comes down to one of two principles, i.e., the *additive principle* and the *multiplication principle*, which we introduce next.

5.1.1 Counting with the Additive Principle

We start with a simple problem: assume you have three different types of vegetables and four different types of drinks, and you want to have *either* a veggie *or* a drink in your picnic basket. How many items can you choose from? The answer is quite simple: just add the two numbers, and you get $3 + 4 = 7$ choices. Figure 5.1 shows the answer using a tree diagram. To demonstrate the additive principle, a tree diagram is more complicated than necessary to understand the principle.

Fig. 5.1 How many single-item choices are there if you have three different types of vegetables and four different types of drinks? The tree diagram shows the two groups; the seven possible choices are shown as small framed boxes below the tree



However, in Sect. 5.1.2, the usefulness of such a visualization will become clearer. This example can be generalized to the *additive principle*:

Definition 5.2 (Additive Principle). Assume that an event or random experiment A can occur in m different variants, and another event or random experiment B can occur in n different and *disjoint* ways. Then the event “ A or B ” can occur in $m + n$ different ways.

The word *disjoint* indicates an important aspect: A and B cannot happen at the same time and are mutually exclusive. The above definitions of the additive principle can easily be extended to more than two events or experiments in the obvious way.

5.1.2 More Counting with the Multiplication Principle

To explain the multiplicative principle, we again use a simple example: assume that you want to pack your picnic basket and that you can choose one piece of vegetable and one drink. The following veggies are available: broccoli, tomato, and eggplant. The drinks you can choose from are hot tea, water, beer, and a glass of milk. How many different snack combinations can you create?

To answer this question, one can arrange the two decisions in a tree diagram, as shown in Fig. 5.2. With this structure, the answer is rather straightforward. For each of the three veggies, there are four possibilities for choosing a drink, which makes altogether $3 \cdot 4 = 12$ different combinations. In such a tree diagram, each complete path from top (the “root” or “stem”) to bottom (the “leaves” that do not have any

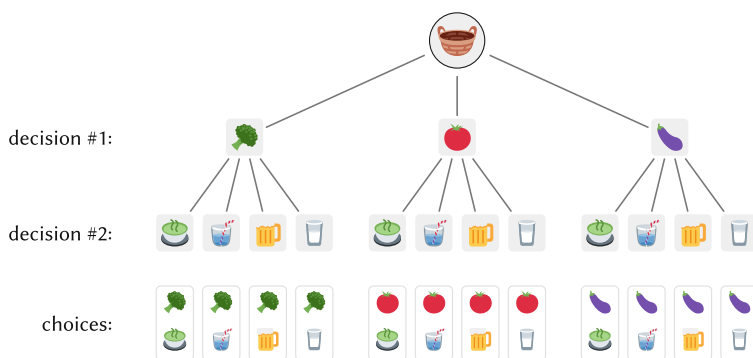


Fig. 5.2 How many different combinations of one veggie and a drink are there? For each of the three veggies, there are four possible drinks. The solid lines indicate one possible path each, i.e., a particular choice of veggie and drink. All possible combinations are shown in the small boxes below the tree

further connections) represents one possible combination (or event). We can more formally define the multiplicative principle as follows:

Definition 5.3 (Multiplication Principle). Assume that an event or random experiment consists of two sub-events or sub-experiments, A and B . The multiplicative principle states that if A has m different outcomes, and each of these m outcomes for A allows for exactly n different outcomes for B , then the overall event or experiment “ A and B ” can occur in $m \cdot n$ different ways.

This principle can also be generalized to more than two events or experiments, as demonstrated in Example 5.1. Note that in the above definition, the word “and” in “ A and B ” is not identical to the mathematical “times.”

Example 5.1 Multiplication Principle and Parameter Studies



Assume that you want to perform a parameter study using a simulation software. There are five different values for parameter 1, three for parameter 2, and four for parameter 3 to be used in this study. How many simulation runs need to be performed?

Just multiply the number of simulations per parameter. Altogether, we then need to run

$$5 \cdot 3 \cdot 4 = 60$$

simulations to perform the study. Due to the multiplication, including more parameter(s) can therefore quickly result in very large numbers of simulations to be performed!

Why is it important to know how many different outcomes exist or how often they occur? Because this kind of counting is crucial, when we need to calculate a probability. For example, in many cases, probabilistic experiments result in outcomes of equal likelihood. The probability then is also the same for each of these outcomes, defined as $1/N$, with N as the total number of possible outcomes. In the case of rolling a six-sided die, for example, the probability p to throw any value of $1 \dots 6$ is $p = 1/6$. Here, “counting” refers to figuring out that six different outcomes exist. Once we use more than one die, it becomes obvious that “counting” is involved.

We will now turn to introducing the mathematical basis, the *factorials*, and then continue with the two big classes of “experiments” in combinatorics: permutations and combinations.

5.1.3 Factorials

The *factorial* is a number that shows up in many different situations, in which probabilities are involved, namely, whenever the product of the first n integers is required. This multiplication operation is abbreviated by an exclamation mark (which might seem like an odd choice):

Definition 5.4 (Factorial). The factorial $n!$ of an integer number $n > 0$ is given by

$$n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1 . \quad (5.1)$$

For $n = 0$ the definition $0! := 1$ is used.

The definition of $0! := 1$ might look surprising but, as it turns out, makes a lot of calculations much easier. For example, with this definition the factorial can also be written using the next smaller factorial:

$$n! = n \cdot (n - 1)! \quad (5.2)$$

Factorials already have quite a long history and were discovered by a number of ancient cultures a long time ago. Many different fields could not do without them. The first seven factorials (starting with the definition for $0!$) are as follows:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \\ 2! &= 1 \cdot 2 = 2 \\ 3! &= 1 \cdot 2 \cdot 3 = 6 \\ 4! &= 1 \cdot 2 \cdot 3 \cdot 4 = 24 \\ 5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \\ 6! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720 \end{aligned}$$

It is quite obvious that with increasing n , the value of $n!$ quickly explodes. Thus, for large n , computations quickly become unfeasible. *Stirling's approximation* gives a good approximation of the values in such cases. One of the most frequently used versions is the following:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n , \quad (5.3)$$

where $e \approx 2.7183 \dots$ is the Euler number, and the approximation approaches the true value for $n \rightarrow \infty$. Another form often used is:

$$\ln(n!) \approx n \ln(n) - n, \quad (5.4)$$

which is also the form used in the famous Boltzmann equation from statistical mechanics, where the so-called *microstates* and *macrostates* need to be counted.

In Python, factorials can be computed using a function from the `math` module:

```
In [1]: from math import factorial
```

```
In [2]: factorial(12)
```

```
Out [2]: 479001600
```

5.1.4 Permutations—Ordered Sets Without Replacement

Permutations of the objects in a set (the definition of a *set* is given in Definition 3.2) denote the arrangement of these objects in particular sequences or orders. Technically, the result is a set of tuples, as for the resulting permutations the order of elements matters. An important question is as follows: *How many different permutations of a particular set of objects do exist?*

Quite unsurprisingly, also a concise mathematical definition exists. However, first we explain the concept by giving a number of examples described in Example 5.2.

Example 5.2 Permutation Examples



1. For a *set with one object*, $\{42\}$, only one way of sorting a single element exists: it is the set that consists of only one tuple, here:

$$\{(42)\}$$

(which clearly are a lot of brackets enclosing just a single number!).

2. For a *set with two objects*, $\{D, C\}$, two ways of sorting two elements exist (here: the letters D and C); the permutations consist of the set of the two tuples:

$$\{(D, C), (C, D)\}$$

(continued)

3. For a set with three objects, $\{\blacktriangle, \blacksquare, \bullet\}$, there are six different ways of sorting three elements. It is the set that consists of the following tuples:

$$\{(\blacktriangle, \blacksquare, \bullet), (\blacktriangle, \bullet, \blacksquare), (\blacksquare, \blacktriangle, \bullet), (\blacksquare, \bullet, \blacktriangle), (\bullet, \blacksquare, \blacktriangle), (\bullet, \blacktriangle, \blacksquare)\}.$$

4. A set with four items $\{0, 1, 2, 3\}$ has the following 24 permutations:^a

$$\begin{aligned} &\{(0, 1, 2, 3), (1, 0, 2, 3), (2, 0, 1, 3), (3, 0, 1, 2), \\ &\quad (0, 1, 3, 2), (1, 0, 3, 2), (2, 0, 3, 1), (3, 0, 2, 1), \\ &\quad (0, 2, 1, 3), (1, 2, 0, 3), (2, 1, 0, 3), (3, 1, 0, 2), \\ &\quad (0, 2, 3, 1), (1, 2, 3, 0), (2, 1, 3, 0), (3, 1, 2, 0), \\ &\quad (0, 3, 1, 2), (1, 3, 0, 2), (2, 3, 0, 1), (3, 2, 0, 1), \\ &\quad (0, 3, 2, 1), (1, 3, 2, 0), (2, 3, 1, 0), (3, 2, 1, 0)\} \end{aligned}$$

^aTip: For writing down all permutations, it generally helps to do this in a systematic manner, e.g., always keep the rightmost item as the one that changes “fastest” and the leftmost as the one that changes the slowest, similar to four nested `for`-loops in Python.

From these examples, we can infer the rule for constructing the different permutations. We summarized these steps in Algorithm 5.1 and have followed them in the fourth example of Example 5.2. In the first column of tuples, the 0 is fixed, and the numbers 1, 2, and 3 are permuted. In the second column, the 1 is kept on the left, and again the remaining numbers 0, 2, and 3 are permuted. The two last columns are processed analogously.

Algorithm 5.1: Systematic Approach for Constructing All Permutations



These are the steps required for systematically finding all possible permutations of a set of n elements:

1. Choose one of the n elements and keep it, e.g., as the leftmost element “fixed”;
2. For the remaining “free” elements, the problem to be solved is simpler and consists only of the permutation of $n - 1$ elements;
3. Apply the same strategy as in 1. to the new problem until only one element is left.

How can this algorithmic strategy be generalized systematically? The multiplication principle again comes in handy. We introduce a variable P_N for the number of possible permutations¹ k for a set of N elements as $P_N = k$. For example, for a set of only one element, only one permutation is possible and $P_1 = 1$. A set with three objects has $P_3 = 6$, and $P_4 = 24$. We already saw how a permutation of N elements can be reduced to a permutation of $n - 1$ elements together with keeping the first element fixed. Thus:

$$\begin{aligned}
 P_N &= N \cdot P_{N-1} \\
 &= N \cdot (N - 1) \cdot P_{N-2} \\
 &= N \cdot (N - 1) \cdot (N - 2) \cdot P_{N-3} \\
 &\vdots \\
 &= N \cdot (N - 1) \cdot (N - 2) \cdots 2 \cdot P_1 \\
 &= N \cdot (N - 1) \cdot (N - 2) \cdots 2 \cdot 1 = N!
 \end{aligned} \tag{5.5}$$

That's why in the context of permutations, a factorial will always pop-up somewhere.

It is often useful to visualize and count the number of possible permutations using a *permutation tree* as shown in Fig. 5.3. It can be seen that for the set of numbers $\{0, 1, 2, 3\}$, initially, there are four possibilities—the four elements of the set. Then, there are three and two possibilities and finally only one possibility. Altogether, this makes $4! = 24$ combinations. This argument can easily be extended to more complex situations, such as the one discussed in Example 5.3 and Example 5.4.

Example 5.3 Permutations of atom positions



Assume that in a crystal there are nine vacancies in which nine atoms should be placed. How many different configurations exist for placing the atoms?

For the first atom, there are nine possibilities to place it, for the second one there are 8, etc. Altogether there are $9! = 362880$ different arrangements.

¹ In later chapters, we will use P for probabilities, but it will always be clear from the context if probabilities are meant or permutations.

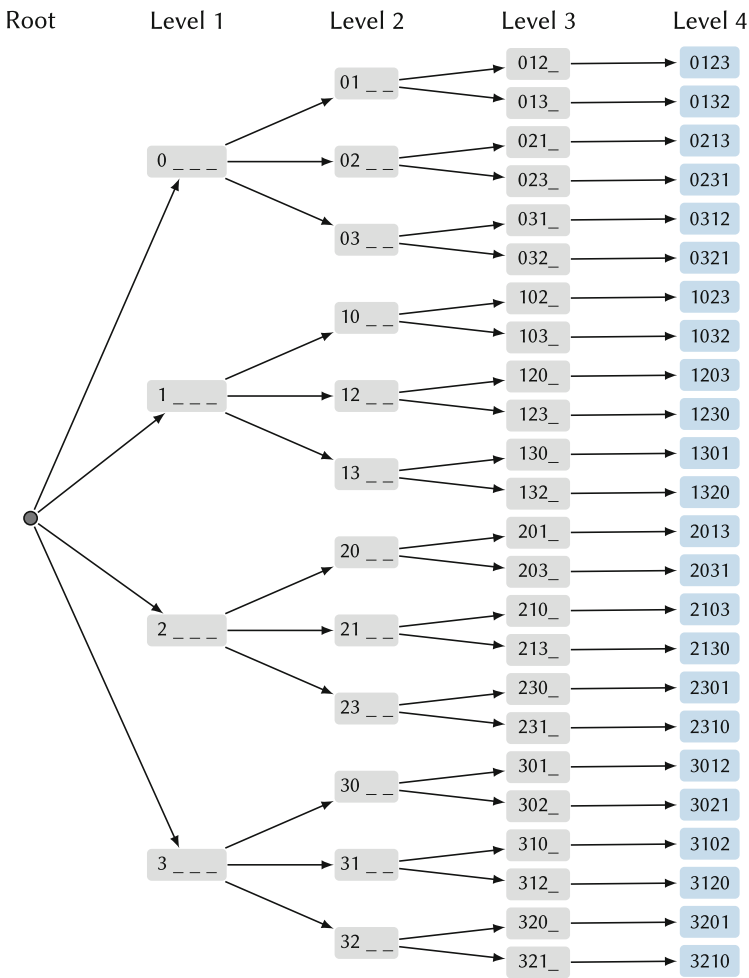


Fig. 5.3 The permutation tree for all permutations of the set of numbers {0, 1, 2, 3}. It should be read starting from the left, which is where the “root” is located. Each of the first four “siblings” (i.e., · · · on the level 1) has the leftmost element fixed. Moving toward the right, more and more elements are already fixed. From level 3 to level 4, only one number is left to choose from, which is why, e.g., from 012, there is only one “descendent”: 0123. For brevity, the numbers are not separated by commas in this tree

Example 5.4 Permutations of atom positions with constraints

Assume the same nine atoms and nine vacancies as in Example 5.3. This time, five particular atoms should fill the first five vacancies in an arbitrary sequence. The remaining four atoms can be placed in the rest of the vacancy positions. How many different configurations exist?

For the first atom, there are five options to place it. For the second, there are four, etc., until the first five atoms will have been placed. Altogether these are $5! = 120$ different arrangements.

For every one of the $5!$ arrangements, there are now another $4!$ ways to arrange the remaining four atoms. Using the multiplication principle, this gives $5! \cdot 4! = 120 \cdot 24 = 2880$ different configurations, which are significantly fewer configurations compared to the case of having no constraints for the first five atoms (which we computed as a total of $9! = 362880$ different combinations).

Python offers a number of ways for creating permutations, and we will use two of them. In both approaches, we seek to obtain the permutations of these three items:

```
In [1]: all_items = ['A', 'B', 'C']
```

The first approach uses three nested `for` loops with the first one iterating through all items of `all_elements`, while the second loop iterates through items that are not contained in `b` and the innermost loop chooses an item if it is not contained in either `a` or `b`:

```
In [2]: perms = []
        for a in all_items:
            for b in all_items:
                if a == b:
                    continue # skip the rest of this loop iteration
                for c in all_items:
                    if c in [a, b]:
                        continue
                    perms.append((a, b, c))
```

```
In [3]: for p in perms:
        print(p)
```

```
Out [3]: ('A', 'B', 'C')
          ('A', 'C', 'B')
          ('B', 'A', 'C')
          ('B', 'C', 'A')
          ('C', 'A', 'B')
          ('C', 'B', 'A')
```

In the second cell, the number of nested loops corresponds to the length of the permutation. Thus, increasing the length of permutations requires changing code and adding more `for` loops. Alternatively, one can use the powerful Python module `itertools` [3], which is part of the Python standard library and provides functionality for, e.g., combinatorics. In order to obtain all permutations of the letters A, B, and C, the code is as follows:

```
In [4]: from itertools import permutations
perms = permutations(['A', 'B', 'C'])
```

Printing them as above would give the same result. Note that `itertools` functions typically return the so-called *iterators*, but neither lists nor arrays. Therefore, you either have to use, e.g., a `for` loop as in the second cell above, in order to use the results of `permutations()`, or you have to convert the result to a list first (e.g., by using `list(permutations(...))`). Further details can be found in the documentation of `itertools`.²

5.1.5 Permutations of n items, k chosen at a time: Generalization of the Multiplication Principle

Assuming we have n different items, out of which we choose k at a time. How many of these permutations can be formed? At the first position, n different items can be placed, at the second and third positions, $n - 1$ and $n - 2$ items, respectively. For the final position, only $n - (k - 1)$ choices are left. Using the multiplication principle, we find the total number of possibilities for placing n items in k positions:

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot (n - (k - 1)) \quad (5.6)$$

A short derivation (cf. exercise 5.4) shows that this term can be summarized as

$$P_{n,k} := \frac{n!}{(n - k)!} \quad (5.7)$$

where P is the number of resulting permutations, n is the number of available items, and k is the number of items in each resulting permutation. Typical examples that use Eq. (5.7) are questions of the following type: if you have nine things, items, or fruits, in how many ways can you arrange them in a group of 4? The answer is $P_{9,4} = 9!/(9 - 4)! = 9 \cdot 8 \cdot 7 \cdot 6 = 3024$.

² <https://docs.python.org/3/library/itertools.html>

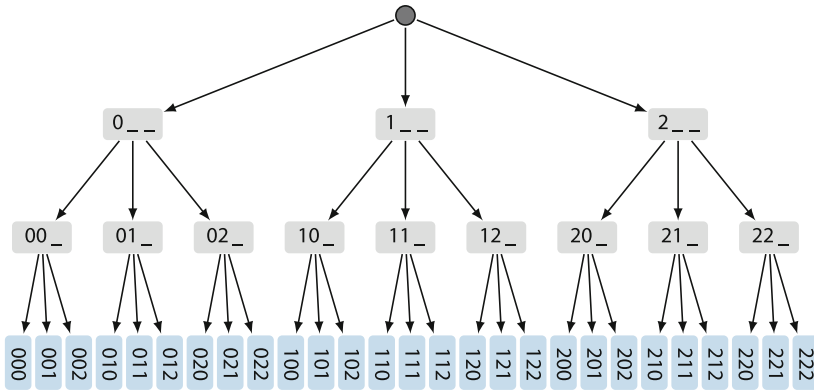


Fig. 5.4 Tree of all ordered samples for the successive drawing of three samples from the set $\{0, 1, 2\}$ with replacement

5.1.6 Permutations with Repetition—Ordered Sets/Samples With Replacement

Creating ordered sets by sampling with replacement, i.e., returning the item that was drawn from a dataset, is yet another special case of the multiplication principle. Let us assume a set with N elements, from which k samples are to be drawn in a way in which the order matters and each drawn element is always returned. For the first sample, we have N possibilities, and for the second one, we have N possibilities again—as for all k samples. Thus, in total there exist

$$P'_{n,k} = \underbrace{N \cdot N \cdot \dots \cdot N}_{k\text{-times}} = N^k \quad (5.8)$$

ways of drawing k items from the set of N elements with repetition.³ Note, that in our notation, we do not differentiate between *with replacement* and *without*. The tree diagram of this case is shown in Fig. 5.4.

Python's `itertools` module is also able to perform this operation using the `product` function. To obtain the $3^3 = 27$ ordered sets for sampling with replacement of the letters A, B, and C and printing them, we could use the following code:

³ An alternative notation for such a product is to use the product operator: $\prod_{i=1}^k N = N^k$

```

In [1]: from itertools import product

In [2]: # parameter `repeat` defines the number of sample drawn from ['A', 'B', 'C']
prod = product(['A', 'B', 'C'], repeat=3)

In [3]: # As `product` returns an iterator we can't just use `print` but have to
# use a for loop to iterate over all elements (or here: all tuples).
i = 1
for p in prod:
    print(i, ": ", p)
    i += 1

Out [3]: 1 : ('A', 'A', 'A')
2 : ('A', 'A', 'B')
3 : ('A', 'A', 'C')
4 : ('A', 'B', 'A')
5 : ('A', 'B', 'B')
...
23 : ('C', 'B', 'B')
24 : ('C', 'B', 'C')
25 : ('C', 'C', 'A')
26 : ('C', 'C', 'B')
27 : ('C', 'C', 'C')

```

5.1.7 Combinations Without Replacement—Unordered Sets/Samples Without Replacement

We now turn to the second big category in combinatorics: *combinations*. As opposed to permutations, in a combination the order of elements does not matter.

The number of combinations that can be obtained from a given set of items is denoted by C (while the number of permutations was denoted by P). More specifically, we are looking for the resulting $C_{n,k}$ combinations for a total number of n available items, placing k items in each of the resulting combinations.

When the order of elements does not matter and each element can occur only once, we have a *combination without repetition*—or just *combination*.

A typical question in this scenario could be as follows: if you have a set of four items, how many different *unordered* pairs of items can you choose? In Fig. 5.5 we show all unordered pairs of the letters A, B, C, and D as a 4×4 matrix. We start by writing down all possible *permutations* and then remove the letters that should not show up in a combination without repetition. Clearly, the repeated elements on the main diagonal have to be excluded because our prerequisite is that each element can occur only once. Additionally, due to symmetry reasons, the sets below the main diagonal are identical to those above the diagonal (remember that we are looking for *unordered* sets). We end up with the set $\{AB, AC, AD, BC, BD, CD\}$ shown in blue in Fig. 5.5.

What is the system used for counting here? If we have a set of N items, then altogether there are N^2 possible combinations of two letters. The main diagonal has N items, all of which need to be excluded. Due to symmetry, we end up with

AA	AB	AC	AD
BA	BB	BC	BD
CA	CB	CC	CD
DA	DB	DC	DD

Fig. 5.5 The matrix shows all pairs of items chosen from the set $\{A, B, C, D\}$. As samples are drawn without replacement, the double letters on the main diagonal are excluded. As we are looking for unordered sets, all elements below the main diagonal can be excluded. Only the six blue elements remain, which are the unordered sets without replacement

$$\frac{N^2 - N}{2} = \frac{N(N - 1)}{2}. \quad (5.9)$$

Alternatively, we can start by filling only the first position—there are N ways of doing this. The second position can then only be filled by one of the remaining $N - 1$ items. As we double counted, we have to divide by two, which then yields the same result as above.

If, instead of pairs, we want to create unordered sets of k items, drawn without replacement from N items, the situation is a bit more complex. However, we can make use of what we already found out about the problem of permutations of n items with k objects chosen simultaneously at a time, cf. Sect. 5.1.5.

1. Consider a particular permutation p (which consists of k items) from the set of all, in total $P_{n,k}$, possible permutations.
2. p is also a valid combination and contained in the set of all, in total $C_{n,k}$, combinations.
3. If we “converted” all permutations into combinations, the particular combination c would be repeated several times—to be precise, it would be repeated $P_{k,k} = k!$ times. We would answer the question: How many permutations of length k can one create from a set of k different elements?
4. If each combination were repeated $k!$ times, the number of permutations would be greater than the number of combinations by the factor $k!$. It follows that

$$C_{n,k} = \frac{P_{n,k}}{k!} = \frac{n!}{k!(n - k)!} \quad (5.10)$$

Turning the task shown in Fig. 5.5 into a small Python example is again straightforward, if you are familiar with the module `itertools`. Luckily, it has a function called `combinations` that fits exactly for our problem:

```
In [1]: from itertools import combinations
```

```
In [2]: for c in combinations('ABCD', 2):
         print(c)
```

```
Out [2]: ('A', 'B')
         ('A', 'C')
         ('A', 'D')
         ('B', 'C')
         ('B', 'D')
         ('C', 'D')
```

```
In [3]: len(list(combinations('ABCD', 2)))
```

```
Out [3]: 6
```

Note that in the last cell, we first had to convert the result of `combinations` to a list in order to show the number of items.

As a short form in the expression in Eq. (5.10), the following definition is used frequently:

Definition 5.5 (Binomial Coefficient). The binomial coefficient is defined as

$$\binom{n}{k} := \frac{n!}{k!(n-k)!} \quad (5.11)$$

It gives the number of combinations, $C_{n,k}$. The combinations consist of k items that are taken (without replacements) from a set of n items. Often this is referred to as “ n choose k .”

The binomial coefficient can easily be used to compute the number of combinations. A number of additional rules and applications for binomial coefficients exist, for which, however, we refer the reader to the vast body of literature on combinatorics.

A Python implementation uses the already introduced functions for factorials, and, e.g., for the example in Fig. 5.5, we could write:

```
In [1]: from math import factorial
```

```
In [2]: n, k = 4, 2
         factorial(n) // ((factorial(n-k)) * factorial(k))
```

```
Out [2]: 6
```

The `//` in the second cell denotes an integer division. The “regular” division would yield a floating point number as a result.

5.1.8 Combinations with Replacement—Unordered Sets/Samples with Replacement

Creating *combinations with replacement* is the second case, in which the order does not matter. In contrast to the previous case, however, whenever an item is drawn, it will be returned, i.e., each element may occur more than once. The number of combinations with repetitions of k objects chosen from a total of n objects is denoted by $C'_{n,k}$. Here, we just state the final result:

$$C'_{n,k} = \frac{(n+k-1)!}{(n-1)! \cdot k!} = \binom{n+k-1}{k} \quad (5.12)$$

An example would be rolling three dices, while the order of them being thrown does not matter (e.g., $\{\square, \square, \square\}$ is identical to $\{\square, \square, \square\}$). This is equivalent to rolling one die three times—or three times drawing a ball from a jar, which contains six balls numbered from 1 to 6, and each drawn ball is returned. Figure 5.6 shows all possible combinations for a dice being thrown three times, which results in altogether $1 \cdot 6 + 2 \cdot 5 + \dots + 5 \cdot 2 + 6 \cdot 1 = 56$ combinations. Using Eq. (5.12) with $k = 3$ samplings out of $n = 6$ numbers, we can calculate the same total number of combinations:

$$C'_{6,3} = \frac{(6+3-1)!}{(6-1)! \cdot 3!} = \frac{8!}{5! \cdot 3!} = \frac{8!}{5! \cdot 3!} = \frac{8 \cdot 7 \cdot 6}{3 \cdot 2 \cdot 1} = 56. \quad (5.13)$$

We conclude this section again with a small Python implementation using the module `itertools` for the task shown in Fig. 5.6:

```
In [1]: from itertools import combinations_with_replacement
```

```
In [2]: comb = combinations_with_replacement([1, 2, 3, 4, 5, 6], 3)
        for c in comb:
            print(c)
```

```
Out [2]: (1, 1, 1)
         (1, 1, 2)
         (1, 1, 3)
         ...
         (5, 5, 6)
         (5, 6, 6)
         (6, 6, 6)
```

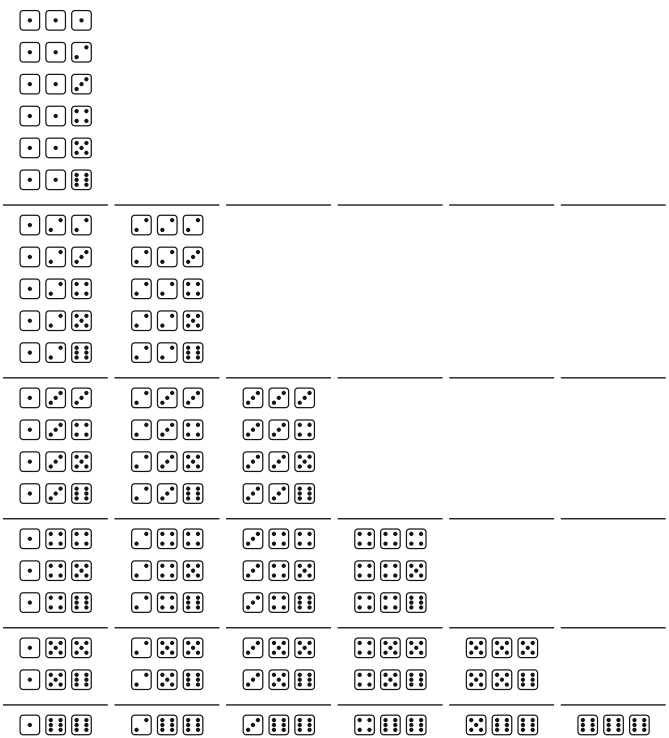


Fig. 5.6 All combinations with replacement of rolling a die three times. In each column, the first dice is fixed; in each section of each column, the second dice value is kept, while the third one is changed. We start at the top left and continue toward the bottom and then switch to the top of the next column. “Duplicates” (e.g., $\{1, 2, 3\} = \{3, 2, 1\}$) are skipped

5.1.9 Summary: Permutations and Combinations

This section introduced *how to count and arrange items* by permutations and combinations. We have seen that there are two types of permutations and two types of combinations, both either with or without replacement.

Things to Remember Permutations and Combinations

★

- Permutations are about the number of different ways items can be arranged
- Combinations are about the number of ways items can be selected
- After “drawing” items from a set ...
 - the item can be returned (called “with replacement” or “with repetitions”) ...
 - or the item can be kept (called “without replacement” or “without repetitions”)

Selection type	order of items matters?	repetitions/ replacements?	number of permutations or combinations	typical question
Permutations	Yes	No	$P_{n,k} = \frac{n!}{(n-k)!}$	1.
	Yes	Yes	$P'_{n,k} = n^k$	2.
Combinations	No	No	$C_{n,k} = \frac{n!}{(n-k)! \cdot k!}$	3.
	No	Yes	$C'_{n,k} = \frac{(n+k-1)!}{(n-1)! \cdot k!}$	4.

The set from which combinations or permutations are created has n Items; the resulting combinations or permutations contain k items. A prime (as in P' or C') indicates that samples are drawn with replacements. Typical questions asked for these four cases are as follows:

1. How many possible sequences are there, if k cards are drawn from a stack containing n unique cards?
2. How many sequences exist, if a card is k times drawn from a stack of n cards and always returned to the stack?
3. How many sets of k cards can you draw from a stack of n cards, if their order does not matter?
4. How many sets of k cards can you draw from a stack of n cards, if the order does not matter and each card is always returned to the stack for the next draw?

5.2 Probabilities

In the previous section in Definition 5.1, we already introduced the notion of *sampling*. Drawing random samples from a set often implies that each element of a set has the same chance to be drawn, but this is not necessarily always the case. Random sampling is one of the most important concepts for investigating probabilities. Probabilities, in turn, can be used for modeling random processes. The underlying uncertainty is then represented by *random variables*.

Relationship Between Probabilities, Statistics, and Machine Learning ?

What is the relation between probabilities, statistics, and machine learning, and why is probability theory important? Probability theory starts with an abstract or mathematical model, e.g., that of a coin toss, and predicts the observations.

In statistics as well as, roughly speaking, in machine learning inference, it is the other way round: we have a collection of observations and then try to figure out, which underlying process or “model” could explain the observations.

Relationship to Machine Learning

Why should you know about probabilities, if you are interested in *machine learning* (ML) in the first place? To start with, a solid understanding of probabilities and related concepts helps to judge the quality and information content of data and allows to make predictions, when purely analytical approaches fail. More concretely, the direct relation to ML is that probabilities and *random variables* help us to quantify uncertainty in the data of the machine learning model and its *predictive power*. And last but not least, a number of ML methods exist that directly make use of estimating uncertainties.

5.2.1 What is Probability?

Probability is a mathematical measure that gives the *likelihood* of a particular event to happen in the future—or not. Thus, it is concerned with *uncertainty*. *Probability theory* is the whole field that deals with probabilities. Questions of interest that can be answered are as follows: (1) How many times do we get “Heads” (H) or “Tails” (T) when flipping a coin? (2) What is the likelihood that a specimen has not failed after a million loading cycles?

The answer to the first question is straightforward and can be figured out analytically for a perfect coin: *on average* (i.e., when we flip the coin many times), we expect it to land the same number of times on the Heads side as on the Tails side. Hence, the probability P for both outcomes is 0.5 or 50% each. Unfortunately, this holds only *on average* and for a large, possibly infinitely large, number of tosses. For only a few coin tosses, the results might suggest completely different probabilities, e.g., when we get five times “Heads” (H) in a row.

The answer to the second question, which is of importance to materials research, requires further details to be specified. But even then, it probably cannot be answered based on mathematical considerations alone and needs a number of well-designed experiments to be performed.

We will now continue with a number of definitions for commonly used notions and formalize the most important concepts around probabilities.

Probability

Here is a definition and explanation of probability in plain words:

Definition 5.6 (Probability). The *probability* of an event represents the likelihood of the event to occur, taking values between 0 and 1. A value of 0 indicates that the event will never occur, while 1 indicates that the event occurs with certainty. The higher the value of the probability, the higher the likelihood of the event to happen.

Often, P is used as variable for the probability of a specific event.

In other words, probability is the fraction of the number of times a specific event occurs. We often denote a probability as a percentage, meaning a probability p of $x\%$ is identical to $p = x/100$.

A slightly different definition is related to our existing and possibly changing knowledge about a situation: Probability is the *degree of belief* about an event. This is a definition that we will use in the context of Bayesian statistics and [ML](#).

Basic Random Experiments

The three best-known and most frequently used examples, for which we can intuitively calculate probabilities, are presented in [Example 5.5](#). Real or virtual experiments for studying random behavior are called *random experiments*.

Example 5.5 Probabilities of Simple Experiments



- The probably simplest example is that of a coin toss: if it is a fair coin and a fair toss, then in half of all cases—at least *on average*—the coin will show Heads (H) or Tails (T), and the corresponding probabilities for either are 0.5 (or 50%).



- The second, equally famous, example is that of rolling a six-sided die. Here, one could ask what the probability was of rolling a three. There are six potential outcomes of rolling the die (1, 2, 3, 4, 5, and 6), all of them occurring with equal likelihood. Then, *on average* every number should occur one sixth of the times, and therefore the probability is $1/6$.



(continued)

- Another typical example is drawing a ball from a jar that contains six blue and four red balls. The probability of drawing a blue ball is $6/10$, while the probability of drawing a red ball is $4/10$ —*on average*.



A more complex type of random experiments, for which it is more difficult to calculate probabilities, is to draw one or more cards from a deck of cards. This is a more complex, because the 52 cards of a standard deck have multiple properties (four different suits, each of them having three “face cards” and ten values). And last but not least, querying your Python random number generator for one or more random numbers is also a random experiment.

Observation

In probability theory, you will often read about *observations*, which means:

Definition 5.7 (Observation). Any information that is recorded, e.g., during such a random experiment, is called an *observation*.

Many of the observations discussed in this chapter consist of *categorical data* that can only take a limited number of values. For example, the result of a coin toss will be either H or T, and the result of drawing a ball from a jar with blue, green, and red balls will be the color of the ball drawn.

Probability and Equal Likelihood

We now introduce an important assumption, which significantly simplifies many formulations: *all n outcomes of a random experiment are equally likely*. With this assumption, we can now easily compute probabilities based on the following definition:

Definition 5.8 (Probability of Equally Likely Outcomes). If a set of outcomes consists of m different, equally likely outcomes, the probability is given by

(continued)

$$P = \frac{\text{number of desired outcomes } m}{\text{total number of possible outcomes } n} . \quad (5.14)$$

The assumption of equal likelihood might be a bad approximation of reality. Hence, it is advisable to choose the investigation such that the violation of the equal likelihood is negligible.

In Eq. (5.14) it can be seen that counting and probabilities are closely related. However, counting the total number of outcomes is not always trivial, which is why in the previous section, we have looked at different types of permutations and combinations.

Here is a simple example: if a fair six-sided die is rolled, and the number 2 is observed, then $m = 1$, $n = 6$. Thus, the probability is $P = 1/6$. Obviously, conducting this experiment only once is not reasonable, as the event of observing a 2 might not even occur at all. However, when the die is rolled many times—whatever “many” exactly means!—one can expect that *on average* 16.6̄% ($= 1/6$) of the time, we will observe a 2.

5.2.2 Two Python Examples

Example 1 Python together with the `numpy` library is also a useful tool to conduct random experiments. For example, we can easily draw ten integer random numbers from the set of numbers $1 \dots 6$ with replacement and with equal likelihood—either using `numpy`’s legacy function `random.randint()` or with the more recent approach by creating a random number generator and then calling its `integers()` function; both approaches return random integer numbers from within a given range:⁴

⁴ When dealing with random numbers in a computer program, a few additional points need to be considered: (i) there are no *real* random numbers, only “pseudo” random numbers, which will repeat themselves after some time (actually: after a very long time). For practical purposes, this is good enough. (ii) “Seeding,” i.e., initializing of the random number generator, determines whether for each run of a program the same sequence of random numbers occurs or not. More details are given in the `numpy` documentation at <https://numpy.org> and will also be given in the text when needed.

```
In [1]: import numpy as np

# create and initialize random number generator
rng = np.random.default_rng()

In [2]: N = 10 # number of samples

# create N random integer numbers in the range of [1,...,6]
values = rng.integers(1, 6, endpoint=True, size=N)
print("values: ", values)

Out [2]: values: [1 1 1 3 1 2 6 5 3 3]
```

This code creates $N = 10$ random numbers for rolling a die $N = 10$ times. We observe that the resulting numbers do not occur with the same frequency, e.g., the number 1 was thrown four times, while the number 4 did not show at all. Is the random number generator defective? We cannot tell, as the numbers were drawn by chance, and ten samples might not be sufficient to determine any “imbalance.”

Example 2 To demonstrate how the number of repetitions of an experiment influences the accuracy compared to the behavior that is expected *on average*, we perform another numerical experiment. We roll a fair die multiple times and record the change of the average value for every additional throw as function of the number of rolls (note that the current average value is the sum of all so far recorded values divided by the current number of rolls). The first step for the implementation is again to import the relevant packages, to initialize the random number generator, and to get 50 samples:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(22524)
n_samples = 50
values = rng.integers(1, 6, endpoint=True, size=n_samples)
```

Note that this time we initialized the random number generator with a specific (but arbitrary) seed value of 22524. This has the effect that the sequence of drawn random numbers will always be the same. Next, we use `cumsum` to create an array with the cumulative sum of `values`, i.e., the element i is the sum of all elements of `values` up to position i :

```
In [2]: sample = np.arange(1, n_samples + 1)
averages = values.cumsum() / sample
```

The function `np.arange` creates all integer numbers starting from 1 up to and excluding `n_samples+1` (meaning `[1, 2, ..., n_samples]`). Exercise 5.7 will show how the cumulative sum is computed in more detail. Finally, we are ready to plot our 1D arrays:

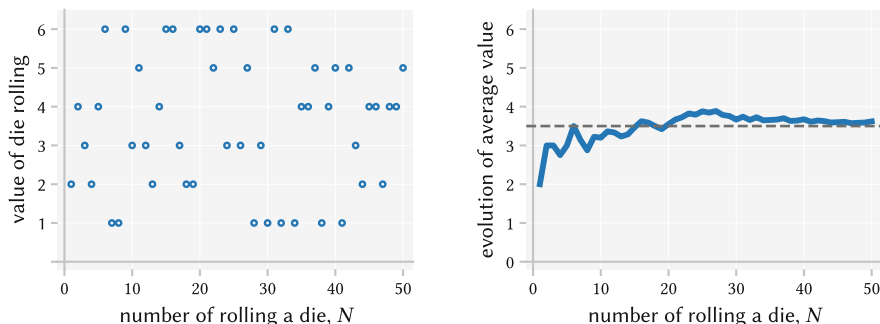


Fig. 5.7 Rolling a fair die 100 times. Left: observed values as a function of the number of die rolling. Right: average value of all die rolls that occurred during the first N rolls. The dashed line shows the expected average value for a fair die

```
In [3]: fig, ax = plt.subplots(ncols=2, figsize=(9, 3))

ax[0].plot(sample, values, '.')
ax[0].set(xlabel='N', ylabel='resulting value', ylim=(0.8, 6.2))

ax[1].plot(sample, averages, '-')
ax[1].axhline(3.5, linestyle='--', c='0.5', lw=1.5)
ax[1].set(xlabel='N', ylabel='average value', ylim=(0.8, 6.2))
```

This results in a figure similar to Fig. 5.7 (add `plt.show()` at the end if you are not using an interactive Jupyter notebook). Note that the random numbers drawn might be different for different Python environments.

The left panel of Fig. 5.7 shows the values from 1 to 6 during rolling a die 50 times. The right panel shows how the average changes accordingly. Initially, the average value was quite low as there were mainly 1s and 2s. Already at $N = 30$, the computed average value is getting closer to the expected average value of $(1 + \dots + 6)/6 = 3.5$ with an error less than 10%. This behavior is typical of random experiments demonstrating that these experiments are powerful for predicting behavior of large ensembles of samplings only. They are nearly useless for predicting the behavior of an individual event.

5.2.3 Outcomes, Events, and the Sample Space

To analyze random experiments, two concepts are required: the concept of the random variable and, associated with it, the *probability function*, which measures the probability of a certain result. These two concepts will be introduced in Sects. 6.1 and 6.2. But first, we will introduce and define the terms *outcome*, *sample space*, and *event* based on the simple random experiment of tossing a coin, in this case four times, while recording the results in the order of occurrence.

We have already used the notion of outcome in a colloquial manner, which is already fairly close to the definition below:

Definition 5.9 (Outcome). *Outcome* denotes a possible result of a random experiment or trial. They are the mutually exclusive “elementary” events of an experiment that cannot be further broken down into a subset of events.

Each single coin toss has two outcomes: Heads (H) or Tails (T). In our case, though, the experiment consists of altogether four coin tosses (it is quite important to be aware of the exact nature of the experiment!). Therefore, the outcome consists of a series of four items, each of which being either H or T. For example, THHH would be an outcome.

Using this example and considering all $P'_{n,k}$ permutations of H and T that result in valid outcomes brings us to the sample space:

Definition 5.10 (Sample Space). The set of all possible outcomes represent the *sample space*. The sample space is denoted by the variable Ω .

For example, for four times tossing a coin, the sample space is the permutation of the four {H, T},

$$\Omega = \{HHHH, HHHT, HHTH, HHTT, \dots, TTTT\} \quad (5.15)$$

The total number of elements in Ω is, thus, $P'_{2,4} = 2^4 = 16$.

To find the sample space in general, a tree diagram can again be helpful to systematically construct all possible outcomes. In case of a very large or even infinitely large sample space, the *statement* or *rule method* can be useful. For example,

$$\Omega = \{x | x^2 > 4\} \quad (5.16)$$

denotes the set of all values x that are larger than +2 or smaller than -2 (note that this example is unrelated to the coin flipping and does not involve Heads or Tails). Discrete values, e.g., the values from throwing a six-sided die, can also be used, e.g., $\Omega = \{n | n \geq 4\}$.

The definition of the sample space allows us to also define the meaning of an *event*:

Definition 5.11 (Event). An *event* is a set of outcomes. We denote events with capital letters, and each of the events is a subset of the sample space, e.g., $A \subset \Omega$.

For example, in the case of our random (coin toss) experiment, the event A could be “having at least two Heads”:

$$A = \{\text{HHTT}, \text{HTHT}, \text{HTTH}, \text{HHTH}, \text{HHHT}, \text{HTHH}, \text{HHHH}\} \subset \Omega. \quad (5.17)$$

As a consequence, each experiment is always associated with the following two events:

- the event A of all possible outcomes, i.e., the whole sample space $A = \Omega$
- the empty set $A = \emptyset$ representing no possible outcome and, therefore, called the *impossible event*

Note the difference between *outcome* and *event*, which might be somewhat counter-intuitive: while the outcome of a random experiment is a single and the most basic “entity,” an event corresponds to possibly several outcomes!

5.2.4 Calculating with Events and Probabilities

A few special cases help the calculation of probabilities, such as the situation that two events do not have any common outcome.

Definition 5.12 (Mutually exclusive). Two or more events A_i are said to be *mutually exclusive* if they do not share any common outcome, i.e., *either* A_1 *or* A_2 *or* ... can occur but not A_1 and A_2 and ... together.

This can also be written more formally as

$$A_i \cap A_j = \emptyset \quad \text{for all } i \neq j. \quad (5.18)$$

The probability that *any* of mutually exclusive events occurs is the sum of all individual probabilities,

$$P(A_1 \cup A_2 \cup A_3 \cup \dots) = P(A_1) + P(A_2) + P(A_3) + \dots. \quad (5.19)$$

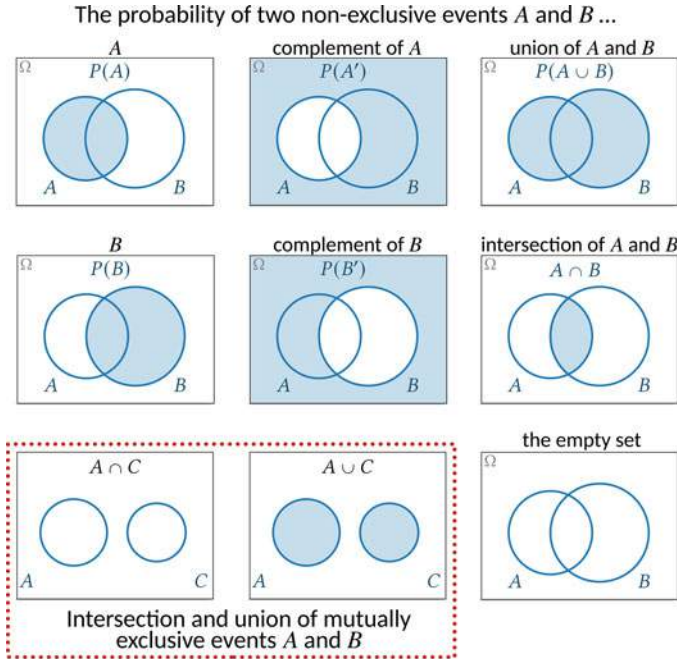


Fig. 5.8 Visualization of different events together with their corresponding probabilities using Venn diagrams

Often events are mutually exclusive, e.g., if the outcome is considered the event. For example, rolling a die gives one out of six mutually exclusive events if a single die roll is the considered experiment. An example for *non-exclusive* events during a die roll is the event of all odd numbers, $A = \{1, 3, 5\}$ and the event of all numbers greater than 2, $B = \{3, 4, 5, 6\}$. The outcomes “throwing a 3” and “throwing a 5” are contained in both events, $A \cap B = \{3, 5\} \neq \emptyset$. Figure 5.8 visualizes these relations in terms of Venn diagrams.

The probability of a union of non-exclusive events can be calculated as:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) . \quad (5.20)$$

Why is this important or what is the relevance? We would now ask what the probability is of *either* an odd number (event A) *or* a number larger than 2 (event B) to occur upon rolling a die. As we already know, the events are

$$A = \{1, 3, 5\} \quad \text{and} \quad B = \{3, 4, 5, 6\} .$$

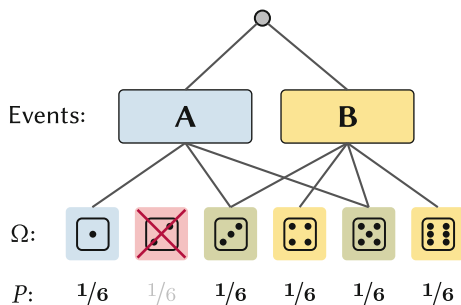


Fig. 5.9 Visualization of the two “overlapping” events where the outcomes of each event are indicated by connecting lines. The outcomes $\boxed{2}$ and $\boxed{3}$ are contained in both events. Out of six possible outcomes of the single-die-throw (the sample space), five are “covered” by the two events A and B (see text for further explanations)

The probabilities, thus, are

$$P(A) = \frac{3}{6} = 0.5 \quad \text{and} \quad P(B) = \frac{4}{6} \approx 0.667. \quad (5.21)$$

The probability of getting an event either from A or from B is obviously not just the sum of the two individual probabilities—this would result in $7/6 \approx 1.667$, but probabilities larger than 1 are not possible. The problem is that we are double counting the events $A \cap B$, which is why the probability needs to be reduced by $P(A \cap B)$:

$$P(A \cup B) = \frac{3}{6} + \frac{4}{6} - \frac{2}{6} = \frac{5}{6} \approx 0.833. \quad (5.22)$$

Figure 5.9 visualizes this situation, making it easy to simply count outcomes and to ignore double counting. Note that this figure (which resembles a tree diagram) does not reflect the properties of mathematical sets, but is rather a convenient way of sorting and counting items. In terms of Venn diagrams reflecting the proper mathematical representation, we would draw two overlapping sets, e.g., as circles, that contain the respective events and that show $\boxed{2}$ and $\boxed{3}$ in their intersection.

As a final subtopic of this section, we consider *independent events* and their probabilities. We already have a good intuitive understanding of this concept and have used it, for example, when we investigated two coin tosses. We could either have tossed a single coin two times or we could have used two coins and tossed them at the same time just once. The results in terms of the probability of specific outcomes are always the same (at least as long as the coins are identical). Generally, this can be stated as follows:

Definition 5.13 (Independent Events). Two or more random events A_i are said to be *pairwise independent* if for their probabilities it is

$$P(A_i \cap A_j) = P(A_i) P(A_j) \quad \text{for } i \neq j. \quad (5.23)$$

The symbol “ $\cap$ ” denotes the intersection of two sets giving the elements that are contained in both sets. In general, the events $A_1, \dots, A_N$ are *independent* of each other, only if, for all events, it is

$$P(A_1 \cap \dots \cap A_N) = \prod_{i=1}^N P(A_i). \quad (5.24)$$

Here, we used the product operator, which is defined as $\prod_{i=1}^N x_i := x_1 \times \dots \times x_N$.

Further down, after the introduction of *conditional joint probabilities*, it will become clear why this is indeed equivalent to the independence of events. Generally speaking, this definition implies that the occurrence of an event does not change the probability of the occurrence of any other event. In Sect. 6.5 we will denote the resulting probability of independent events as *joint probability*.

As an example, imagine that you roll a six-sided die and flip a coin. We could define the events as $A_1 = \{4, 6\}$ for the die and $A_2 = \{H\}$ as the event of interest for the coin. It is $P(\{(4, H), (6, H)\}) = 2/12$, while for the individual probabilities, it is $P(\{4, 6\}) = 2/6$, and $P(\{H\}) = 1/2$. The product of the two individual probabilities is again $P(\{4, 6\}) \times P(\{H\}) = 2/6 \times 1/2 = 2/12$.

5.2.5 The KOLMOGOROV Axioms

Much of what we have just introduced can be condensed into or rather derived from the *Kolmogorov Axioms*⁵ that were introduced in 1933 by the Russian mathematician ANDREY KOLMOGOROV (see [2] for an English translation). The following three axioms form the basis of probability theory as we know it today. They help determine the types of mathematical operations that can be performed on probabilities:

⁵ An *axiom* is a statement that cannot be proven and is assumed to be true. Axioms act as the point of departure for constructing further theoretical frameworks.

1st Kolmogorov Axiom: The probability P of any event is a non-negative real number ($P \in \mathbb{R}^{0+}$), also simply written as

$$P \geq 0 . \quad (5.25)$$

This is what we are used to in our intuitive understanding of the meaning of probability. The higher the number, the higher the likelihood. However, there are mathematical concepts and spaces, for which this does not hold automatically, thus underscoring the importance of the axiom.

2nd Kolmogorov Axiom: The probability that at least one of the outcomes in the entire sample space Ω will occur is 1. This is also called the *assumption of unitary* and can be summarized as

$$P(\Omega) = 1 . \quad (5.26)$$

From this axiom we understand that if a probability value is larger than 1 or smaller than 0, it actually cannot represent a probability—at least not unless we move to very strange mathematical theories and spaces. Everything that happens in the “real world” has a likelihood to happen that is between (and including) absolute certainty and no likelihood at all.

3rd Axiom: for any sequence of mutually exclusive events E_i , it holds that the probability of the union of all events is equal to the sum of all individual probabilities,

$$P\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} P(E_i) . \quad (5.27)$$

The last axiom tells us that we can add probabilities (which also holds for both finite and infinite sets). It also tells us that for mutually exclusive events E_i and E_j , it is

$$P(E_i \cup E_j) = P(E_i) + P(E_j) . \quad (5.28)$$

Starting from these three axioms, many more important probabilistic theorems and relations can be derived and are covered in most text books on probability theory and statistics; see, e.g., [1].

5.3 Conditional Probabilities, Product rule, and Bayes' theorem

We conclude this chapter with a short overview of conditional probabilities and some other formulations that often occur in the context of Bayes' theorem.

As a prerequisite for Bayes' theorem, we briefly summarize some important notation and conventions:

- If the probability of an event A does not depend on the outcome of any other random variable, then we denote the probability by $P(A)$ and call it the *marginal probability*.
- The *joint probability* that two events A and B take place simultaneously is written as $P(A \cap B)$
- The *conditional probability* of an event A , given that another event B takes place, is written as $P(A | B)$. It can be related to the joint probability through

$$P(A \cap B) = P(A | B) \cdot P(B) , \quad (5.29)$$

which is also known as the *product rule*. This gives us a way to compute the conditional probability

$$P(A | B) = \frac{P(A \cap B)}{P(B)} . \quad (5.30)$$

Words of advice



Note that, while the joint probability is symmetric, $P(A \cap B) = P(B \cap A)$, the same does not hold for the conditional probability and, in general, it is $P(A | B) \neq P(B | A)$.

With these notions and definitions, we can now state Bayes' theorem (or, as it is also sometimes called, Bayes' rule). It is defined as:

Definition 5.14 (Bayes' Theorem). Given are two events, A and B . Then, for their conditional probability, it holds:

(continued)

$$P(B | A) = P(A | B) \cdot \frac{P(B)}{P(A)} . \quad (5.31)$$

Bayes' theorem helps to compute the probability that events belong to a certain class, given some prior knowledge. Often, for the terms occurring in the above definitions, specific names exist. For example, $P(A)$ is the *prior probability*, while $P(A | B)$ is the *posterior probability*. Furthermore, $P(B | A)$ is called *likelihood* and, last but not least, $P(B)$ is the *evidence*. All of these are terms that we will again encounter in some probabilistic ML models.

5.4 Summary

In this section, we introduced the concept of probabilities and random experiments. After systematically defining outcomes, sample space, and what an event is, we could then start to calculate with events and probabilities. There, Venn diagrams helped us to visualize some of the set operations. An important aspect (also for probabilistic machine learning methods) was the concept of independent events and the corresponding probability. Last but not least, we introduced the notions of conditional and joint probabilities, which are a prerequisite for Bayes' theorem.

Things to Remember (Probabilities)



- The probability P of any event is a number $0 \leq P \leq 1$
- The probability of any mutually exclusive events A and B is $P(A \cup B) = P(A) + P(B)$.
- For two events A and B , the following holds
 - A and B are independent $\Leftrightarrow P(A) \cdot P(B) = P(A \cap B)$
 - $P(A \cap B)$ is the joint probability that both events take place simultaneously.
 - conditional probability of event A given that event B takes place: $P(A | B)$
 - product rule: $P(A \cap B) = P(A | B) \cdot P(B)$
 - Bayes' theorem: $P(B | A) = P(A | B) \cdot \frac{P(B)}{P(A)}$

5.5 Exercises

5.1. Plot the first eight factorials together with their approximation by the Stirling formula. Which is the largest integer that your computer can still compute the factorial for? How do you explain this?

5.2. Write a small Python function that computes the factorial of a number based on the “recursive” formula Eq. (5.2).

5.3. Formulations of the multiplication principle: use Eq. (5.6) to derive Eq. (5.7).

5.4. Draw the tree diagram for Fig. 5.5.

5.5. The lattice structure of Cu_3Au is $L1_2$. The unit cell consists of four atoms, three of which are Cu and one Au. The lattice is a face-centered cubic type, with Cu atoms on each face of the lattice and Au atoms at the corners. At higher temperatures, often the structure can get disordered, and the individual species are no longer bound to their ordered positions.

- If we consider a total of four lattice sites, how many possible orderings of the atoms are possible? Consider that all three Cu atoms are identical.
- If all the possible orderings have equal energy, what is the probability of each of the configurations?

5.6. A jar contains 15 marbles of the same size. Ten of them are blue and five are yellow. Imagine that you are blind-folded and take one marble with each hand. Then the colors are recorded.

- What is the name of this random procedure?
- Write down the sample space.
- How can you write the following event in terms of the sample space: “the marble in the left hand is yellow”?

5.7. Write down the mathematical function for a *cumulative sum* of the first N positive integer numbers. Write a Python function that computes the cumulative sum of all elements of a given array. Test and compare your implementation with numpy’s function `numpy.cumsum()`.

5.8. Which of the following options denote feasible sample spaces:

- $\Omega = \{-1, 0, 1, 2, 3\}$
- $\Omega = \{1, \dots, 10\}$
- $\Omega = \{\text{☺}, \text{☹}, \text{☹}, \text{☹}\}$

4. $\Omega = \{x \mid x > -1 \text{ for all } x \in \mathbb{R}\}$
5. $\Omega = \{\}$

5.9. The sample space (cf. Sect. 6.1) can be systematically created using a tree diagram. Assume a random experiment that consists of two steps: (1) flip a coin and (2) if H was drawn, flip the coin a second time—but if T was drawn, throw a six-sided die. Create a tree diagram of all possible outcomes and write down the elements contained in the sample space S .

5.10. Given are two types of events. Event A is the set of numbers below 50, and even B are all odd numbers below 100. We randomly draw 20 samples with returning them from the set of all numbers from 1 to 100. Compute the likelihood of event A , event B , and that both event A and B occur.

5.11. Assume the same events as in Exercise 5.10. Write a Python program that draws n samples with returning them from the set of all numbers in between 1 and N where n and N are parameter. Use the Python code with $n = 20$ and $N = 100$ and (based on the drawn random numbers) compute the likelihood of event A , event B , and that both event A and B occur. Compare your results with the theoretical values from Exercise 5.10 and discuss them.

References

1. M. Bonamente. *Statistics and Analysis of Scientific Data*. Springer New York, 2017. DOI <https://doi.org/10.1007/978-1-4939-6572-4>.
2. A. Kolmogorov, A. Bharucha-Reid, and N. Morrison. *Foundations of the Theory of Probability: Second English Edition*. Dover Books on Mathematics. Dover Publications, 2018. ISBN 9780486821597.
3. G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

Random Variables and Probability Functions

6

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.

John von Neumann (1903–1857)

Hungarian-American mathematician and computer scientist

6.1 Random Variables

While the idea of the sample space, where we simply write down *all* possibilities, might seem compelling, often this is not practical or contains many more (or too many) details than actually needed. Instead, we use a *random variable*, which represents the data in a compact numerical format (by definition the data has to be numerical if it is to be used with a random variable). In the literature, *random variable* is often abbreviated as **RV**. However, in this text, we write it out in order to avoid the use of too many abbreviations.

Probability functions can be categorized depending on the type of the random variable, i.e., whether it results in discrete or continuous data. Example 6.1 presents the categorization based on a few examples.

Example 6.1 (Classification of Random Variables and Distributions) ✖

- Typical examples for **discrete random variables** involve *counting* items, e.g., the number of cars passing by within a half hour. Upon a specific event

(continued)

only (a car driving by), the value of the random variable is changed, and there are “gaps” between all events.

A materials science example is counting the number of defects observed with a microscope on the surface of a specimen. Each sample results in a value, and the defect numbers of all samples result in the discrete distribution.



- We encounter **continuous random variables** whenever a measurement or observation takes place continuously, i.e., without “gaps” between individual measurements. Measuring the temperature w.r.t. time is a continuous process (at least when observing a thermometer or neglecting the sampling rate of the measurement device).

We often want to infer the most likely underlying continuous distribution of a dataset because the shape of a distribution might give us important information about the underlying physics. For example, the type of grain size distribution (rather log-normal distribution or not) can give information regarding the type of grain growth.



- **Categorical random variables** represent qualitative data of different classes or categories without any particular order. An example is drawing colored marbles from a jar and recording the colors. Although we *could* assign numbers to each color, e.g., red $\mapsto$ 1, blue $\mapsto$ 2, green $\mapsto$ 3, there is no point in asking if 2 (green) is larger than 1 (blue)!

Materials science examples include the material type or chemical element or if a specimen was fractured or not.



6.1.1 Discrete Random Variables

A random variable X reflects the relevant aspects of an experiment that we are interested in. For example, when X represents the number of Tails thrown, X takes the integer values 0 for HHHH, 1 for HHHT, or HHTH, etc., and ranges over all numbers from the set $\{0, 1, 2, 3, 4\}$. The concept of *random variables* is defined for a discrete sample space in Definition 6.1.

Definition 6.1 (Discrete Random Variable). A discrete random variable is a *function* that maps each individual item from the sample space to a number. Formally, this is written as

$$X : \Omega \rightarrow \mathbb{Z} \quad (6.1)$$

where X is the random variable and $\mathbb{Z}$ represents the set of all integer numbers $\{\dots, -2, -1, 0, 1, 2, \dots\}$. The “destination” of X is called *image* of the random variable.

The purpose of a *discrete* random variable is to map a discrete, possibly non-mathematical, description of a particular outcome to a mathematical number, such that from then on mathematical calculus can be used. Note that even though X is called *random variable*, it is not a classical variable, but rather a function or “lookup table.” The naming often creates some confusion, but it is, nonetheless, the commonly used notion. As opposed to continuous random variables, a discrete variable can take *countable* values in a finite period of time, even though there may be an infinite number of them (as, for example, in the set of integers, $\mathbb{Z}$).

Another often used and probably slightly confusing notation is the following: while the random variable is denoted by an uppercase letter X , each of its values is denoted by a lowercase x , e.g., x_1 . More details will be given below.

The purpose of the random variable is therefore twofold: (i) To condense all outcomes of an event into a single variable (ii) To replace, e.g., the textual labels such as “Heads” or HTHH by numerical values—to be more precise: in that case by integer numbers. Only then are we able to perform calculations and compute probabilities as demonstrated in Example 6.2 for the case of tossing a coin four times.

Words of advice

In machine learning we often encounter categorical variables that represent a finite number of different classes. As categories are not necessarily numbers, they might not have a logical order. In particular, there is no unique way of mapping the categories to integer numbers. As categorical data cannot be ordered, there is also no unique “cumulative distribution” (cumulative distribution functions will be introduced in Sect. 6.4.2).

Keeping this limitation in mind, our definition of the discrete random variable in Definition 6.1 implicitly covers categorical data as well.

6.1.2 Continuous Random Variables

So far, the outcomes of a random experiment were always of categorical type, and the number of different outcomes was usually countable. *Continuous random variables*, on the other hand, can only take on continuous values

Definition 6.2 (Continuous Random Variable). A continuous random variable is a *function* that maps each of the infinitely many elements of the sample space to a real number. Formally, this is written as

$$X : \Omega \rightarrow \mathbb{R} \quad (6.2)$$

where X is the random variable and $\mathbb{R}$ denotes the real numbers. The “destination” of X is called *image* of the random variable.

The lifetime of a specimen is an example for a continuous random variable. For example, the lifetime could have arbitrary values from within the range of 0 to 2 years (zero because some specimens are defective right from the beginning and a maximum value irrespective of the kind of usage assuming, e.g., that after 2 years some parts simply fail or are worn out).

6.1.3 Assigning Probabilities to Discrete Random Variables

Assigning probabilities to events is straightforward for simple experiments. For example, for the case of 4 times tossing a coin, each of the 16 possible outcomes is equally likely and has a probability of $1/16$. Computing the probability of events follows Eq. (5.14) and, therefore, requires counting the number of outcomes that

belong to a particular event. To denote the probability of event number 3, we would write $P(x_3)$, while the collection of all x_i represents the random variable X . See Example 6.2 for a demonstration.

Example 6.2 (Random Variables and Event Probabilities (Part 1))



Consider the random experiment of four times tossing a coin. The according sample space Ω consists of $N = 16$ elements. What is the random variable and how can we compute probabilities?

The **random variable** X could be the number of Heads occurring in the experiment. Each “entry” or measurement of this variable (or mapping) is denoted by a lowercase variable name, e.g., no Heads is represented by $x_1 = 0$, one Head by $x_2 = 1$, etc., up to $x_5 = 4$. The subscript i in x_i does *not* stand for the number of Heads; the value of x_i does. For **assigning probabilities to events**, we count how many outcomes contribute to each entry of the random variable:

Event	$x_i = ?$	Outcomes	Number of contributing outcomes n_i
0 Heads	$x_1 = 0$	TTTT	1
1 Heads	$x_2 = 1$	TTTH, TTHT, THTT, HTTT	4
2 Heads	$x_3 = 2$	HHTT, HTHT, HTTH, THHT, THTH, TTTH	6
3 Heads	$x_4 = 3$	HHHT, HHHT, HTHH, THHH	4
4 Heads	$x_5 = 4$	HHHH	1

As each outcome has an equal likelihood of $1/16$, the probabilities of the contributing outcomes for X are $P(x_1) = n_1/N = 1/16$, $P(x_2) = 1/4$, $P(x_3) = 3/8$, $P(x_4) = 1/4$, and $P(x_5) = 1/16$.
[This example is continued in Example 6.3]

6.1.4 Assigning Probabilities to Continuous Random Variables

In the case of discrete events, we could simply write down probabilities and assign them to probabilities, e.g., in the form of a table as in Example 6.2. However, in the case of continuous random variables, this is not feasible anymore because then the random variable may assume infinitely many values. As an example, consider the measurement of a temperature T : even if the temperature varies only within a small range of $[T - \Delta T, T + \Delta T]$, this interval still contains infinitely many real numbers, and thus, we cannot write all of them down in a table. Mapping the values of the

random variable to probabilities can be done by another function, the *probability function*. Such a function can be defined for both discrete and continuous variables and will be introduced in what follows.

6.2 Introduction of Probability Functions

The previous sections introduced the two different types of random variables, and we have also seen that random variables are *somehow* linked to probabilities (compare Example 6.2). We will now formalize this relation.

6.2.1 Why Do We Need Probability Functions?

While linking random variables to probabilities is straightforward for the case of discrete random variables, the case of continuous random variables is more complex because the *image of the random variable* (cf. Definition 6.1) is not countable anymore. Thus, it is not possible to write down all values of the random variable together with the respective probabilities. The remedy comes in the form of another function, the *probability function*. A visual representation of how these concepts are linked is shown in Fig. 6.1.

In this figure, the *probability mass function* maps a discrete random variable to the probability value, while for the case of a continuous random variable, the mapping is achieved by the *probability density function*. Both functions will be introduced in more detail below. We start with a general definition:

Definition 6.3 (Probability Function). A *probability function* is a mathematical function that assigns a specific probability value to any outcome of a random experiment.

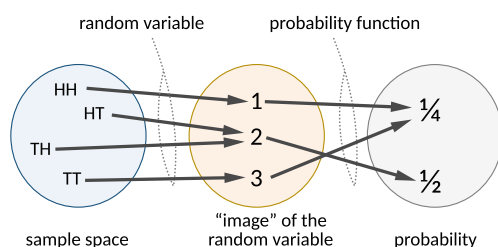


Fig. 6.1 Visualization of how discrete random variables are linked to the sample space, on the one hand, and to probabilities, on the other hand, using the example of two coin tosses. The four discrete events from the sample space are mapped to integer numbers, each of which is associated with a probability. The set in the middle is called the “image of the random variable”

Hence, probability functions are closely related to the concept of random variables: while the random variable assigns a value to each event or outcome, the probability function gives us the probability value of the event represented by the number.

Words of advice



Probability functions are also called *probability distribution functions* or in short *probability distributions*. Note that a probability distribution function is different from the probability *density* function, which is often abbreviated as PDF and is a special type of a probability distribution. However, this distinction is not always followed consistently in the literature.

Probability functions can be categorized depending on the type of the random variable, i.e., whether it is a qualitative or quantitative data. A second way of classification is based on the number of random variables. The knowledge about these two categories will help us later to choose the appropriate data analysis or *machine learning* (ML) method.

6.2.2 Classification of Distribution Types

The First Type of Classification

This classification is with regard to the type of random variable. Already in Sect. 6.1, we differentiated between discrete and continuous random variables. We will now extend the classifications to probability functions. The following cases are to be differentiated:

- Discrete variables take only a countable number of particular values. A discrete variable is always numeric, and therefore, their values can be sorted, e.g., from the smallest to the largest number. A probability function with discrete variables is called a *probability mass function* (PMF) and will be introduced in more detail in Sect. 6.3.
- Continuous variables can take an infinite number of values. The corresponding probability function that uses continuous variables is called a *probability density function* (PDF) and will be introduced in more detail in Sect. 6.4.
- Categorical variables are a third type that has relevance for ML. They represent a finite number of different classes. As categories are not necessarily numbers, they might not have a logical order. In particular, there is no unique way of mapping the categories to integer numbers. A probability function with categorical variables is called a *categorical distribution* or a *generalized Bernoulli distribution*. As such, it is also a kind of discrete distribution.

The difference between discrete and categorical variables is not always pointed out clearly in the ML literature. It is often assumed that simply numbering the different

categories is sufficient to turn categorical variables into discrete variables. While for many ML classification problems this works well, there is a fundamental difference, as already highlighted in the previous chapter: as categorical data cannot be ordered, there is also no unique *cumulative distribution* (cf. Sect. 6.4.2).

A Second Type of Classification

An entirely different type of categorization relates to the *number of random variables*:

- Univariate distributions are the simplest type of probability functions that contain only one random variable. Univariate distributions are often encountered and serve as the fundamental “building blocks” for distributions with more variables.
- Bivariate distributions are distributions that are based on two random variables. Many real-life problems are represented as bivariate distributions because they can still be easily visualized.
- Multivariate distributions are the logical extension of the univariate distribution toward probability distributions with more than one variables and, therefore, are the most general case.

In what follows, we will focus on the first type of classifications and introduce the respective distribution types.

6.3 Discrete Probability Distributions

When the random variable(s) take only discrete values, the discrete probability functions are fully described by simply stating the corresponding probability.

6.3.1 Probability Mass Functions

Discrete probability functions result in the corresponding probability for discrete values. Such a probability for each possible outcome x_i is usually called *relative frequency* $P(X = x_i)$. This function is then called **PMF**, $P(X)$, which performs the following mappings:

$$x_1 \mapsto p_1, \dots, x_n \mapsto p_n . \quad (6.3)$$

Here, the x_i are particular values of the random variable X (e.g., different measurements), and the p_i are the corresponding probabilities. For practical purposes, this mapping can be represented in form of a table, which contains the summarized recordings of a random experiment. The recordings consist of the absolute frequencies (i.e., the number of times) with which each specific event occurs. Such a table is called *frequency distribution table*. An example is shown for a coin tossing experiment in Fig. 6.2. This table is a complete representation of the probability

Result x_i	Absolute Frequency n_i
Heads	4
Tails	3

Fig. 6.2 Frequency distribution table for seven rounds of a coin toss. Recorded were the number of times the event Heads or Tails occurred

function. Additionally, from such a table, the information of the probability as the *relative* frequency $P(X = x)$ can be derived. It is given by

$$P(X = x_i) = \frac{n_i}{N} \quad (6.4)$$

with N as the total number of samples and n_i the number of times that i occurs. A table that shows the probabilities for all events x_i is called *probability table*. It is a purely “data-based” method to represent random experiments and does not say anything about possible underlying functional relations.

An important property of the probability P is that the sum of over all events must equal 1,

$$\sum_i P(X = x_i) = 1. \quad (6.5)$$

This also serves as a good test if a dataset is in a consistent state.

Example 6.3 (Random Variables and Event Probabilities (Part 2)) ✂

[This is the continuation of Example 6.2] Summing up all individual probabilities, we get the probability of 1:

$$\sum_{i=1}^5 P(x_i) = \frac{1}{16} + \frac{1}{4} + \frac{3}{8} + \frac{1}{4} + \frac{1}{16} = 1, \quad (6.6)$$

which means that our list of events indeed covers all possible cases. Two of the many possible visualization types of a random experiment with discrete events are shown in Fig. 6.3. The plots show the results of conducting the above experiment of coin flipping 500 times. The left panel shows the number of times each item of the sample space was tossed. The dotted horizontal line indicates the ideal “theoretical” value—each of the 16 outcomes should have the same value of $500/2^4$. The panel on the right shows how often each of the five events has occurred. Small deviations from the theoretical values can be seen. It is expected that these deviations would vanish as the sample size approaches infinity.

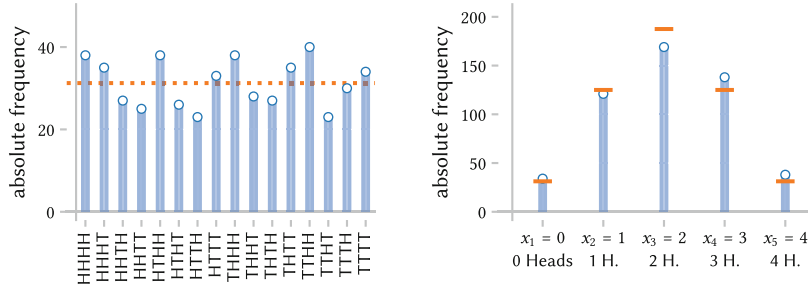


Fig. 6.3 A random experiment of four coin tosses. The plots show the data points as small circles. The vertical columns are used to guide the eye. *Left:* number of times that each possible outcome has occurred; the dotted horizontal line indicates the “theoretical” value. *Right:* number of times that 0, . . . , 4 Heads has occurred. The short orange dashes indicate the ideal or theoretical values

Result x_i	Absolute Frequency n_i	Relative Frequency $P(x_i)$	Cumulative Frequency	Relative cumulative frequency
Heads	4	0.571	4	0.571
Tails	3	0.429	7	1.000
$\sum n_i = N$		$\sum P(x_i) = 1$		

Fig. 6.4 Frequency distribution table and derived quantities (highlighted in light blue) for $N = 7$ rounds of a coin toss experiment

6.3.2 Cumulative Frequencies

Further derived quantities are the *cumulative frequencies*. In this case, the i -th value is obtained by summing up all values of the absolute frequency to and including the i -th event. If instead the relative frequencies are summed up, the *relative cumulative frequency* is obtained. With these definitions, the above frequency distribution table can be extended, as shown in Fig. 6.4. As we will see later, the cumulative distribution function is defined for both discrete and continuous variables, in contrast to the **PMF**, which is only defined for discrete random variables.

6.4 Continuous Probability Distributions

Whenever the possible outcomes of a random experiment can take on continuous values within a certain range of numbers, the relation between values of the random variable and the corresponding probabilities are typically described by the probability density function.

6.4.1 Probability Density Functions

A continuous probability density function (PDF) is often given by the function f . Probability density functions are ubiquitous in materials science and a direct consequence of the continuous nature of many phenomena. For example, “nature never skips” any temperature values between two observed temperature values, but rather varies smoothly—even though quick changes are possible.

Transferring what we have already learned about PMFs to continuous situations is not entirely trivial because measuring the probability “ $P(x)$ ” (i.e. the relative frequency) of a continuous value is not possible. The reason is that the probability with which a random variable takes on a specific value x is infinitesimally small. Why? Because continuous random variables can take on infinitely many different values (recall that an interval with real numbers contains infinitely many real numbers), such that encountering any of them has a vanishing probability.

Therefore, we normally consider the *probability that x is within a certain range* $[a, b]$. This is exactly what can be obtained from *integrating* the PDF f . This is also the reason why the PDF is called a *probability density* (a proper definition will be given in Definition 6.5). Mathematically, this is written as

$$P(a \leq x \leq b) = \int_a^b f_X(x) dx \quad (6.7)$$

$$= \int_{-\infty}^b f_X(x) dx - \int_{-\infty}^a f_X(x) dx \quad (6.8)$$

In other words, the probability P that x is within the interval $[a, b]$ is given by the corresponding area under the curve of the PDF f_X . In this text, we use an explicit notation by including the name of the random variable as a subscript, e.g., the random variable X in $f_X(x)$. Then, x denotes a particular sample. An alternative notation is $f(x)$, which can also be found in the literature and requires the name of the random variable to be given in the context.

We directly find that in Eq. (6.7), the *probability of a point* is obtained from computing $P(x) \equiv P(a \leq x \leq a)$. As a consequence, the two integral limits have the same value and, thus, $P(x) = 0$. This consideration is critical to understand how to interpret PDFs. In terms of a geometrical interpretation, we could say that the area

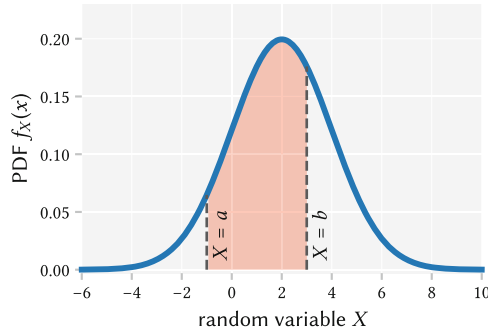


Fig. 6.5 Probability density function (PDF): the light-shaded region graphically represents the area that corresponds to the probability that a random value is between -1 and $+3$. The shaded area would be roughly estimated to be a bit more than half of the total area under the curve, and therefore, the associated probability is somewhat larger than 50% (the exact, computed value is 62.5%)

of a line is zero. An additional consequence is that

$$P(a \leq x \leq b) = P(a < x < b) , \quad (6.9)$$

because the interval “end points” do not contribute any area to the integral. Finally, the area under the whole PDF must be 1,

$$\int_{-\infty}^{+\infty} f_X(x) dx = 1 . \quad (6.10)$$

This is the continuous analogon to the discrete case, in which the sum of the probabilities of all possible events has to be 1, cf. Eq. (6.5).

Figure 6.5 shows a visualization of a PDF. We see that the likelihood of observing a value of $x \approx 2$ is the highest (even though we cannot interpret the maximum value of 0.2 directly). Finding very large positive and negative values, e.g., $|x| > 10$, is quite unlikely. In this way, we can use the PDF to compare different likelihoods, while we cannot quantify them (meaning: we cannot directly express the values of the PDF in terms of a percentage as for probabilities).

Words of advice



In Fig. 6.5, the probability of finding a value of $X = 2$ is *not* 0.2 (or 20%). A PDF needs to be interpreted differently: you always need to associate *areas* under the curve with probability and *not individual points*!

However, we can use Eq. (6.7) to compute the probability of finding a value of, e.g., $-1 \leq x \leq 3$: we compute the integral in the limits of $a = -1$ and $b = 3$, which in Fig. 6.5 is highlighted as the shaded area under the curve. Here, this would give a value of ≈ 0.625 or 62.5%. For example, if this random variable represented a number of displacement measurements in units of millimeters, the probability that the displacement is within $-1 \dots 3$ mm would be 62.5%.

Now assume that the range of displacements is narrowed down to, say, $-1 \dots 1$ mm. Obviously, the area under the curve in this range is smaller, and thus, the probability to find a displacement value between -1 and 1 mm is also smaller (it is only $\approx 24\%$ to be precise). Shifting the whole range by 1 mm again increases the probability. Hence, the probability depends first and foremost on the characteristics of the situation that we describe with a PDF, i.e., the shape of the distribution. Second, the probability also depends on the size and the location of the range within which the area under the curve is calculated.

How continuous is nature?



Many phenomena in physics and materials science can be modeled as continuous functions, which are usually even considered as always differentiable. However, the length and time scales considered (or the resolution with which a problem is viewed) are often decisive in materials science and physics problems:

For example, during a martensitic transformation, the atomic positions of a specimen are switched nearly instantaneously resulting in a different crystal symmetry. If we studied this phenomena by, e.g., taking an *electron backscatter diffraction* (EBSD) image every other second, the transformation would be observed as a time-discrete event with only two possible outcomes (the two states of the material). However, on a nanoscale level, the atomic positions still change only gradually (within a few nanoseconds). Similar scale aspects can be observed in crystalline defects, such as dislocations or when we look at shock waves in solids or fluids.

6.4.2 Cumulative Distribution Functions

An alternative to the PMF and the PDF is the *cumulative distribution function* (CDF). We start by giving a general definition:

Definition 6.4 (Cumulative Distribution Function). The cumulative distribution function F_X of a random variable X is defined as

$$F_X(x) = P(X \leq x) \quad \text{for all } x \in \mathbb{R} . \quad (6.11)$$

where X is the random variable and $\mathbb{R}$ denotes the real numbers.

In this definition, $F_X(x)$ is the probability, and the random variable X has a value smaller than or equal to x . In this text, we use an explicit notation by including the name of the random variable X . This is equivalent to the shorter variant $F(x)$, which can also be found in the literature. In Example 6.4 an example is given, which shows how to obtain the *cumulative distribution function (CDF)* for a discrete random variable. “Jumps” at positions, at which the *PMF* is non-zero for a discrete random variable, are typical features of the *CDF*.

Example 6.4 (CDF of Two Subsequent Coin Tosses)



Assume that we perform two coin tosses in a row and that we are interested in the number of Tails, which represents our random variable X . What is the corresponding *CDF*?

We already know the frequency and can easily infer the probabilities for the three events (i.e., the number of Tails):

$$P(X = 0) = 1/4, \quad P(X = 1) = 1/2, \quad \text{and } P(X = 2) = 1/4 ,$$

from which we directly could plot the *PMF*. To obtain the *CDF* we need to look at three different cases:

$$\begin{aligned} \text{for } x < 0 : \quad & F_X(x) = P(X \leq x) = 0 \\ \text{for } 0 \leq x < 1 : \quad & F_X(x) = P(X \leq x) = P(X = 0) = 1/4 \\ \text{for } 1 \leq x < 2 : \quad & F_X(x) = P(X \leq x) = P(X=0) + P(X = 1) = 1/4 + 1/2 = 3/4 \\ \text{for } x \geq 2 : \quad & F_X(x) = P(X \leq x) = P(X = 0) + P(X=1) + P(X=2) \\ & = 1/4 + 1/2 + 1/4 = 1 . \end{aligned}$$

Unlike for continuous variables, there is a difference between “<” and “≤,” which in Fig. 6.6 is reflected by two different types of markers. Also observe that even though the *PMF* consists of only three points at the coordinates of (0; 0.25), (1; 0.5), and (2; 0.25), the *CDF* consists of lines—even outside the region, in which the *PMF* “has data.”

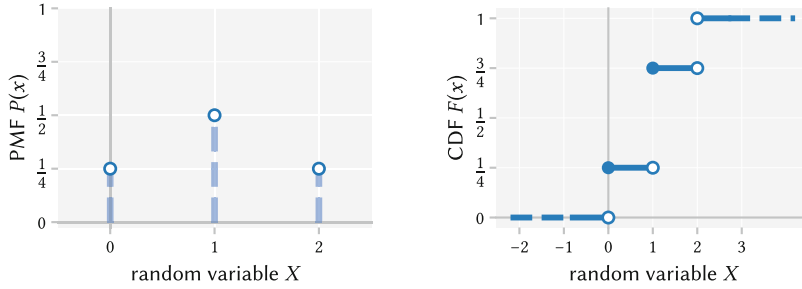


Fig. 6.6 Cumulative distribution function for two coin tosses, where X denotes the number of times we get Tails. The full circle indicates that the value is included, and the open ones indicate a “jump,” i.e., the value is not included as in an open interval

In the case of a continuous random variable, the **CDF** is continuous, and if the **CDF** is differentiable, we can easily obtain the **PDF** as a derivative. In fact, this is one of the definitions of the **PDF**:

Definition 6.5 (Probability density function). Assume that X is a continuous random variable and $F_X(x)$ is the corresponding continuous **CDF**. If $F_X(x)$ is differentiable, the **PDF** $f_X(x)$ is defined as

$$f_X(x) = \frac{dF_X(x)}{dx} . \quad (6.12)$$

With this definition, it follows that the **CDF** of a continuous random variable X is the integral of its **PDF** f_X , and the **CDF** gives the area under the **PDF** from $-\infty$ to x ,

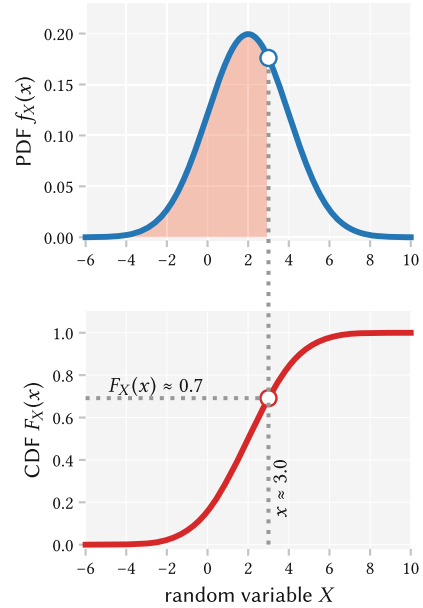
$$F_X(x) = \int_{-\infty}^x f_X(t) dt . \quad (6.13)$$

The probability that the random variable is in the interval $(a, b]$ is

$$P(a < X \leq b) = F_X(b) - F_X(a) = \int_a^b f_X(t) dt . \quad (6.14)$$

Integrating over the complete $\mathbb{R}$ from $-\infty$ to $+\infty$ the integral must be 1

Fig. 6.7 The top panel shows the probability density function (PDF), f_X , and the bottom panel shows the cumulative distribution function (CDF), F_X . Both distributions are evaluated at the point x , which is chosen slightly off the maximum of f_X . The value of the CDF at $X = x$ is obtained as the red-shaded area under the PDF from $-\infty$ up to x . The CDF has an inflection point at $x = 2$, while the PDF reaches its maximum. Values of the variable X at the maximum of the PDF are most likely. Any CDF must approach 1, when the random variable goes toward infinity



$$\int_{-\infty}^{\infty} f_X(t) dt = 1, \quad (6.15)$$

which is equivalent to stating that the total area under the PDF must be 1. The relationship between PDF and CDF is visually explained in Fig. 6.7.

While, in principle, PDF and CDF contain the same information, a CDF always looks “smoother” because it is obtained as an integral of the PDF. Generally, during integration (or also during summation), small values and large values are added up, which effectively is a kind of “averaging.” Visually, but sometimes even computationally, it is difficult to identify the peak of a PDF based on a CDF because inflection points are more difficult to spot than extrema (again: because $F_X(x)$ is smoother). If the shape of a distribution is of interest and we start by looking at a the CDF an additional step is required, e.g., we have to try to create the derivative of the CDF in our mind, which basically shows us the PDF. A convenient property of the CDF, though, is that we can easily tell the probability for a random variable to be below or above a certain value, as this property is, in a way, directly contained in the definition of the CDF. In Fig. 6.7 the probability that $X \leq 3$ is approximately 70%.

Example 6.5 is a complete example that shows how to use the above axioms and equations and how to derive the CDF from a PDF. It also shows how to compute probabilities from the CDF and, therefore, serves as an important template for other problems as well. Exercise 6.3 is a variation of this problem.

Example 6.5 (Computing with PDFs and CDFs)

Assume that for the random variable X , the PDF of the negative exponential function (with the scalar parameter λ) is given by

$$f_X(x) = \begin{cases} \lambda \exp(-x) & \text{for } x \geq 0 \text{ and } \lambda > 0 \\ 0 & \text{otherwise} \end{cases} \quad (6.16)$$

Task 1: Determine λ such that $f_X(x)$ is indeed a PDF For this, we have to find λ such that Eq. (6.15), $\int_{-\infty}^{\infty} f_X(t) dt = 1$, holds. Insertion of f_X gives

$$\int_{-\infty}^0 f_X(t) dt + \int_0^{\infty} f_X(t) dt = -\lambda \exp(-x)|_0^{\infty} = \lambda, \quad (6.17)$$

from which follows that λ must be 1. Additionally, we have $f_X(x) \geq 0$ for all x .

Task 2: Derive the formulation of the CDF We start from the definition on Eq. (6.13), $F_X(x) = \int_{-\infty}^x f_X(t) dt$ and insert the PDF obtained above. For $x < 0$ it is $F_X(x) = 0$, and for the case that $x \geq 0$, it is

$$F_X(x) = \int_0^x f_X(t) dt = -\exp(-t)|_0^x = 1 - \exp(-x). \quad (6.18)$$

Task 3: What is the probability that a randomly drawn sample will have a value in the range of $[1, 3]$ Thus, we are looking for $P(1 < x \leq 3)$ (the $<$ or $\leq$ does not matter here):

$$P(1 < x \leq 3) = F_X(3) - F_X(1) = (1 - \exp(-3)) - (1 - \exp(-1)) \approx 0.319 \quad (6.19)$$

The probability is approximately 32%. The figure below shows the PDF and CDF together with some values relevant for Task 3 (Fig. 6.8).

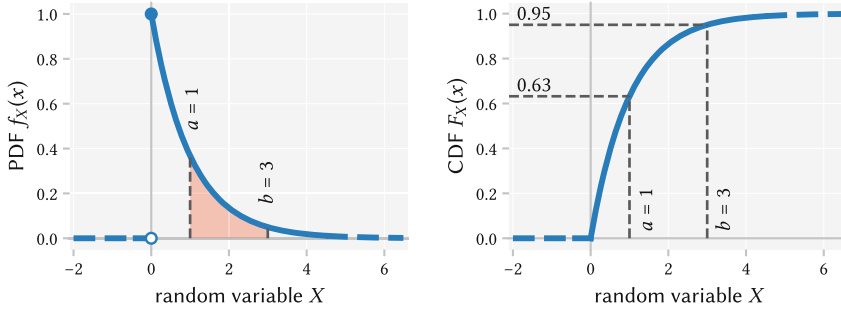


Fig. 6.8 PDF and CDF of the negative exponential function. The vertical difference between the dashed lines is $P(1 < x \leq 3)$

6.4.3 The Difference Between a Function and a Distribution

We conclude this section by answering the question why we differentiate between a function and a distribution. Let's first summarize what a function is: A function in general is a mapping from an n -dimensional space $\mathbb{R}^n$ to an m -dimensional space $\mathbb{R}^m$. For the special case of $m = n = 1$, it is a function that takes a scalar and results in a scalar, e.g.,

$$f(x) \in \mathbb{R} \quad \text{with} \quad x \in \mathbb{R} ,$$

meaning f is the mapping of a value x to a value $f(x)$, which is written as

$$f : x \mapsto f(x) . \quad (6.20)$$

A distribution can be understood as a more general object: some distributions (i.e., the continuous ones) are, in fact, represented by functions, but others may consist of discrete data and are represented by a table as in the case of the PMF.

Because we want to explicitly state this difference in this text, we use the notation $f(x)$ for functions and $f_X(x)$ for distributions. Last but not least, our notation for samples, data records, and the data matrix, as introduced in the ML chapters, also reflects this difference.

6.4.4 A Brief Summary of the Most Important Properties

Here is a brief summary of the most important properties and abbreviations used:

Things to Remember (PDFs and CDFs)

The most important properties of a PDF $f_X(x)$ for a continuous random variable X can be sorted in analogy to the three Kolmogorov axioms, cf. Sect. 5.2.5:

- corresponding to the 1st axiom: $f_X(x) \geq 0$ for all $x \in \mathbb{R}$
(corresponds to Eq. (5.25))
- corresponding to the 2nd axiom: $\int_{-\infty}^{+\infty} f_X(t) dt = 1$
(corresponds to Eq. (5.26))
- corresponding to the 3rd axiom: $P(a < X \leq b) = F_X(b) - F_X(a) = \int_a^b f_X(t) dt$
(corresponds to Eq. (5.27))

The relation between a PDF and a CDF is given by the equation

$$F_X(x) = \int_{-\infty}^x f_X(t) dt. \quad (6.21)$$

6.5 Multivariate Discrete and Continuous Distribution

So far, everything has been formulated in terms of a single random variable. In many situations, however, the relation between several (i.e., ≥ 2) random variables is of interest. For example, consider a Monte Carlo simulation of point defects in metals: we might be interested in how the rate of point defect nucleation is correlated with the effective heat conduction of the material.

We will now start with the most general case of multivariate distributions. Instead of a single random variable, we encounter N random variables $X_1, \dots, X_N$, which we abbreviate¹ as $\mathbf{X}$. The *joint probability* is a notion that we will always encounter in this context. It is closely related to the independence of events (compare Definition 5.13) and defined as follows:

¹ In probability theory, a boldface $\mathbf{X}$ is often used as a short form for a set of random variables X_i . In this text, we use this notation for convenience (and brevity) but in particular for consistency with the later chapters.

Definition 6.6 (Joint Probability). Given are two or more random variables $X_1, \dots, X_N$, each of which representing the occurrence of *independent events* (cf. Definition 5.13), $E_1, \dots, E_N$, such that

$$P(E_1 \cap \dots \cap E_N) = \prod_i P(E_i) . \quad (6.22)$$

The *joint probability* is the probability that *all* N events happen at the same time.

The property of events being independent of each other implies that the events do not influence each other. For example, if two events were the elongation of a specimen in a tensile test and the temperature value, the first event would not influence the latter (if this is different, we have to be very careful when analyzing the data). Note that temperature and elongation can still be correlated, though—this is a completely different cup of tea!

Joint probability distributions, or sometimes *joint probability functions*, are the multivariate pendant to the [PMF](#) and the [PDF](#).

Multivariate Distributions vs Joint Distributions



A multivariate distribution is also called *joint distribution*. Usually, when we talk about a particular example of a distribution, we say, e.g., “the joint distribution of X_1 and X_2 has a maximum at ...” and only when we talk about multivariate distributions in general, then we would rather say, e.g., “multivariate distributions are based on more than one random variable.” Last but not least, when we refer to probabilities of a multivariate distribution, we typically denote this as “joint probability.”

6.5.1 Generic Formulation of the Joint CDF

The joint cumulative distribution function can be defined in a way that holds both for continuous and discrete random variables. Assume a given sample $\mathbf{x} = [x_1, \dots, x_N]$ of N random variables, based on which a [CDF](#) is evaluated. This can be written as

$$F_X(\mathbf{x}) = P(X_1 \leq x_1, \dots, X_N \leq x_N), \quad (6.23)$$

which is the probability that each of the N random variables X_i has a value $\leq x_i$. Recall that the argument of P in this equation is a compact notation for

$$\begin{aligned}
 &P((X_1 \leq x_1) \text{ and } (X_2 \leq x_2) \text{ and } \dots \text{ and } (X_N \leq x_N)) \\
 \Leftrightarrow &P((X_1 \leq x_1) \cap (X_2 \leq x_2) \cap \dots \cap (X_N \leq x_N))
 \end{aligned} \tag{6.24}$$

Each expression $(X_i \leq x_i)$ defines a set of points, and the resulting joint probability is the probability of finding the data in the intersection of all N sets. For brevity, however, we use the shorter form from Eq. (6.23) most of the time.

6.5.2 Joint Probability Functions for Discrete Random Variables

For the joint CDF, the generic formulation from Sect. 6.5.1 holds. A more concrete formulation can be given in terms of summation, as shown below. Subsequently, also the joint mass function is introduced and demonstrated using a formulation with two random variables.

Joint Probability Mass Function (Joint PMF)

The *joint probability mass function* (or sometimes just *joint function* or *joint probability*) is the multivariate pendant to the univariate PMF. It returns the probability that the random variable X_1 has the value x_1 , random variable X_2 has the value x_2 , etc. It is given by

$$P(X_1 = x_1, \dots, X_N = x_N) . \tag{6.25}$$

The comma between the “ $X_i = x_i$ ” terms should be understood as an “and” condition between the terms. As a shorthand notation, we sometimes do not explicitly state the name of the random variable, and only write $p(x_1, \dots, x_N)$, when it is clear from the context that the x_i are the values of random variables instead of mere function arguments.

As in the univariate case (cf. Sect. 6.3), the joint PMF can be described by a “probability table.” For N random variables, the table effectively becomes an N -dimensional array of data. An example for the bivariate case is shown below.

Joint Cumulative Distribution Function (Joint CDF)

The *joint cumulative distribution function* can be obtained from summing joint PMFs p , in analogy to the univariate case:

$$F_X(\mathbf{x}) = P(X_1 \leq x_1, \dots, X_N \leq x_n) = \sum_{\tilde{x}_1 \leq x_1} \cdots \sum_{\tilde{x}_N \leq x_N} p(\tilde{x}_1, \dots, \tilde{x}_N) \tag{6.26}$$

6.5.3 Joint Probability Functions for Continuous Random Variables

As in the discrete case, also for continuous random variables, the generic formulation of the joint cumulative distribution function from Sect. 6.5.1 holds.

Joint probability density function (Joint PDF)

As in the univariate case, we use f to indicate that this is a density distribution that needs to be integrated in order to obtain a probability. The *joint probability density function* of N random variables is then given by

$$f_X(X_1 = x_1, \dots, X_N = x_N) . \quad (6.27)$$

f is a PDF of the random variables $X_1, \dots, X_N$, if (and only if) the following three conditions are met:

- Probability values are always positive or zero:

$$f_X(X_1 = x_1, \dots, X_N = x_N) \geq 0 \quad \text{for all } \mathbf{x} \in \mathbb{R}^N . \quad (6.28)$$

- The integral of all probabilities must be 1:

$$\int_{-\infty}^{+\infty} \dots \int_{-\infty}^{+\infty} f_X(\tilde{x}_1, \dots, \tilde{x}_N) d\tilde{x}_1 \times \dots \times d\tilde{x}_N = 1 . \quad (6.29)$$

- For any subset $A := A_1 \times \dots \times A_N \subset \mathbb{R} \times \dots \times \mathbb{R} = \mathbb{R}^N$, it is

$$P((X_1, \dots, X_N) \in A) = \int_{A_1} \dots \int_{A_N} f_X(\tilde{x}_1, \dots, \tilde{x}_N) d\tilde{x}_1 \times \dots \times d\tilde{x}_N . \quad (6.30)$$

These relations look somewhat more complicated due to a number of variables and integrals; they are, however, the same as those presented in Sect. 6.4.2 for the univariate case.

Joint cumulative distribution function (Joint CDF)

In analogy to the univariate case, the CDF of a continuous random variable X is obtained as the integral of the PDF f_X . For the multivariate case, one has to perform integration for each of the N random variables:

$$F_X(\mathbf{x}) = \int_{-\infty}^{x_1} \dots \int_{-\infty}^{x_N} f_X(t_1, \dots, t_N) dt_1 \times \dots \times dt_N . \quad (6.31)$$

6.5.4 Marginal Probability

Assume that a random variable X assumes the value x regardless of which value the other random variables have. This probability is then written as $P(X = x)$ or, in short, as $p(x)$. As an example, in Fig. 6.9 the number of events contributing to $p(x_i)$ is indicated by the highlighted vertical column. The marginal probability is computed as

$$P(X = x_i) = \frac{1}{N} \sum_{j=1}^r n_{ij} \quad \text{with } N \text{ the total number of events} \quad (6.32)$$

The marginal probability that Y takes the value y_j is computed in analogy by summing over all elements of the respective row,

$$P(Y = y_j) \equiv p(y_j) = \frac{1}{N} \sum_{i=1}^s n_{ij}. \quad (6.33)$$

6.5.5 Conditional Probability

Assume that we now only consider the events with $X = x$. The probability of those events, for which additionally $Y = y$, is the *conditional probability* $P(Y = y | X = x)$. It is written in short form as $p(y|x)$ (read: “probability of y given x ”). In terms of the distribution shown in Fig. 6.9, $p(y|x)$ can be computed as follows:

- (i) restrict the set of all considered outcomes to the highlighted column y_i , i.e., we only consider the absolute frequencies $n_{1j}, \dots, n_{qj}$;
- (ii) then, compute the probability of y_i *only* w.r.t. the events in that column.

Writing this mathematically gives

$$P(Y = y_j | X = x_i) \equiv p(y_j | x_i) = \frac{n_{ij}}{\sum_{j=1}^s n_{ij}}, \quad (6.34)$$

and in full analogy for the other conditional probability

$$P(X = x_i | Y = y_j) \equiv p(x_i | y_j) = \frac{n_{ij}}{\sum_{i=1}^s n_{ij}}. \quad (6.35)$$

Is $p(y_j | x_i)$ the same as $p(x_i | y_j)$?



No, $p(y_j | x_i)$ and $p(x_i | y_j)$ are not the same. To see this, we can take a look at Fig. 6.9. In the first case, the relative frequency of the event at row i and column j is computed with respect to the elements highlighted in the row, while in the second one, it is computed with respect to the column. As the random variables can have different numbers of states, the two conditional probabilities generally are not the same.

As a concrete example, assume that all n_{ij} have the value 1, but X has six possible states and Y has seven. For this case we get $p(y_i | x_i) = 1/7$ and $p(x_i | y_j) = 1/6$.

6.6 Bivariate Distributions as a Special Case

6.6.1 Joint Probability for Two Discrete Random Variables

If we have more than one random variable, the table representing the PMF is a multidimensional array. For two variables, X and Y , this can still be easily visualized. The probability for event x_i and event y_j *both* to happen is the joint probability

$$P(X = x_i, Y = y_j) = \frac{n_{ij}}{N} \quad (6.36)$$

where again N is the number of all possible outcomes and n_{ij} the number of outcomes of event x_i and event y_j . An alternative notation, which is commonly used and borrowed from set theory, is

$$P(X = x_i \cap Y = y_j) . \quad (6.37)$$

A visualization is shown in Fig. 6.9.

6.6.2 Joint Probability for Two Continuous Random Variables

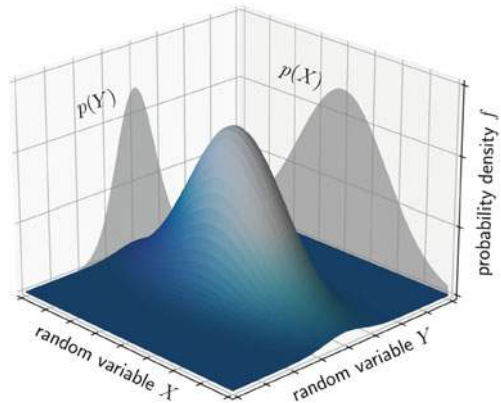
A *joint probability density function* of two random variables, X and Y , is the continuous extension of the PDF in the univariate case. The probability that two variables X and Y are located within a certain value range $[a_X, b_X]$ and $[a_Y, b_Y]$ is obtained from the joint PDF $f_{XY}(x, y)$ as

$$P(X, Y) = \int_{a_X}^{b_X} \int_{a_Y}^{b_Y} f_{XY}(x, y) dx dy . \quad (6.38)$$

	y_1	y_2	y_3	$\cdots$	y_j	$\cdots$	y_r
x_1	n_{11}	n_{12}	n_{13}	$\cdots$	n_{1j}	$\cdots$	
x_2	n_{21}	$\ddots$			n_{2j}		
$\vdots$	$\vdots$				$\vdots$		
x_i	n_{i1}	n_{i2}	$\cdots$	$\cdots$	n_{ij}	$\cdots$	n_{ir}
$\vdots$	$\vdots$				$\vdots$		
x_q	n_{q1}				n_{qj}		

Fig. 6.9 Bivariate probability mass function: visualization as table of absolute frequencies of a discrete distribution for two random variables X and Y with q and r different states, respectively. The number of events with states x_i and y_j are n_{ij} ; the joint probability $P(X = x_i, Y = y_j) = p(x_i, y_j)$ is $n_{ij}/(q \cdot r)$ and highlighted with the dark shading

Fig. 6.10 Bivariate probability density function of the variables X and Y . $p(X)$ and $p(Y)$ indicate the marginal distributions



Geometrically, this can be integrated as the section of a surface integral, which results in the probability $P(X, Y)$. If the considered value ranges of both variables cover the whole $\mathbb{R}$, the probability has to be 1,

$$\int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f_{XY}(x, y) dx dy = 1. \quad (6.39)$$

Figure 6.10 shows a bivariate distribution together with the two marginal distributions.

6.7 Summary

As this was a chapter with two large thematic blocks, we also summarize it with two boxes. Some of the most important aspects of probability theory were:

Things to Remember (Notations in Probability Theory)



- For random variables, we use upper case roman letters, e.g., X , Y , etc.
- A particular value of the random variable X is written as a lower case roman letter, e.g., x ; several measurements or samples of the same random variable X are differentiated by subscripts, e.g., x_1 , x_2 , ...
- Probability density functions (PDFs) and Probability mass functions (PMFs) are both written as $f(x)$; or by additionally indicating the random variable, they are denoted by $f_X(x)$.
- A somewhat sloppy but alternative notation for PMFs is the $p(x)$, which generally stands for a probability *value* (and which is different from a *distribution function*). Similarly, $P(x)$ is used for the (discrete) CDF.

These are the most important notions about discrete and continuous distributions:

Things to Remember Discrete and Continuous Distributions



- For a discrete variable, the probability function is called *probability mass function* (PMF). Its values are the actual probabilities.
- For a continuous random variable, the probability function is the *probability density function* (PDF). The values of the PDF are probability *densities*.
- PMF and PDF are both obtained as derivatives from the *cumulative distribution function* (CDF).
- Some care must be taken for computing the CDF from a PMF as the CDF will contain “discontinuities,” at which the value left of the discontinuity is typically excluded.
- The probability *value* corresponding to a single value of the continuous random variable is zero.

6.8 Exercises

6.1. Assume that the toss of a fair coin was done three times. What is the probability (i) of obtaining three Heads as an outcome; (ii) that the outcome of all tosses are identical; and (iii) that one of the coins lands on the rim?

6.2. Rolling of two dies—what are their summed values, how often do they occur?

- (a) What is the PMF or frequency distribution table that describes the probability distribution for the resulting sum of S from the two dies?
- (b) Implement a Python function that simulates rolling the two dies and that returns the values of two dies.
- (c) Plot the frequency distribution from 100 dice rolls for sum values from 1 to 14 and compare it to your reference solution from (a).
- (d) Plot the PMF that corresponds to the data from (c)

6.3. Here, we will now use the negative exponential distribution as one of the simplest distributions to compute probabilities. The exponential distribution is given by

$$f(z) = \frac{1}{\lambda} \exp(-z/\lambda) \quad (6.40)$$

where λ is a real constant.

- (a) Plot the PDF for $\lambda = 1/2$ and for $\lambda = 2$.
- (b) Derive the position of the maximum of the distribution $f(z)$. What happens for $z = 0$ and for $z \rightarrow \infty$?
- (c) What is the probability that a value, randomly chosen from numbers that follow the exponential distribution, will have z in the range of $1/4$ to $1/2$?

6.4. For the given data, classify them into one of the three categories: discrete *random variables* (RVs), continuous RVs, and categorical RVs:

1. A Geiger-Müller tube is used to detect radioactivity, and the measuring device gives the “count per minute” (CPM) of the ionization events due to radioactivity (this can be made audible as the “clicking” sounds). What kind of RV is the CPM?
2. During an experiment, the elongation d of a rod is measured at distinct points in time—what kind of RV is d ?

6.5. The Maxwell-Boltzmann equation defines the distribution of speed for an ideal gas at a given temperature.

$$f(c) = 4\pi c^2 \left(\frac{m}{2\pi k_B T} \right)^{\frac{3}{2}} e^{\left(\frac{-mc^2}{2k_B T} \right)} \quad (6.41)$$

c is the scalar quantity of speed, m is the mass, T is the temperature, k_B is the Boltzmann constant.

- (a) Plot the PDF for He at room temperature. Note that $m_{He} = 4$ amu.
- (b) Find the most probable speed.
- (c) How does the PDF change with increasing temperature?

- (d) Additionally, consider different noble gases, such as Ne ($m_{Ne} = 20$ amu), Ar ($m_{Ar} = 40$ amu), and Xe ($m_{Xe} = 131$ amu). How does the most probable speed change?

6.6. Two artists in a circus claim that they are able to communicate telepathically. Since they insist that you test their claim, you ask them each to go to a different room, far away from each other. Then you show person A a number of cards, each of which either shows an apple, a cherry, or a banana. Each of the fruits are equally likely to show up. For each card that you show to person A, person B writes down the name of the fruit, e.g., “cherry,” according to what she believes the other person is seeing. What is the probability that person B correctly “saw” at least one card after ten cards were shown?

“Life is so constructed, that the event does not, cannot, will not, match the expectation.”

Charlotte Brontë (1816–1855)

English novelist and poet

7.1 Expected Values of Discrete Random Variables

In the previous chapters, we have already encountered a few situations, in which a number of numerical outcomes of random experiments were to be simplified and described by a single number. For example, we mentioned a number of times phrases such as “*on average the outcome is ...*” or “*the mean of throwing a die 10 times is ...*” and assumed that the reader intuitively knew the meaning. In what follows, we will introduce the expected value and the variance, both of which have the purpose of representing important features of distributions of numbers as a simple number. Subsequently, we will then introduce the concept of moments as the generalization of expected value and variance.

If we were to summarize the most important aspects of a distribution of numerical data through a single value, this would most likely be the *expected value* (or expectation value) of the outcome of a random experiment, more commonly known as the *average* or *mean*. The subsequent formulations and examples are based on discrete random variables. However, the formalism can be easily generalized to continuous random variables, as will be done in the section about *moments*.

7.1.1 Definition and Examples

Assume the random experiment of tossing a coin four times, as also investigated in Example 6.2. While the outcomes of the experiment in Example 6.2 was non-

numerical (e.g., the sequence HHHT), we obtained numerical values with the mapping of the random variable X (which differentiated between five different types of events). Then the *expectation value* (also called *expected value* or just *expectation*) of a random variable X is, roughly said, computed as the arithmetic mean of a large number of outcomes of X . Different values of the random variable could occur with different probabilities, in which case the weighted arithmetic mean would be used:

Definition 7.1 (Expected Value) Let X be a discrete random variable with n possible outcomes given by the $x_1, \dots, x_n$. Each of the outcomes may occur with the probability p_i . Then the expected value of X , written as $E[X]$, is defined as

$$E[X] = x_1 p_1 + x_2 p_2 + \dots + x_n p_n = \sum_{i=1}^n x_i p_i . \quad (7.1)$$

$E[\cdot]$ is called *expectation operator*. If all outcomes are equally likely, the expected value reduces to the “regular” arithmetic mean:

$$E[X] = \frac{1}{n} \sum_{i=1}^n x_i . \quad (7.2)$$

There are a number of different notations for the expectation value. A capital upright “E,” as used in the above definition, is one of the most commonly used notations in statistics and probability theory. To indicate that E is an operator, we use square brackets instead of regular parentheses. A short form is μ_X or, if there is no ambiguity, just μ . Another often used notation in statistics is $\mathbb{E}[\cdot]$. An example illustrating how to compute the expected value is given in Example 7.1 (which is the continuation of the above example Example 6.2).

Example 7.1 Random Variables & Event Probabilities (Part 3, Expected Value) ✕

[...continued from Example 6.2] What is the expected value of four times tossing a coin? Since our random variable gives the number of Heads, we have a numerical random variable (even though H and T are not numbers!), for which it is reasonable to compute an expected value.

$$\begin{array}{ccccccccc} x_1 = 0, & x_2 = 1, & x_3 = 2, & x_4 = 3, & x_5 = 4 \\ p_1 = 1/16, & p_2 = 1/4, & p_3 = 3/8, & p_4 = 1/4, & p_5 = 1/16, \end{array}$$

(continued)

from which follows for the expected value

$$E[X] = 0 \times \frac{1}{16} + 1 \times \frac{1}{4} + 2 \times \frac{3}{8} + 3 \times \frac{1}{4} + 4 \times \frac{1}{16} = 2. \quad (7.3)$$

In other words: the expected value of this random process would be $X = 2$ or the event of 2 Heads.

Even though the expectation value for outcomes of equal likelihood is computed in the same way as the “arithmetic average,” there is a slight conceptual difference: the *average* value is typically computed *after* the data was produced, while the expectation value suggests that we investigate a random experiment; the experiment to obtain the values of the random trials is still to be performed. Beware that there is also the notion of the *mean*, which we will sometimes use in the context of distributions and which comes in three different “flavors”: the arithmetic, the harmonic, and the geometric mean (note that usually the arithmetic mean is referred to if someone talks about “the mean”).

The way we interpret the word “expected” in expected value in our daily language is slightly different from the mathematical meaning. Imagine you throw a six-sided die six times and you get 6, 6, 3, 1, 6, 5. Based on this experiment, throwing a 6 would be considered the most likely event. However, the expected value is $(1 + 3 + 5 + 3 \times 6)/6 = 4.5$, which in fact is a number that can never occur (as the die shows only integer numbers). The expected value of 4.5 has to be interpreted in terms of a prediction for a much larger number of trials, e.g., if we consider the situation where we only take a look at the summed up results of 1000 trials, then this would result in a value of 4500. Clearly, finding this exact value is fairly unlikely to happen as there are many other possible sums.

7.1.2 Calculating with Expectation Values

A number of basic properties for expectation values help to perform calculations. In the following we only list the most important properties and sketch the proofs, which helps to see how the relations can be used for calculations with expectation values.

Additivity

The expectation operator $E[\cdot]$ is a linear operator, and it holds for two random variables X and Y :

$$E[X + Y] = E[X] + E[Y]. \quad (7.4)$$

The short proof uses the definition of the expectation operator for the case of equal likelihoods and discrete random variables:

$$E[X + Y] = \frac{1}{n} \sum_{i=1}^n (x_i + y_i) = \frac{1}{n} \sum_{i=1}^n x_i + \frac{1}{n} \sum_{i=1}^n y_i = E[X] + E[Y] . \quad (7.5)$$

The property of additivity of expectation values can be useful, in particular when a random variable can be written as a sum of random variables that are easier to compute. On the other hand, when computing the individual expectation values is difficult, it might be possible to compute the expectation of the sum.

Homogeneity

Assume that c is a real valued constant and X is a random variable. Then, it is

$$c E[X] = E[cX] . \quad (7.6)$$

The short proof again uses the definition of the expectation operator and assumes equal likelihood of events and discrete random variables:

$$c E[X] = c \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} \sum_{i=1}^n cx_i = E[cX] . \quad (7.7)$$

Thus, the property Eq.(7.6) tells how the expected value changes upon *scaling of a random variable (RV)* by a factor: also the expected value is scaled by the same factor. As a special case, we obtain the expectation value of a constant as the constant itself,

$$E[c] = c . \quad (7.8)$$

This is intuitively clear, e.g., by visualizing some data points on a coordinate axis and considering, e.g., a “zooming in” on the axis scale.

Linearity

From the property of additivity and homogeneity, we conclude that the expectation operator $E[\cdot]$ is a linear operator and can in general be written as $E[aX + bY] = a E[X] + b E[Y]$. Finally, by induction, it can be easily shown that linearity (and therefore additivity) also holds for sums of arbitrary numbers of random variables. Note that we did not have to make any assumptions about the independence of the two random variables. In other words, linearity holds always, irrespective of whether the random variables are independent or not.

Example and Application of the Linearity Theorem

How is this relation useful? Imagine a random process: two dices are thrown, and we record their sum. For computing the expected value of the sum of rolling the two

dices, we could either consider the $6 + 5 + \cdots + 1 = 21$ combinations and compute their expected value—or we make use of the already known expected value for each of the two dices, which is 3.5. We then conclude that $3.5 + 3.5 = 7$ is the expected value of the sum.

Translating

Last but not least, there is a theorem that shows how a shift of a [RV](#) influences the expectation value. The proof is omitted here, but follows the same idea as before and can be done as an exercise. Assume events of equal likelihood described by a random variable X and additionally a constant a . Then the shifting (or: translation) of the random variable is given by

$$E[X + a] = E[X] + a . \tag{7.9}$$

This relationship denotes that shifting the values of a random variable by a constant also shifts the corresponding expected value, which is intuitively clear.

Expected Value of a Product

A last theorem requires that the two random variables, X and Y , are independent variables. Then, it can be shown (cf. Theorem [B.1](#) in [Appendix B](#)) that the expectation of the product of the two random variables is equal to the product of the expectations:

$$E[XY] = E[X] \cdot E[Y] . \tag{7.10}$$

An example together with a Python solution is given in [Example 7.2](#).

Example 7.2 Expected Value of a Product

A simple example is to throw a six-sided die and additionally to flip a coin. Let us furthermore map the resulting symbols by the two random variables X and Y as follows:

symbol:	H	T	☐	☐	☐	☐	☐	☐
	↓	↓	↓	↓	↓	↓	↓	↓
value:	$x_1 = 3,$	$x_2 = 4,$	$y_1 = 5,$	$y_2 = 6,$	$y_3 = 7,$	$y_4 = 8,$	$y_5 = 9,$	$y_6 = 10$

The values x_i and y_i are chosen arbitrarily. Then, the expected values for the two variables are $E[X] = 3.5$ and $E[Y] = 7.5$. Altogether, there are $2 \times 6 = 12$ outcomes with equal likelihood. E.g., (H, ☐) corresponds to



(continued)

$x_1y_1 = 3 \times 5 = 15$, (H, $\square$) corresponds to $x_1y_2 = 3 \times 6 = 18$, etc., up to (T, $\blacksquare$) corresponds to $x_2y_6 = 4 \times 10 = 40$.

Instead of computing the terms and products manually, we can use the following Python code. We start by setting up two dictionaries (our “maps”):

```
In [1]: X = {'H': 3, 'T': 4}
        Y = {i: i+4 for i in range(1, 7)}
        print('X =', X, '\nY =', Y)

X = {'H': 3, 'T': 4}
Y = {1: 5, 2: 6, 3: 7, 4: 8, 5: 9, 6: 10}
```

Next, create pairs of the results from the coin and the die using a Cartesian product. Summing up the x_iy_j and dividing by the total number of combinations gives $E[X \cdot Y]$:

```
In [2]: from itertools import product
        sum_XY = 0
        for x, y in product(X.values(), Y.values()):
            sum_XY += x * y
        E_XY = sum_XY / (len(X) * len(Y))
        print("E_XY =", E_XY)

E_XY = 26.25
```

As an alternative we compute the expectation value for the individual random variables

```
In [3]: E_X = sum(X.values()) / len(X)
        E_Y = sum(Y.values()) / len(Y)
        print("E_X * E_Y =", E_X * E_Y)

E_X * E_Y = 26.25
```

..., and their product gives the same value as before. Obviously the calculation of $E[X] \cdot E[Y]$ is much easier than computing $E[X \cdot Y]$, in particular for larger numbers of items or for increasing numbers of variables.

Independent and Identically Distributed Random Variables

An important notion is introduced in the following definition:

Definition 7.2 (Independent and Identically Distributed (IID))

A number of random variables is called *independent and identically distributed* if (i) each random variable is associated with the same probability distribution and (ii) all random variables are mutually independent.

In this case, it is

$$E[X_1 + \cdots + X_N] = N E[X] , \quad (7.11)$$

where X is any of the random variables X_1 to X_N . As a consequence, if we can show that for a set of random variables Eq. (7.11) holds, we can conclude that they are IID. Making the assumption that variables are IID simplifies many formulations in statistics and probability theory.

7.2 Variance and Standard Deviation

Imagine two random variables: the first one has the outcomes $[3, 4, 5, 6, 7]$ and the second one the outcomes $[-5, 0, 5, 10, 15]$. Each of them has the same expectation value of 5, but the values of the second random variable are clearly much more “spread out.” A random variable can be characterized by various measures other than the expectation value alone. One of such commonly used characteristics in probability theory is the *variance*, which quantifies the “spread” of values and the statistical “uncertainty.”

7.2.1 Definition and Examples

Similar to the expectation value, the variance is also an operator; it is defined as follows:

Definition 7.3 (Variance) The variance $\text{Var}(X)$ of a random variable X is given by

$$\text{Var}(X) = E[(X - E[X])^2] = E[(X - \mu)^2] \quad (7.12)$$

(continued)

where we used the already above-introduced abbreviation $\mu = E[X]$.

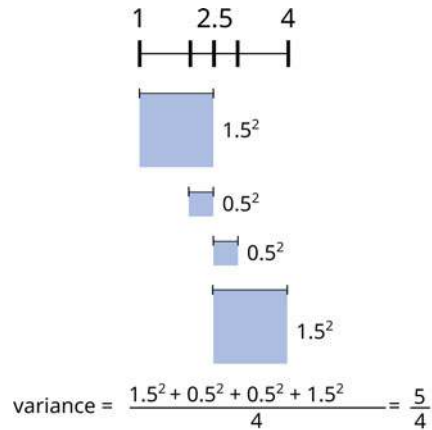
From the definition of the expectation value, Definition 7.1, it follows: if $x_1, \dots, x_n$ are the possible outcomes, and each outcome x_i may occur with the probability p_i , then

$$\text{Var}(X) = \sum_{i=1}^n p_i \cdot (x_i - \mu)^2. \quad (7.13)$$

Why does it measure the “spread” of a variable? First, the values are “centered” around zero due to the subtraction of the “mean.” This allows the comparison of different variables because, as a consequence, the mean will not have an influence on the value of the variance. The squared term then acts as a nonlinear “weighting”; thus, values further away from the expectation value have a more pronounced influence on the value of the variance. Additionally, the squared term makes the variance independent of the sign of the values (Fig. 7.1).

Note that in the context of statistics (c.f., Chap. 8), we will encounter two ways to compute variances: one of them considers all possible data, i.e., the *population variance*, and the other one considers only a subset of the data, i.e., the *sample variance*. There is a small mathematical difference between them, which is important, in particular, when we deal with small amounts of data. We will discuss this in detail in Chap. 8.

Fig. 7.1 Visualization of the variance for the values of $\{1, 2, 3, 4\}$: due to the squared term in the definition, the distance of the data points from the mean of 2.5 is represented as the square areas



7.2.2 The Standard Deviation

The squared term contained in the equation for the variance has the effect that it is more difficult to interpret than the expected value. This is the reason for introducing the *standard deviation* as the square root of the variance:

Definition 7.4 (Standard Deviation) The standard deviation σ of a random variable X is defined as

$$\sigma(X) = \sqrt{\text{Var}(X)} . \quad (7.14)$$

The standard deviation has the same unit as the random variable. An example for computing variance and standard deviation is given in Example 7.3.

Example 7.3 Variance of Results from Rolling a Six-Sided Die



What is the variance and standard deviation of throwing a fair, six-sided die? How can this be interpreted?

Starting from Definition 7.3 with the expectation value $E[X] = 3.5$ we have

$$\text{Var}(X) = E[(X - 3.5)^2] = \frac{1}{6} \sum_{x=i}^6 (x_i - 3.5)^2 . \quad (7.15)$$

For the six-sided die, it is $x_1 = 1, x_2 = 2, \dots$ and therefore

$$\text{Var}(X) = \frac{1}{6} \left((1 - 3.5)^2 + \dots + (6 - 3.5)^2 \right) = 2.92 . \quad (7.16)$$

The standard deviation σ is then, according to Definition 7.4

$$\sigma = \sqrt{\text{Var}(X)} = \sqrt{2.92} \approx 1.71 \quad (7.17)$$

7.2.3 Calculating with Variances

For calculating with the variance of random variables, a number of useful theorems and properties can be stated. We will summarize them in the following, while some of the proofs are given in Appendix B.2.

- **Non-negativity:** The values of the variance are non-negative, $\text{Var}(X) \geq 0$, which is a side effect of the squaring in the definition of the variance.
- **Constant values:** The variance of a constant c is zero. This implies in turn that if the variance of a random variable is zero, all values x_i are identical.
- **Additivity:** For two *independent* random variables, it is

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) .$$

While this was obvious for the case of the expected value of two random variables, it is not straightforward for the variance because the variance is not a linear operator as the expected value operator.

- **Scaling:** For a random variable X and a scalar value c , it is

$$\text{Var}(cX) = c^2 \text{Var}(X)$$

- **Translation:** The variance is invariant w.r.t. a shifting of the random variable values, $\text{Var}(X + a) = \text{Var}(X)$.

7.3 Raw Moments

In this and the following sections, the concepts of different moments are introduced for both discrete and continuous RVs. We will find that the expectation value and the variance are both measures that can be obtained as special cases from particular types of moments; thus, moments can be understood as the more general concept. Additionally, the formulations for multivariate distributions can also be easily obtained. In the following we start with the most basic variant, i.e., the raw moments.

7.3.1 General Formulation

The n -th moment about 0 of a probability function is the expected value of the random variable X^n ; it is called a *raw moment* or (less commonly) also *crude moment*:

Definition 7.5 (The n -th raw moment of X) The n -th raw moment of a random variable X is denoted as $E[X^n]$. It is defined for a discrete probability mass function p with the discrete values x_i as

$$E[X^n] = \sum_i x_i^n p_i \quad (7.18)$$

(continued)

and for a continuous probability density function $f(x)$ as

$$E[X^n] = \int_{-\infty}^{+\infty} x^n f(x) dx . \quad (7.19)$$

Moments of multivariate distributions can also be obtained. However, in most cases it is sufficient to characterize each random variable separately. This can be done through *marginalization* and the *marginal probability*, as introduced in Sect. 6.5.4. For example, for a bivariate continuous probability density function with the random variables X and Y , the n -th raw moment of X and Y , respectively, is

$$E[X^n] = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} x^n f(x, y) dx dy , \quad (7.20)$$

$$\text{and } E[Y^n] = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} y^n f(x, y) dx dy . \quad (7.21)$$

These formulations can be generalized to more RVs by performing the additional integrations. The formulations for the discrete case follow accordingly with the definition of the raw moment and the joint probability from Sect. 6.5.2.

Moments in Physics and Mechanics



The reader might have come across “moments” already in a different context, e.g., in mechanics or in physics. In general, a moment is a mathematical measure for the particular shape of a mathematical function. Translating this into the world of physics or solid mechanics, the mathematical function becomes a physical density in units weight per area. In that case the zero-th moment gives the total mass, the first moment divided by the total mass is the center of mass, and the second moment is the rotational inertia.

7.3.2 The Zero-th Raw Moment

The *zero-th moment* of any probability function is obtained by setting $n = 0$: then, the sum or integral is formed over the whole probability, which is 1 (already per definition):

$$E[X^0] = E[1] = \sum_{i=1}^n p_i = 1 \quad \text{for discrete RVs} \quad (7.22)$$

$$E[X^0] = E[1] = \int_{-\infty}^{+\infty} f(x) dx = 1 \quad \text{for continuous RVs} \quad (7.23)$$

7.3.3 The First Raw Moment or the Mean

Even though in principle also higher raw moments can be used, in practical situations only the *first raw moment* is relevant. It is obtained for $n = 1$ and as the expectation value

$$E[X^1] \equiv E[X] = \sum_{i=1}^N x_i p_i \quad \text{for discrete RVs} \quad (7.24)$$

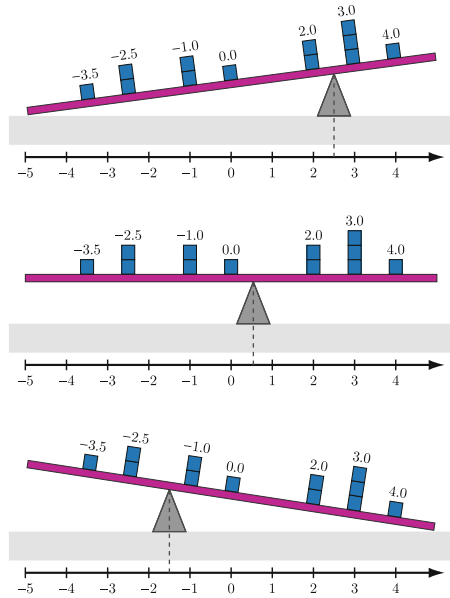
$$E[X^1] \equiv E[X] = \int_{-\infty}^{+\infty} x f(x) dx \quad \text{for continuous RVs.} \quad (7.25)$$

These two equations are usually used to define the mean $\mu = E[X]$. The mean of a discrete RV has the property that each data point contributes the same fraction of its value as the other ones, including potential outliers (i.e., rare values further away from the mean of the other values). For example, in the dataset $\{2, 4, 6, 50\}$, each of the four numbers contributes with a weighting of 0.25, and therefore the “outlier” value of 50 increases the mean strongly: the mean of the first three values is 4, whereas the mean of *all* values would be 15.5. Geometrically, the mean can be understood as the point at which the sum of the data to the right is exactly balanced by the sum of the data to the left, as depicted in Fig. 7.2. This can be also transferred to continuous random variables, for which the mean divides the total area into two sub-areas of equal size.

7.4 Central Moments

Central moments are the moments w.r.t. to the mean. As opposed to the raw moments, they describe aspects of the spread or shape of the function *independent* of translations and achieved by subtracting the mean.

Fig. 7.2 Visualization of the mean as the point at which the dataset is “balanced.” The dataset consists of the following points: $\{-1, 2, -3.5, 0, 3, 3, -2.5, 4, -2.5, 2, 3\}$. The height of the blue bars corresponds to the different frequencies with which the values occur. The sketch in the middle shows the point that balances the data at the mean value of ≈ 0.54 . Each blue bar can be interpreted as a mass, which creates an angular momentum proportional to the number of rectangles (the “force”) and the distance from the mean



7.4.1 General Formulation

In the following the definition of arbitrary degrees of central moments of a distribution is given:

Definition 7.6 (n -th central moment) The n -th central moment of a random variable X is denoted by $\mu[X^n] = E[(X - \mu)^n]$. In the discrete case it is given for the N data points by

$$\mu[X^n] = \sum_{i=1}^N (x_i - \mu)^n p_i, \quad (7.26)$$

while in the continuous case, it is given by

$$\mu[X^n] = \int_{-\infty}^{+\infty} (x - \mu)^n f(x) dx. \quad (7.27)$$

In this definition, $\mu = E[X]$ is the mean, as before. All other quantities x_i , x , and f are defined as in Definition 7.5. As in the case of raw moments, also the central moments can be computed for multivariate distributions.

7.4.2 Zero-th and First Central Moments

It follows directly that the 0-th central moment of any probability density distribution is 1. Furthermore, the *first central moment* ($n = 1$) of any distribution is zero because it is the expectation value of a variable that was already “centered” at the mean.

7.4.3 The Second Central Moment or the Variance

The second moment is the variance of the observations,

$$\text{Var}[X] \equiv \text{Var}[X] = \mathbb{E}[(X - \mu)^2] = \sigma^2, \quad (7.28)$$

which can also be calculated from the following relation:

$$\text{Var}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2. \quad (7.29)$$

The variance characterizes the spread from the average: a low value of variance implies that all data are close to the mean, while a high variance indicates that the data are strongly spread out.

For a discrete variable, the variance is given by

$$\text{Var}[X] = \sum_i (x_i - \mu)^2 p_i = \sum_i x_i^2 p_i - \mu^2 \quad (7.30)$$

and in case of a continuous random variable by

$$\mathbb{E}[(X - \mu)^2] = \int_{-\infty}^{+\infty} (x - \mu)^2 f(x) dx. \quad (7.31)$$

Example 7.4 Implementation details in terms of linear algebra



If we define the column vector $\mathbf{o} = [1, \dots, 1]^T$ (with n entries), and the dataset is given by the column vector $\mathbf{x} = [x_1, \dots, x_n]^T$, the mean can be represented by scalar products

$$\bar{x} = \frac{\mathbf{o}^T \cdot \mathbf{x}}{\mathbf{o}^T \cdot \mathbf{o}}, \quad \text{where } \mathbf{o}^T \cdot \mathbf{o} = n$$

(continued)

which is the projection of $\mathbf{x}$ onto $\mathbf{o}$. For the (population) variance, we can write

$$\text{Var}(\mathbf{x}) = \frac{\|\mathbf{x} - \bar{\mathbf{x}}\mathbf{o}\|^2}{n} \quad \text{where } \|\mathbf{y}\|^2 = \mathbf{y}^T \mathbf{y}$$

This can now be “translated” to Python code, which follows the same structure as the equations.

```
1 import numpy as np
2 x = np.random.rand(5)
3 o = np.ones_like(x)
4 mean = o.dot(x) / o.dot(o)
5 var = np.linalg.norm(x - mean * o) ** 2 / (o.dot(o))
6 np.testing.assert_allclose(mean, x.mean())
7 np.testing.assert_allclose(var, x.var())
```

The last two lines check if the computed mean and variance are, within some tolerance, identical to the “reference values” directly obtained from `numpy`. Such formulation in terms of scalar products of vectors is also used for efficient implementations of many numerical methods.

7.5 Standardized Moments

The attribute *standardized* (or also *normalized*) implies that moments are calculated after the distribution was normalized with the standard deviation. This makes normalized central moments dimensionless and, thus, scale invariant. Also standardized moments allow to indirectly compare the shapes of different distributions.

7.5.1 General Formulation

We start again with the general formulation of the n -th standardized moment:

Definition 7.7 (Standardized Central Moment) The standardized n -th central moment of the random variable X is defined as

$$\tilde{\mu}_n = \frac{\mu_n}{\sigma^n} = \frac{\mathbb{E}[(X - \mu)^n]}{\sigma^n}. \quad (7.32)$$

In this definition, σ denotes the variance, and σ^n is the variance to the n -th power.

7.5.2 Aspects of Standardized Moments

As for the first central moment, also the first standardized moment is zero, $\tilde{\mu}_1 = 0$, as it is the first moment of a centered variable. The second standardized moment is always 1 because the second *centered* moment (the nominator of Eq. (7.32)) is the variance of X . The third and fourth moments are dimensionless measures for the skewness and kurtosis of distributions and will be discussed in more detail in the context of descriptive statistics in Sect. 9.3.

7.6 Exercises

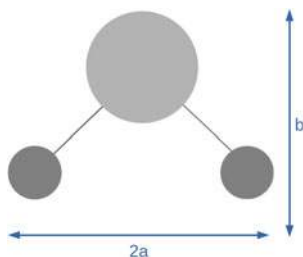
7.1 Show that the first central moment of any distribution is zero.

7.2 This is the continuation of Exercise 6.5:

- (a) Based on the Maxwell-Boltzmann distribution calculated for He at room temperature, obtain the mean and standard deviation.
- (b) How does the standard deviation change with increasing temperature?

7.3 A fair, n -sided die can be modeled as a discrete random variable, X , with outcomes $1 \dots n$, each with equal probability $1/n$. What is the general formula for the variance of the outcome, X , of an n -sided die?

7.4 Consider a two-dimensional water molecule, as schematically drawn in the figure below. Assume that the masses of the H and O atoms are 1 and 16, respectively (dimensionless).



- (a) Calculate the first and second moment.
- (b) How do these moments relate to the center of mass and moment of inertia?

7.5 In quantum mechanics, one of the most fundamental concepts is that of a particle in a box. The wave function (or simply put, a function that describes the ground-state position of a particle) is given by:

$$f(x) = \left(\frac{2}{L}\right)^{\frac{1}{2}} \sin\left(\frac{\pi x}{L}\right)$$

where L is the length of the box. The corresponding probability density is:

$$p(x) = \left(\frac{2}{L}\right) \sin^2\left(\frac{\pi x}{L}\right)$$

- (a) Calculate the expectation value of the position of the particle.
- (b) Calculate the uncertainty in the position of the particle, given by the standard deviation.



Statistics is the grammar of science.

Karl Pearson (1857–1936)

English Statistician

8.1 And What Now Is Statistics?

To start with, probability theory and statistics are both subfields of mathematics. However, they approach problems from opposite directions: in probability theory the point of departure is some random process, and based on a mathematical description of this, we try to derive the result—typically using purely mathematical reasoning and theoretical derivations. In statistics, we roughly take the opposite approach: instead of using theoretical derivations, we rather *observe* certain phenomena and, based on that, try to infer what the underlying process might be, as demonstrated in Example 8.1.

What is the benefit of statistics and probability theory for data science? In later chapters it will turn out that in fact much of *machine learning* (ML) is statistics and probability theory. Additionally, statistics helps to decide if the data used for training of a machine learning model is in “a good state” (or not) and if differences between ML experiments are real or just effects of noise in the data.

In this text, we will consider statistics as a collection of mathematical tools and concepts that are able to help us explore and answer questions about data (e.g., what does the data look like? what is the most common observation?) and that will guide us during the process of gathering data. We try to reduce the amount of technical or theoretical considerations to a reasonable amount; for more in-depth considerations, the reader is referred to the great body of literature.

Example 8.1 (Probability Theory vs. Statistics)



Imagine a jar filled with colored marbles. Typical perspectives are in ...

- ... *Probability Theory*: Start by thinking about a possible underlying probabilistic model that, e.g., could mathematically describe the random filling process of the jar. And based on that, we could then predict the probability to draw either a red or a blue marble.
- ... *Statistics*: Draw a number of marbles from the jar (which is the process of “sampling”) and assume that the rest of the marbles in the jar have colors in proportion to what we got in our sample. Additionally, by describing properties of our sample, we would conclude how the property of all marbles in the jar (e.g., the number of marbles with the same color) are.



8.2 The Sample and the Population

One of the strengths of statistics becomes clearly visible once the number of objects in which we are interested becomes larger. For example, in the above example, we might be interested in finding out what the ratio between the number of blue marbles and the total number of marbles is.

8.2.1 The Sample and the Population

The straightforward approach is to first count all blue marbles, then count all remaining marbles so that we can compute the total number of marbles, and finally divide the number of blue marbles by the total number of marbles. While this gives an accurate answer to the investigated question, in most situations this is a highly impractical approach. In statistics we take a different approach and rather randomly draw a number of marbles from the jar. Instead of investigating the *whole* jar, only this subset of marbles is then investigated. The hope of the statistician is that this subset is a *representative* set from which the property of the whole jar can be *estimated*. The subset is called *sample*, and the whole jar of marbles is the *population*. Let us now define the notion of the population:

Definition 8.1 (Population) In statistics, a *population* is the aggregations of all objects of interest.

There are two noteworthy aspects: the beginning of the definition “in statistics” suggests that other fields might have different definitions of populations. For example, in everyday language usage, we would associate “population” with the inhabitants of a city or a country. The population in statistics does not necessarily have anything to do with people. The second interesting aspect in the short definition above concerns the “objects of interest.” Once we are interested in different aspects or properties, this part of the sentence implies that the relevant population can change and is nothing that is fixed. Example 8.2 highlights the populations of two different statistical experiments.

Example 8.2 (Population)



- In Example 8.1 the population consists of the content of the whole jar.
- If we performed 400 mechanical tests (60% bending tests and the rest tensile tests) and are interested in the experiments in which specimens fail during the tensile loading, then the population consists of 160 experiments or specimens. If we are interested in how many specimens fail *in total*, then the population consists of all 400 specimens.

In many situations we are not able to investigate an entire population, e.g., because it is too expensive, it would take too long, or because the population consists of infinitely many members. In statistics, this is a well-known and fundamental problem. Therefore, at the heart of statistical analysis is the concepts of the sample:

Definition 8.2 (Sample and Sampling) A *sample* is a subset of the entire population. The process of picking the data from the population is called *sampling*.

The purpose of a sample is that it represents the populations while at the same time being a smaller and much easier to handle subset of the population.

8.3 Two Flavors: Descriptive and Inferential Statistics

This is then the point of departure for two different statistical approaches: By computing statistical measures such as the mean value or the variance of the data in the sample, we ...

- characterize and describe the sample through *descriptive statistics*.
- hope to be able to predict statistical properties of the population based on the properties of the sample. This is the purpose of *inferential statistics*.

A significant part of Chap. 9 is dedicated to various aspects of the so-called *exploratory data analysis* (EDA), which includes descriptive statistics as a major topic. Aspects of inferential statistics will be discussed in the context of machine learning starting from Chap. 11.

A visualization of the concepts of the population and the sample is given in Fig. 8.1, which also gives an overview of the task for which the different sub-disciplines within statistics are responsible. There, the two circles within the population indicate two different samples. Sample #1 results in an equal number of blue triangles, orange squares, and red circles. *Making inference* then is the process of extrapolating the properties of the sample to the whole population. Here, one could *infer* from sample #1 that also the whole population will consist of equal number of triangles, squares, and circles—which is already pretty close

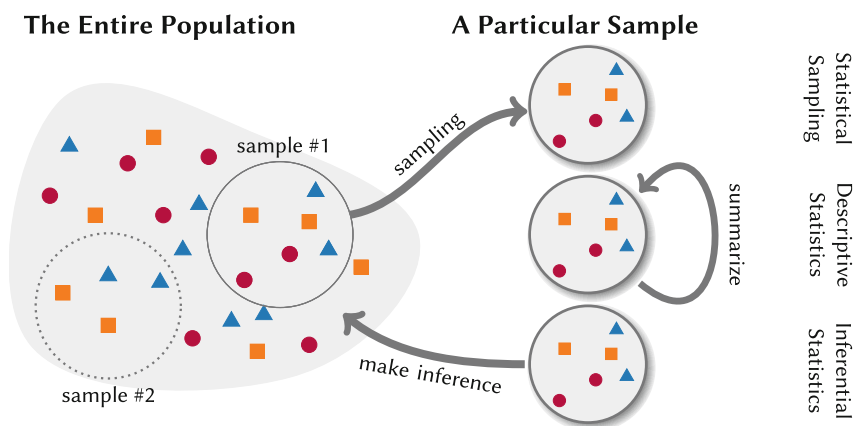


Fig. 8.1 Depending on if and how a sample from a population is used, different disciplines become relevant: *Probability theory* would try to mathematically derive how the population is composed and what its properties are. Instead, statistical methods operate on subsets of the population: (i) *statistical sampling* is responsible for the sampling itself; (ii) *descriptive statistics* characterizes properties of the sample, e.g., visually or by computing simple measures such as the mean; finally, (iii) *inferential statistics* takes one or more samples and makes predictions about properties of the entire population. The two circles on the left highlight two possible samples (see text for further details)

(there are nine rectangles, eight triangles, and eight circles, so we “guessed” only one rectangle wrong). A sample that is representative for the population is called a *representative sample*. However, if the population is not known, then it can be difficult to judge if a sample is truly representative or not—which is one of the fundamental challenges in statistics.

8.3.1 What is “Sampling”?

Therefore, an important subject in statistics is the process of *sampling*. It is important because the type of sampling, i.e., the way how we choose the data contained in the sample, may strongly influence the results. For example, if we would have used sample #2 in Fig. 8.1, we would have concluded (i) that the entire population consists of equal numbers of orange squares and blue triangles and (ii) that the population does not contain any red circles. If in a sample certain outcomes are over-represented it is said that the sample is *biased* toward that particular outcome (in our example the second sample would be biased toward triangles and rectangles). A famous example for such a kind of *selection bias* is given in Example 8.3.

Example 8.3 (Selection Bias)



“Selection bias” can come in different ways; one of these occurs when the particular sample represents some members of the population better than others, e.g., because there are many more of the one type than the other. This is called “undercoverage.”

A textbook example is the “*Literary Digest* voter poll,” which became famous for its disastrous prediction of the victory of the Republican Alfred Landon: from the poll data, it was predicted Alfred Landon would beat Franklin Roosevelt in the 1936 presidential election. Afterward, the commonly accepted explanation for the catastrophic failure was that poll participants were mainly taken from phone books and car registration lists. Back in 1936 only the better-off population (this time “population” really means “people”!) was able to afford cars and telephones—and those would have typically voted for the Republicans. Thus, the sample obtained during the voter poll suffered from an undercoverage of a big group of voters, who rather would have voted for a Democrat.

Nonetheless, there has been some discussion and investigation if this was in fact the true reason. Part of this discussion can be found in [1].

A good choice of the sample size together with a suitable “sampling strategy” can help to avoid erroneous approximations of the populations. Also in [ML](#) this will play an important role and a whole chapter is dedicated to the topic of “cross-validation”

(Chap. 16) which, among others, also helps to make sampling problems visible. We finish this section by some thoughts about the meaning of *sample* in the context of (real) experiments:

Words of Advice



Often, in Materials Science and Physics, the notion of *sample* and *specimen* are used interchangeably. However, once you have a bit of a statistics background, the word *sample* tends to have a specific connotation. A material sample then refers to a small amount of material that is taken from a much larger piece of material; the purpose of the sample is to represent details of the larger amount of material, i.e., by testing the sample, we can infer how this particular material would behave in general. As opposed to this, a specimen often refers to a *particular* individual, e.g., for which geometry details or boundary conditions are specified. Nonetheless, keep in mind that in many cases there is no commonly agreed definition of these terms. Luckily, often the context resolves any ambiguity.

Before we continue with an overview of different sampling strategies in the next section, here is a short summary of the four most important notions that we encountered here:

Things to Remember (Important Notions & Concepts from Statistics)



- *Descriptive Statistics vs Inferential Statistics*: The distinction between descriptive and inferential statistics is important because it determines the goal of “doing statistics”: descriptive statistics observes and describes, e.g., the shape of distributions, whereas the goal of inferential statistics is to establish a relation between the data and any underlying causality.
- *Sample vs Population*: These concepts are absolutely fundamental in statistics. A *sample* is a subset of the possibly very large dataset consisting of all data, the *population*. The sample has the goal to represent the relevant aspects of the population. At the same time, the sample should have a “convenient” size.

8.4 Sampling Strategies

Sampling is a fundamental concept of statistics and therefore, a number of strategies for sampling data were developed. The goal is always to create samples that are as representative as possible. In the context of ML we will encounter related tasks when we discuss *cross-validation* in Sect. 16.2. In the following a few of the most common strategies for sampling are introduced. The list of possibilities, however, is rather long, which is why we left out in particular techniques for highly specialized situations.

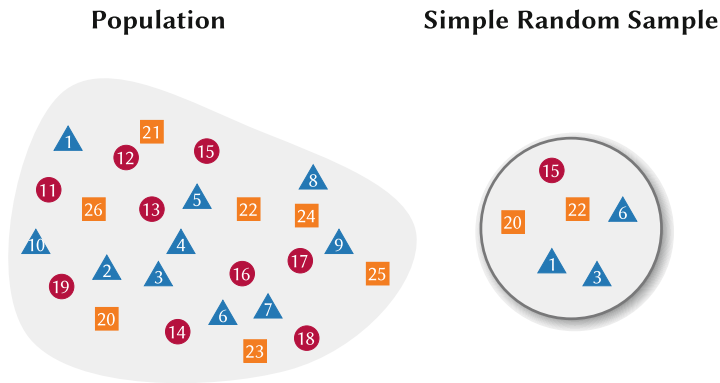


Fig. 8.2 Visualization of the simple random sampling strategy. Here, the numbers 6, 22, 1, 3, 20, and 15 were randomly drawn

8.4.1 Simple Random Sampling

The easiest approach for sampling is *simple random sampling* (SRS). It is a method that is based on assigning the same probability to each item that could be chosen during sampling. Then, the items contained in the sample are chosen without replacement (also see Chap. 6). Note that random sampling *with replacement* is possible as well but has slightly different properties; for larger populations, however, the difference becomes small. A visualization of this sampling technique is shown in Fig. 8.2.

Here is an example where we compute the mean value of a population and a random sample. We start by importing `numpy` and creating a population that consists of the numbers from 1 to 99:

```
In [1]: import numpy as np
        population = np.arange(1, 100)
        population.sum() / population.size
```

```
Out [1]: 50.0
```

Next, a random number generator `rng` is created. It comes with the convenient method `choice` that chooses from the given `population` a number of items (here: 10) without replacement:

```
In [2]: rng = np.random.default_rng()
        sample = rng.choice(population, size=10, replace=False)
        sample
```

```
Out [2]: array([21, 64, 38, 74, 19, 13, 77, 12, 58, 90])
```

Now, we can also compute the mean value of the sample

```
In [3]: sample.sum() / sample.size
```

```
Out [3]: 46.6
```

From this random sample, we estimate the population mean to be 46.6, which is close to the true value of 50.0. Keep in mind that typically we do not know the true value of the population.

If more than one sample is to be created, then each sample is chosen independently from the other samples, i.e., for each sample items are chosen from the full set of items. Thus, the same item can occur in more than one sample, but no more than once in a single sample. Thus, based on the above code, we could create three samples `s1`, `s2`, and `s3` by

```
In [4]: n = 10
s1 = rng.choice(population, size=n, replace=False)
s2 = rng.choice(population, size=n, replace=False)
s3 = rng.choice(population, size=n, replace=False)
print("mean values:", s1.sum() / n, s2.sum() / n, s3.sum() / n)
```

```
mean values: 33.6 58.7 62.2
```

where we observe that each sample is characterized by a different sample mean. In other words, we have now three different estimates of the population mean.

The advantage of this simple probabilistic sampling method is that it does not require any knowledge about the populations. On the other hand, this also implies that typically more data are required as compared to methods that are able to make use of knowledge about the distribution of data within the population.

8.4.2 Systematic Sampling

Systematic sampling or *linear sampling* is a very simple methodology where from an ordered list of N elements the first sample is chosen randomly. Subsequently, every k -th element is chosen. After the end of the list was reached, one can return to the beginning of the list. k is called *sampling interval* or *skip* and can be computed by

$$k = \frac{N}{n}, \quad (8.1)$$

where n is the prescribed sample size, and k should be rounded to the next larger integer.

As a consequence, every element in the list (i.e., the population) has the same probability to be picked. The probability is commonly called *epsem*. In this respect, system sampling is comparable to simple random sampling. One of the prerequisites is that the population needs to be homogeneous. Additionally, if the data contained

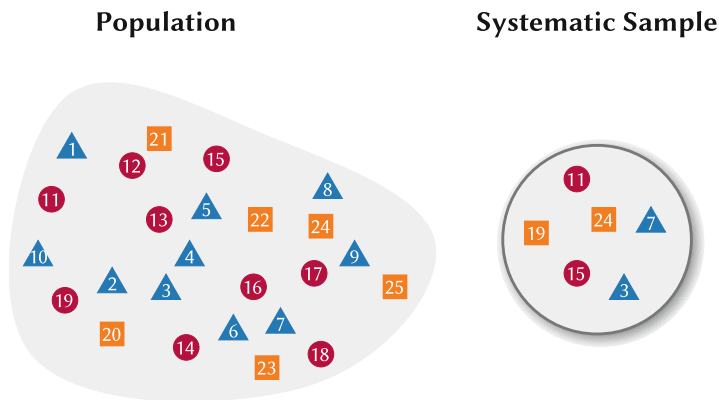


Fig. 8.3 Visualization of the systematic sampling strategy. The randomly chosen start item was number 3, and the sampling interval was 6

in the population follows a certain pattern, then systematic sampling might hide this pattern (e.g., imagine that every k -th materials science experiment was done by a certain person who is known as a particularly meticulous experimenter resulting in small errors. Then, always picking the experiment from *this* particular series might introduce a bias). Note that time series data often contains repeated patterns, which is why systematic sampling is in general not a good choice for such data. A visualization of this strategy is given in Fig. 8.3.

A small Python example is as follows: we import `numpy` and create a random number generator

```
In [1]: import numpy as np
        rng = np.random.default_rng()
```

Next, a population is created with 25 items (`range(N)`), to and we append a copy (note that `2 * list(...)` duplicates and concatenates the list):

```
In [2]: N = 25
        population = np.array(2 * list(range(N)))
        population

Out [2]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19, 20, 21, 22, 23, 24,  0,  1,  2,  3,  4,  5,  6,  7,  8,
                9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24])
```

Appending the copy is used, in case that we reached the end of the list. Because then we have to start from the beginning—there are more general ways of doing this, and our “appending” strategy only works as long as it is clear that we will not exceed the length of the two lists. We now can choose the sampling interval,

```
In [3]: n = 6  
        k = int(N / n)  
        k
```

```
Out [3]: 4
```

and pick elements starting from `start` every `k` elements. The second square bracket chooses only the first `n` elements.

```
In [4]: start = rng.integers(0, N)  
        population[start::k][:n]
```

```
Out [4]: array([24,  3,  7, 11, 15, 19])
```

8.4.3 “Take from Top” Sampling

As opposed to systematic sampling, *take from top* is a technique that is well suitable for time series data. It uses the first n elements from a population and thereby preserves the temporal structure of time series data. However, generally there is no guarantee that the first chunk of data is in fact statistically representative of the population.

8.4.4 Weighted Sampling

Sometimes, during data acquisition, certain observations are recorded with non-uniform probabilities or left out completely (“gaps”), leading to a biased dataset. Using simple random sampling then leads to skewed descriptions (e.g., of the mean), which does not properly approximate the population. Weighted sampling counterbalances this behavior by assigning different probabilities when items are chosen during sampling. Effectively, in a region with dense data points, the sampling probability then is lower as in sparsely populated regions. This should result in an unbiased sample.

As an example, assume that a dataset consists of data that corresponds to two distinct classes, and one class has significantly more data than the other class. As a strategy for compensating for this, e.g., *oversampling* duplicates samples from the class with the smaller amount of data, the *minority class*. Another approach is *undersampling*, which simply ignores some data records from the class with more data, the *majority class*. An example for oversampling is also given in Sect. 16.2.7 where a technique called *bootstrap* is introduced. Effectively, both over- and undersampling introduce a bias by their selection procedure that is supposed to counteract the (unwanted) bias contained in the dataset.

8.4.5 Stratified Sampling

Stratification denotes the process of splitting the whole population into “sub-populations,” the so-called *strata* (plural of *stratum*). The partitioning has to be non-overlapping, i.e., each element is exactly in one of the strata. Subsequently, simple random sampling (Sect. 8.4.1) can be employed, which has to be done for all strata.

The goal of the partitioning is that elements of each of the strata are as similar as possible (i.e., the variability within each stratum is minimized). At the same time, the mean of each stratum should be as different as possible (i.e., the variability *between* strata is maximized). Thus, stratified sampling is applicable in particular when data can be separated into different categories. The advantage is that such a stratified sample is more representative of the population than the sample obtained from simple random sampling. A visualization of the concept is shown in Fig. 8.4, and two examples are presented in Example 8.4. Note that for each individual stratum, different sampling techniques can be employed, e.g., one stratum can be sampled using simple random sampling and the other one using weighted sampling.

Example 8.4 (Stratified Sampling)

1. Stratified sampling is commonly used when strata can be formed based on geography since there the strata have a direct meaning: data from different cities or countries end up in different strata.
2. In materials science such a partitioning could make sense if a number of experiments are conducted, each of which results in a different number

(continued)

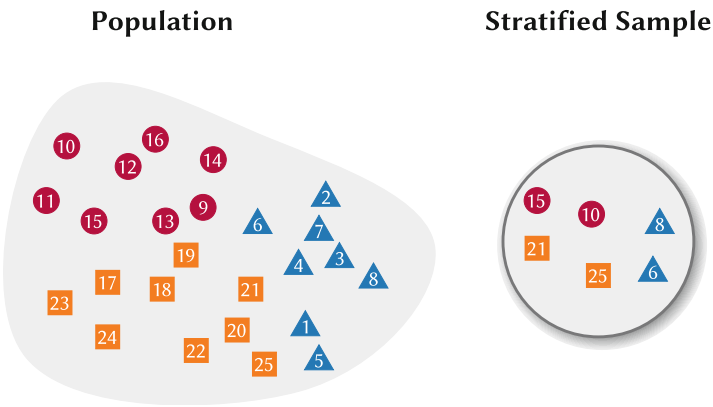


Fig. 8.4 Visualization of the stratified sampling strategy. From each of the three *strata*, a number of samples is drawn in proportion to the stratum size

of measurements, e.g., experiment A has 30 measurements of a yield stress, experiment B has 60 measurements, and experiment C has 120 measurements. Each experiment might be performed with a different machine and therefore affected by different errors. We could take 30 random samples from these 210 measurements, but there is the possibility that one of the experiments is by chance over-represented, introducing a bias. Considering each experiment as a stratum and randomly taking 10 elements from A, 20 from B, and 40 from C ensures that the sample contains data in proportion to the subpopulations A, B, and C.

However, in many cases it is not directly clear how to choose the criteria or variables for splitting the population into strata. Then stratified sampling might not be the method of choice. Additionally, the overall computational cost for implementing this method is significantly higher than implementing one of the simpler methods.

Here is a simple Python implementation that, in a simplified approach, randomly assigns the population data to strata. We again start by creating a population.

```
In [1]: import numpy as np
        rng = np.random.default_rng()

        N = 25
        population = np.arange(N)
        population
```

```
Out [1]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19, 20, 21, 22, 23, 24])
```

This is then split into four strata:

```
In [2]: strata = np.array_split(population, 4)
        strata
```

```
Out [2]: [array([0, 1, 2, 3, 4, 5, 6]),
          array([ 7,  8,  9, 10, 11, 12]),
          array([13, 14, 15, 16, 17, 18]),
          array([19, 20, 21, 22, 23, 24])]
```

In a compact approach, we create a list of all random samples from each stratum.

```
In [3]: s = [rng.choice(stratum, size=2, replace=False) for stratum in strata]
        sample = np.concatenate(s)
        sample
```

```
Out [3]: array([ 0,  3,  9,  8, 14, 18, 23, 22])
```


This “inline `for` loop” is called *list comprehension* and adds for each element of the `for` loop the result of `rng.choice` to the list.

8.5 The Law of Large and Truly Large Numbers

8.5.1 The Law of Large Numbers

The *law of large numbers* (LLN) is one of the important concepts in probability and statistics. It states that if a random experiment is repeated “sufficiently” many times and all results are averaged, we will obtain a result that is approximately the expected value. In Fig. 8.5 we can observe this during repeatedly rolling a die and monitoring how the deviation from the mean value changes as more and more results from die rolling are added. This is shown for altogether five runs with different seed values of the random number generator, giving five different sequences of (pseudo) random numbers shown as the colored lines in the right panel. The left panel shows the random values that correspond to the blue line.

The Python code for producing this figure is similar to that already shown in Sect. 5.2.2. It starts with importing the required modules and with initializing the random number generator:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
rng = np.random.default_rng(42)
```

Then, we obtain 1000 random values and use the cumulative summation to obtain the means for all lengths of random number sequences:

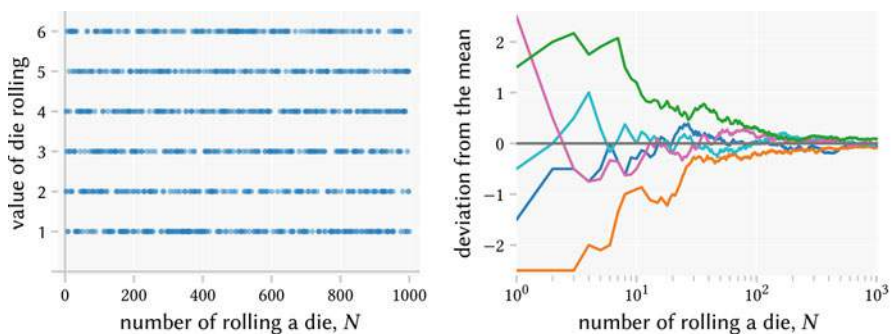


Fig. 8.5 The Law of Large Numbers during rolling of a die. Left: observed values as a function of the number of die rolling. Right: deviation of the mean value of all die rolls that occurred during the first N rolls from the expected value

```
In [2]: n_samples = 1000
        values = rng.integers(1, 6, endpoint=True, size=n_samples)
        sample = np.arange(1, n_samples + 1)
        averages = values.cumsum() / sample
```

Finally, we plot the deviations of the mean values from the expected value of 3.5 in a semilogarithmic plot. This scaling of the axis helps to reveal more details of the left region of the plot, where variations are very large:

```
In [3]: fig, ax = plt.subplots(ncols=2, figsize=(9, 3))
        ax[0].plot(sample, values, 'o', mfc='C0', mec='none', alpha=0.6)
        ax[1].plot(sample, averages - 3.5, '-')
```

Generally, we observe the behavior of “the larger the sample sizes, the smaller error”—if the opposite would be the case, then the concept of sampling would be obsolete.

We can also see that in all of the five sequences, some sample means overestimate the expected value, while others underestimate it, depending on the particular sequence of random numbers. There is no way to predict such details of the plot.

Words of Advice



Sometimes, people think that the Law of Large Numbers is also applicable in situations where the samples are small and that this helps to estimate the expected value. This is no correct, and there is no such thing as a “Law of Small Numbers.”

The law of large numbers is directly related to what will be introduced in Example 12.1 as *regression to the mean*. In the context of ML the LLN tells us that training data must be representative of the true population. This might require that some data ranges are explicitly excluded, e.g., during data cleaning, so that there are not too many different phenomena contained in the dataset and the sample mean

Example 8.5 (Diffusion and the Law of Large Numbers)



The LLN can be applied to coarse-graining of (idealized) diffusive processes. Imagine that we investigate how diffusion takes place, if one half of a container contains idealized atom(s), separated by a barrier from the other half, and the barrier is removed.

Then, the trajectory of a *single atom* follows a “random walk,” which exhibits strong fluctuations. If there would be *a few handfuls of atoms* in the one half of the container, we could observe that more and more atoms are moving into the other container until both halves contain very roughly speaking the same density of atoms. However, we would observe many fluctuations about an equilibrium state, and never would see a truly stationary

(continued)

state. If there were *incredibly many atoms*, such that we would not be able to see them individually any more, as, e.g., in a fluid in a centimeter-sized container, then we would only observe a continuous process that follows FICK's law and all processes would follow the "expected trajectory" without fluctuations (obviously, we have to assume that adding more atoms does not change the interactions or introduce new physical phenomena). The LLN guarantees that such a stable, stationary state exists.

8.5.2 The Law of Truly Large Numbers

An extension of the Law of Large Numbers is the *Law of Truly Large Numbers*. It states that if larger and larger samples of observations are investigated, the likelihood of seeing something unusual (e.g., a particularly small or large value, rare events, or even entirely implausible observations) increases.

Such unusual events have the property that we tend to notice and remember them much more than we would notice average events (e.g., people tend to remember the big lottery wins but never the many failed betting attempts). In extremely large samples, it is therefore paradoxically rather easy to find correlations. While a statistician would typically always agree that larger samples are better, it is important to beware that inference based on single events in very large samples can be very deceptive. *Cherry picking* of data, i.e., choosing only those data that fit to a hypothesis, is generally problematic. The law of truly large numbers warns us that this can also happen unintentionally.

8.6 Central Limit Theorem

Assume that a number of n random experiments are performed, for each of which the mean value $\bar{x}_i$ of the random variable X is recorded. Then, the *central limit theorem* (CLT) states that the shape of the distribution of the n values of the sample means $\bar{x}_i$ always approaches the shape of a Gaussian distribution.

As an example, consider an experiment of sampling integers from a uniform random distribution with values in between -10 and 10 , all of which have the same likelihood to appear. For these, we will compute the mean using a Python program. As a preparation we import the used Python modules. Additionally, we define the Gaussian distribution function that is given by the location `loc` and the standard deviation `s`:

```
In [1]: import numpy as np

def gaussian(x, loc, s):
    return (1 / np.sqrt(2 * np.pi * s ** 2) *
            np.exp(-(x - loc) ** 2 / (2 * s ** 2)))
```

Next, we define the size of a sample, for which the mean is to be computed, as well as the number of means to be calculated.

```
In [2]: sample_size = 10000 # larger values = narrower distributions
        n_means = 500      # larger values = more "Gaussian-like"
```

Now, we create a list of the mean values of all samples:

```
In [3]: means = []
        for i in range(n_means):
            sample = np.random.randint(-10, 11, size=sample_size)
            sample_mean = np.mean(sample)
            means.append()
```

which can then be visualized as a histogram

```
In [4]: import matplotlib.pyplot as plt

plt.hist(means, bins=30, density=True)
x = np.linspace(-0.4, 0.4, 200)
plt.plot(x, gaussian(x, 0, 0.06))
```

as shown in Fig. 8.6. We also superimposed a line plot of a Gaussian function for which we identified the best fitting value of the standard deviation manually. Figure 8.6 shows the results for two different numbers of mean values. It is quite remarkable and rather surprising how well the agreement with the normal distribution is. Note that the distribution from which the samples were taken was a uniform random distribution, i.e., all values within the given range occur with equal likelihoods. It is even more surprising as this behavior is in fact independent of the statistical distribution from which the samples are drawn—as long as trials are *independent and identically distributed* (IID). This is quite a useful theorem, and there are (at least) two direct applications:

- We can use the theorem to make inference about the distribution of the means of samples; we also have a way of estimating how reliable the size of the collected dataset might be.
- The CLT can be used to generate Gaussian distributed random numbers. All that is needed are random numbers sampled from *any* distribution. However, this strategy is relatively slow, as compared to other methods, such as the Box-Muller method.

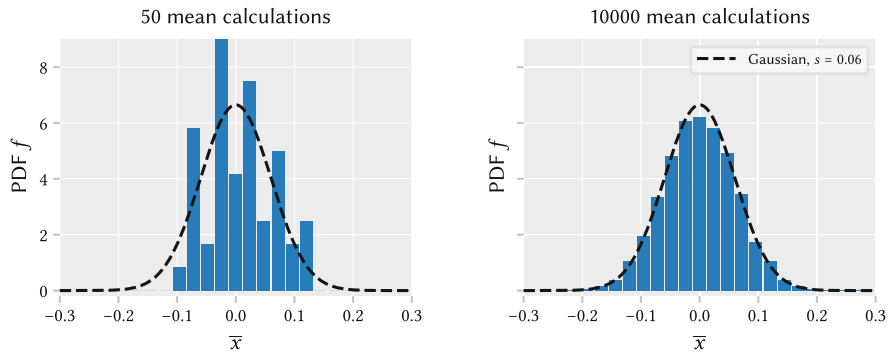


Fig. 8.6 Visualization of the consequence of the central limit theorem (see the text for further explanations)

Words of Advice



Sometimes, the central limit theorem is confused with the Law of Large Numbers. These are, however, two entirely different theorems: The Law of Large Numbers is intuitive and tells us that agglomerating more data helps to obtain more accurate estimates of the population. The central limit theorem, however, is not intuitive at all and also does not conclude anything about a *single* mean. It is a great tool for analyzing a larger set of sample means.

8.7 Relations Between Multivariate Variables: Covariance and Correlation

In many experiments and simulations, variables are used or recorded that are not entirely independent of each other. An example is the obvious relation between force and displacement during a tensile test: as the displacement increases, so does the force. Another example is the radius r of a circle and its circumference S : there exists a (in this case linear) relation between the two variables as the circumference is proportional to the radius. Thus, if the radius increases, so does S ; if r decreases also, S becomes smaller; the two variables have a positive relationship. However, dependencies between variables can also be “weak” dependencies where we even might not be aware of a clear mathematical relation between variables.

8.7.1 Covariance

Covariance is a measure for the degree of relationship between two variables and is related to *variance*: while variance is a measure for the spread of distribution, *covariance* is a measure of how strongly these two variables vary together. Here is the formal definition:

Definition 8.3 (Covariance) The (true) covariance is defined for two univariate random variables X and Y (each of which consists of N records) as the expected or mean value of the product of the random variable deviations from the respective expected values:

$$\text{Cov}(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] . \quad (8.2)$$

If the expected value $\mathbb{E}$ of the population is unknown (which is usually the case), then the sample means $\bar{x}$ and $\bar{y}$ are used as estimators

$$\text{Cov}(X, Y) = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{X})(y_i - \bar{Y}) . \quad (8.3)$$

which explains the scaling factor of $1/(N-1)$ resulting from *Bessel's correction*, cf. Sect. 9.3.4.

From this definition it is directly clear that the covariance is symmetric in its arguments such that $\text{Cov}(X, Y) = \text{Cov}(Y, X)$. Similar to the mean and the variance, also the covariance can be written in terms of linear algebra operations, as shown in Example 8.6. This approach is then generalized in Sect. 8.7.2 in form of the *covariance matrix*.

Example 8.6 (Covariance Using a Vectorized Formulation)



Equation (8.3) can also be rewritten using the vector product between the two column vectors $X = [x_1, x_2, \dots, x_N]^T$ and $Y = [y_1, y_2, \dots, y_N]^T$ each of which contains N records/measurements:

$$\text{Cov}(X, Y) = \frac{1}{n-1} (X - \bar{X})^T \cdot (Y - \bar{Y}) \quad (8.4)$$

Consider $X = [1, 2, 3]^T$ and $Y = [9, 7, 5]^T$. The mean values are $\bar{X} = 3$ and $\bar{Y} = 10.5$. Then, the covariance is computed as

$$\begin{aligned} \text{Cov}(X, Y) &= \frac{1}{3-1} ((1-2)(9-10.5) + (2-2)(7-10.5) \\ &\quad + (3-2)(5-10.5)) \\ &= \frac{1}{2} ((-1)(-1.5) - 5.5) = -2 \end{aligned}$$

(continued)

The vectorized Python implementation follows Example 7.4:

```
In [1]: import numpy as np
X, Y = np.array([1, 2, 3]), np.array([9, 7, 5])
mean_X = np.mean(X)
mean_Y = np.mean(Y)
covXY = 1 / (len(X) - 1) * (X - mean_X).dot(Y - mean_Y)
covXY
```

```
Out [1]: -2
```

8.7.2 Covariance Matrix

Often, the situation occurs where we need to investigate the correlation of more than two random variables. In those cases, a compact mathematical form can be obtained when the individual *random variable* (RVs) are represented in form of a feature matrix. The feature matrix $\mathbf{X}$ represents a multivariate random variable, which is given by the following column vectors, joined into a matrix (cf. Eq. (3.11)),

$$\mathbf{X} = \begin{bmatrix} | & & | \\ X_1 & \cdots & X_n \\ | & & | \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} \quad (8.5)$$

where each of the X_i is a random variable and consists of m records.

Recall that the covariance of two variables is a scalar value. Now, we have altogether n RVs. The *covariance matrix* consists of the covariance values evaluated for each pair $\{X_i, X_j\}$, Eq. (8.3), summarized in form of a matrix:

$$\text{Cov}(\mathbf{X}) = \begin{bmatrix} \text{Cov}(X_1, X_1) & \text{Cov}(X_1, X_2) & \cdots & \text{Cov}(X_1, X_n) \\ \text{Cov}(X_2, X_1) & \text{Cov}(X_2, X_2) & \cdots & \text{Cov}(X_2, X_n) \\ \vdots & & \ddots & \vdots \\ \text{Cov}(X_m, X_1) & \text{Cov}(X_m, X_2) & \cdots & \text{Cov}(X_m, X_n) \end{bmatrix}. \quad (8.6)$$

Thus, the covariance matrix is a compact representation of the correlations between *all* variables. In what follows we derive a formulation that can easily be applied to the whole feature matrix using only a computationally efficient and mathematically compact matrix-matrix multiplication.

Step-by-Step Derivation of the Structure of the Covariance Matrix $\mathbf{S}$

A number of later chapters such as the “principal component analysis” in Sect. 15.2 will require the covariance matrix and will be using it as part of more complex formulations. Since this is the first time that we are calculating with the feature matrix, we will do it in more detail than usual.

As before, we have to differentiate between the population covariance matrix, abbreviated by $\mathbf{\Sigma}$, and the sample (or data) covariance matrix, abbreviated by $\mathbf{S}$. $\mathbf{S}$ serves as an estimator for the population covariance matrix $\mathbf{\Sigma}$. The following derivations will be written only for the sample covariance matrix, but they hold for the population covariance matrix as well, if the factor from Bessel’s correction is adapted.

The first step according to Eq. (8.3) requires that the data of each feature needs to be centered around the mean value. For this, the sample mean of each variable X_j is computed by

$$\bar{X}_j = \frac{1}{m} \sum_{i=1}^m x_{ij} . \quad (8.7)$$

These values are then subtracted from the records of each feature, resulting in the *centered data matrix* $\bar{\mathbf{X}}$:

$$\bar{\mathbf{X}} = \begin{bmatrix} x_{11} - \bar{X}_1 & x_{12} - \bar{X}_2 & \cdots & x_{1n} - \bar{X}_n \\ x_{21} - \bar{X}_1 & x_{22} - \bar{X}_2 & \cdots & x_{2n} - \bar{X}_n \\ \vdots & \vdots & & \vdots \\ x_{m1} - \bar{X}_1 & x_{m2} - \bar{X}_2 & \cdots & x_{mn} - \bar{X}_n \end{bmatrix} = \begin{bmatrix} \bar{x}_{11} & \bar{x}_{12} & \bar{x}_{13} & \cdots & \bar{x}_{1n} \\ \bar{x}_{21} & \bar{x}_{22} & \bar{x}_{23} & \cdots & \bar{x}_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ \bar{x}_{m1} & \bar{x}_{m2} & \bar{x}_{m3} & \cdots & \bar{x}_{mn} \end{bmatrix} , \quad (8.8)$$

where in the second matrix we introduced the abbreviation $\bar{x}_{ij} := x_{ij} - \bar{X}_j$. With this we can now compute the sample covariance matrix $\mathbf{S}$ as

$$\mathbf{S} = \frac{1}{n-1} \bar{\mathbf{X}}^T \bar{\mathbf{X}} \quad (8.9)$$

which is a quadratic $n \times n$ matrix that additionally is also symmetric (cf. Appendix A.2.12). To elucidate the structure of this product and the correspondence to Eq. (8.3), we explicitly write down the elements of the matrix product as shown in Fig. 8.7 and then compute two particular elements of $\mathbf{S}$. The first one is on the main diagonal (highlighted in blue in Fig. 8.7):

$$s_{11} = \frac{1}{n-1} \left(\bar{x}_{11}^2 + \bar{x}_{21}^2 + \cdots + \bar{x}_{m1}^2 \right) . \quad (8.10)$$

Observe that s_{11} is the variance of the first variable or feature. Each other element s_{ii} on the main diagonal of $\mathbf{S}$ gives the variance of the i -th feature. Next, we take a

$$\mathbf{S} = \frac{1}{n-1} \begin{bmatrix} \bar{x}_{11} & \bar{x}_{21} & \bar{x}_{31} & \cdots & \bar{x}_{m1} \\ \bar{x}_{12} & \bar{x}_{22} & \bar{x}_{32} & \cdots & \bar{x}_{m2} \\ \bar{x}_{13} & \bar{x}_{23} & \bar{x}_{33} & \cdots & \bar{x}_{m3} \\ \vdots & \vdots & \vdots & & \vdots \\ \bar{x}_{1n} & \bar{x}_{2n} & \bar{x}_{3n} & \cdots & \bar{x}_{mn} \end{bmatrix} \begin{bmatrix} \bar{x}_{11} & \bar{x}_{12} & \bar{x}_{13} & \cdots & \bar{x}_{1n} \\ \bar{x}_{21} & \bar{x}_{22} & \bar{x}_{23} & \cdots & \bar{x}_{2n} \\ \bar{x}_{31} & \bar{x}_{32} & \bar{x}_{33} & \cdots & \bar{x}_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ \bar{x}_{m1} & \bar{x}_{m2} & \bar{x}_{m3} & \cdots & \bar{x}_{mn} \end{bmatrix}$$

Fig. 8.7 Matrix product for computing the covariance matrix. Two examples of involved rows and columns are highlighted in the two different colors

look at an element that is off the diagonal. The corresponding rows and columns are highlighted in light orange color in Fig. 8.7:

$$s_{23} = \frac{1}{n-1} (\bar{x}_{12}\bar{x}_{13} + \bar{x}_{22}\bar{x}_{23} + \cdots + \bar{x}_{m2}\bar{x}_{m3}) \quad (8.11)$$

s_{23} is the covariance of variables X_2 and X_3 , or in general, s_{ij} is the covariance of variable X_i and variable X_j . For $i = j$, we obtain the elements on the main diagonal,

$$s_{jj} = \frac{1}{n-1} (\bar{x}_{1j}^2 + \bar{x}_{2j}^2 + \cdots + \bar{x}_{mj}^2) \quad (8.12)$$

$$= \frac{1}{n-1} \sum_{i=1}^n (x_{ij} - \bar{X}_j)^2 \quad (8.13)$$

$$= \text{Var}[X_j], \quad (8.14)$$

each of which is identical to the variance of the RV X_j . Examples of the resulting covariance matrices for different datasets are discussed below in Example 8.7.

Where Is the Covariance Matrix Typically Used?



In later chapters (e.g., Chap. 12), we will “fit” a *regression line* to given data and identify the covariance matrix as one of the main ingredients for finding this line in a closed form solution.

The covariance is a measure for how strong the linear relationship between two variables is. In the context of *principal component analysis* (PCA) in Sect. 15.2, this is used to estimate the redundancy between variables: if a variable X_2 is a linear function of a variable X_1 , then it does not add new information to a dataset and is said to be *redundant*. Thus, a small value

(continued)

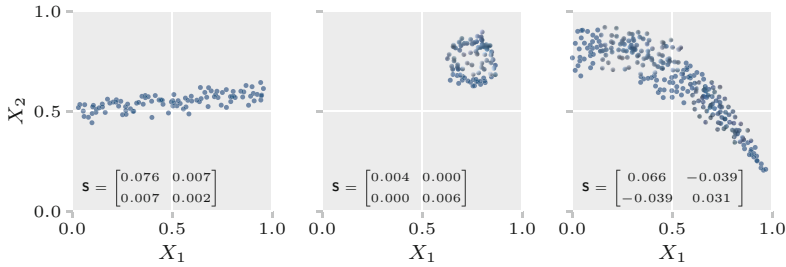


Fig. 8.8 Three different datasets together with the resulting covariance matrix S

of covariance implies a low redundancy, while a high value signifies a high degree of redundancy.

Furthermore, it is the main constituent of the *correlation matrix*, which will be introduced next.

Python Implementation and Example

A Python implementation is straightforward, thanks to the convenient form of the equation derived in Eq. (8.9). The following code lines demonstrate this for an artificially created dataset. We start by importing the `numpy` module and define the number of records:

```
In [1]: import numpy as np
        rng = np.random.default_rng() # create a random number generator
        n = 100                       # number of records
```

Next, we create values for X_1 and randomly “jitter” them a bit. X_2 is a linear function of X_1 with $X_2 = 0.5 + 0.1X_1$. *RVs* are superimposed with some noise. To create the data matrix $\mathbf{X}$ we stack the two row vectors on top of each other and transpose the resulting matrix such that each feature is a column:

```
In [2]: X1 = np.linspace(0.03, 0.97, num=n) + rng.normal(loc=0, scale=0.01, size=n)
        X2 = 0.5 + 0.1 * X1 + rng.normal(loc=0, scale=0.03, size=n)
        X = np.vstack((X1, X2)).T
```

This dataset is shown in the left panel of Fig. 8.8. $\bar{\mathbf{X}}$ in Eq. (8.9) denotes the *centered* data matrix that is obtained by subtracting the mean of X_1 and X_2 from the two columns of $\mathbf{X}$:

```
In [3]: Xc = X - np.mean(X, axis=0)
```

where `axis=0` indicates that the mean along the first axis is to be computed, such that the result is a `numpy` array with two elements, the mean of X_1 and the mean of

X_2 . Now we can write down Eq. (8.9) as a Python expression where `@` represents the matrix-matrix multiplication of two numpy arrays, cf. Appendix A.2.4.

```
In [4]: 1 / (n - 1) * (Xc.T @ Xc)

Out [4]: array([[807.54640804,  80.78520835],
                [ 80.78520835,   8.08245467]])
```

A shortcut exists in form of the numpy function `cov`, which also takes care of centering the data. Note, however, that this function expects that features are contained in rows and not in columns. This can be changed by setting the parameter `rowvar=False`:

```
In [5]: np.cov(X, rowvar=False)

Out [5]: array([[807.54640804,  80.78520835],
                [ 80.78520835,   8.08245467]])
```

giving an identical result as for the “manual” calculation.

We are now able to compute the covariance matrix for arbitrary datasets. Example 8.7 shows three examples and discusses the resulting covariance matrices. Furthermore, these examples elucidate that the covariance represents *average* features of a dataset: visually we can easily detect a predominant orientation of the data, and this orientation determines then the relation between two variables on average.

Example 8.7 (Interpretation of the Covariance Matrix Values)



In Fig. 8.8, three different datasets are shown along with the corresponding covariance matrices.

The left dataset has a large spread along the X_1 direction and a small spread along X_2 which results in $s_{11} \ll s_{22}$ because the diagonal elements represent the variance of X_1 and X_2 , respectively. The value s_{12} is very small as there is only a very small dependency between the two variables (when X_1 changes, X_2 changes only slightly). $s_{12} > 0$ implies a positive relation between the two variables: if X_1 increase, then also X_2 increases.

In the middle dataset, most of the points are “clustered” around the position (0.75, 0.75). This explains why the two off-diagonal covariance values are approximately zero: there is no other relation between the two variables than that of the concentration around the point. As the spread of the cluster is approximately the same for the X_1 and X_2 directions, the values of the variances are also approximately the same, $s_{11} \approx s_{22}$. The spread is roughly five times as large as the spread of X_2 in the left dataset which is also consistent with the two values of s_{22} .

(continued)

The rightmost dataset has a strong spread into both directions, which shows in the appreciable values of both s_{11} and s_{22} . The latter is about half of the values of s_{11} because the range of the values projected on the X_2 direction is much smaller than the range of the values projected on the X_1 direction. Here, s_{12} has a large value where $s_{12} < 0$ implies a negative relation between the two variables: if X_1 increase, then X_2 decreases (on average).

8.7.3 Correlation

A main difficulty with interpreting the covariance is that the value does not exactly tell us how strong the dependency between two variables is. For example, a value of 350 does indicate a positive correlation. However, for comparing the “strength of correlation” between two pairs of RVs, a comparison of the covariance value is not meaningful. The reason is that the one pair of variables might be scaled differently or using different units both of which will affect the mean values that are contained in the calculation of the covariance. The remedy to that problem is the concept of *correlation*.

To be more precise, the remedy consists in dividing the covariance by the standard deviations of *both* variables, i.e., to normalize the covariance. This results in the correlation coefficient:

Definition 8.4 (Correlation Coefficient) Pearson’s correlation coefficient of the random variables X and Y is given by the normalized covariance

$$\text{Cor}(X, Y) \equiv \rho_{X,Y} = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y} = \frac{\text{E}[(X - \text{E}[X])(Y - \text{E}[Y])]}{\sigma_X \sigma_Y} \quad (8.15)$$

The population correlation coefficient is typically denoted by ρ (or explicitly: ρ_{XY}) whereas the sample correlation coefficient is denoted by r (or more explicitly by r_{XY}) and is written in terms of the n records of X and Y :

$$r_{X,Y} = \frac{\sum_{i=1}^n (x_i - \bar{X})(y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{Y})^2}} \quad (8.16)$$

As compared to the covariance, the correlation coefficient is advantageous as its value range is limited to $[-1, 1]$. The reason is the normalization: the covariance

is always smaller than the product of the two standard deviations. Then, +1 and -1 denote the strongest (positive or negative) linear correlation, while a value close to zero denotes the strongest disagreement. Additionally, the normalization also removes the units from the covariance and makes the correlation coefficient a dimensionless measure. The correlation coefficient is not affected by shifting the data along one direction or by scaling of the variables.

For an interpretation of the correlation coefficient, we first take a look at the extreme cases: a value of 1 implies a linear relationship between the two variables and all points would be located on a line with positive inclination such that $y = x$. For -1 the line would have a negative inclination and we have $y = -x$. If the correlation coefficient is 0, then there is no linear relationship between the two variables. In turn, this also implies that if two variables are related to through a purely nonlinear relation, then the correlation coefficient will be zero. Figure 8.9 shows several examples for RVs with linear and nonlinear relationships together with the respective correlation coefficients and cross-covariance value. Comparing the first two columns, it becomes clear that the change of scale for X has a tremendous impact on the covariance, while the correlation is unaffected. Additionally, we clearly observe that a negative (or positive) slope results in a negative (or positive) correlation coefficient. The less noise is “fogging” the linear function the higher the correlation coefficient; the left and middle figure in the top row approaches already the limits of ± 1 . Finally, the rightmost column shows data that are nonlinearly related. For those, the correlation coefficient is nearly zero.

Interpretation of Pearson’s Correlation Coefficient

Quite conveniently, the magnitude of the correlation coefficient, ρ or r , is dimensionless and can be directly interpreted. As a rule of thumb, the following value ranges can be differentiated:

?

coefficient value:	-1	-0.8	-0.5	0	+0.5	+0.8	+1
type and strength of correlation:	perfect negative corr.	strong negative corr.	moderate negative corr.	variables uncorrelated	moderate positive corr.	strong positive corr.	perfect positive corr.

Correlation and Causation

There is a well-known phrase “*correlation does not imply causation*,” which is a warning about how to interpret correlations, e.g., governed by PEARSON’s correlation coefficient. It implies that just because two variables *seem* to be correlated, this does not mean that there is indeed an underlying relationship where the change of one variable is caused by a change in the other variable.

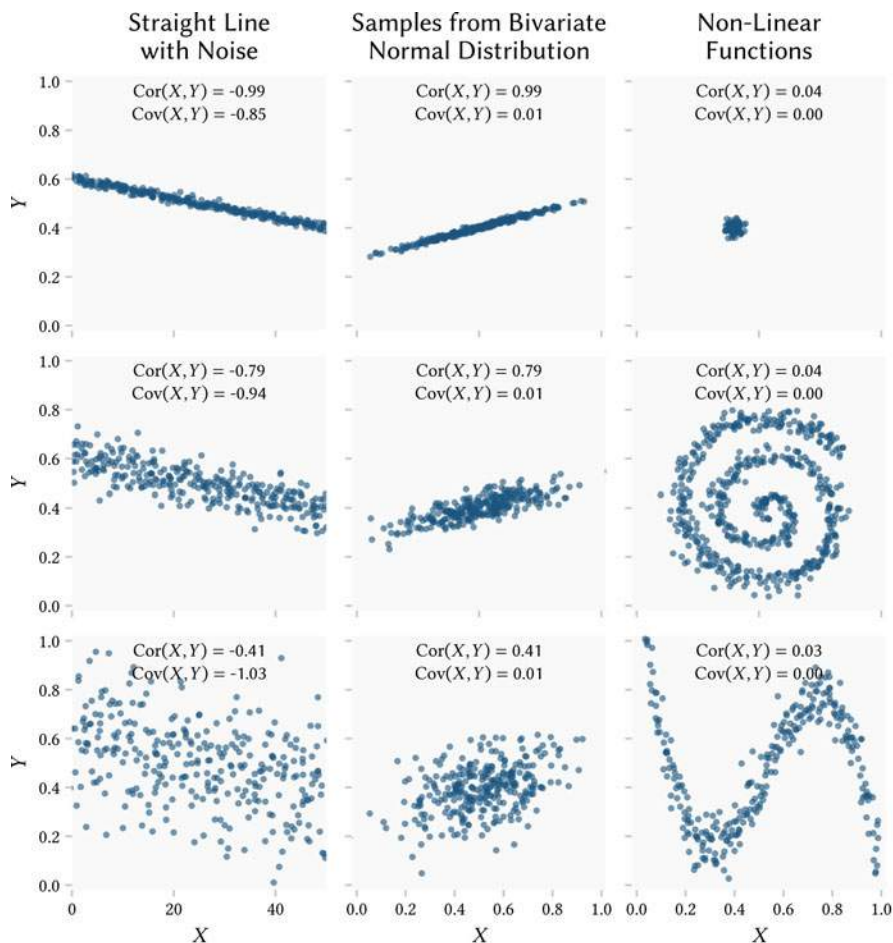


Fig. 8.9 Correlation coefficients and covariance values for different data distributions of two random variables. The left and middle column shows data with increasing randomness from the top to the bottom; note the different ranges of the X values that strongly influence the covariance value. The right column shows that the correlation coefficient is insensitive w.r.t. nonlinear relationships of the two variables, hence the (nearly) zero value for $\text{Cor}(X, Y)$

For example, consider the oscillation of a pendulum with mass m . We observe that the frequency ν reduces as m increases—there is a clear causality which we know from physics. A different situation is if we collect data about the average amount of steel production and the number of sold video games per year. Then, we would find that both are increasing over the years resulting in a positive correlation. Obviously, this does not imply that buying more video games increases the steel production. As there is correlation without causation, this type of correlation is called *spurious correlation*. In this case, there might be a third variable that was

unknown and the links of the two variables: the increasing size of the population on the earth (this time, population *does* imply people) results in a higher demand for metallic components and also lets the absolute number of game buyers increase. This “hidden” variable is also called *confounding variable*. Furthermore, spurious correlation can also occur by chance (e.g., due to a too small sample size).

What are the options for avoiding spurious correlations? The importance of a solid physical intuition about the problem at hand should not be underestimated. Ensuring that the sample is statistically representative, e.g., by using suitable sampling strategies together with a careful design of the experiment are other important strategies.

8.7.4 Correlation Matrix

For multivariate **RVs** the correlation coefficients can also be represented in form of a *correlation matrix*, in analogy to the covariance matrix. The normalization of the correlation coefficients has the convenient effect that all values are in the range of $[-1, 1]$ and can therefore easily be visualized. Example 8.8 shows a Python implementation and interprets the correlation matrix for the “DS-1 (Iris Flowers, Sect. 4.7)” dataset.

Words of Advice



An often encountered confusion arises in the case of a value of zero for the correlation coefficient. This *does not* mean that two variables are not related *at all*. It rather indicates that there is *no linear* relationship between them.

Example 8.8 (Matrix of Correlation Coefficients for the Iris Dataset) ✖

The Iris dataset has four features that might be correlated with each other. As a small Python example, we compute the covariance matrix and plot a visualization of the values. For this, we start by importing the Iris dataset (see Python Listing 9.1 on page 181 for more details).

```
In [1]: # This function is a just placeholder for reading the dataset and
# creating the feature matrix with the four features in columns.
X, y = import_iris_dataset()
print(f"X has {X.shape[0]} records and {X.shape[1]} features.")
```

X has 150 records and 4 features.

For computing the matrix of correlation coefficients, we use the numpy function `corrcoef` which, in analogy to the covariance matrix, takes the feature matrix and needs to be informed that the features are in columns:

(continued)

```
In [2]: import numpy as np
cor = np.corrcoef(X, rowvar=False)
cor

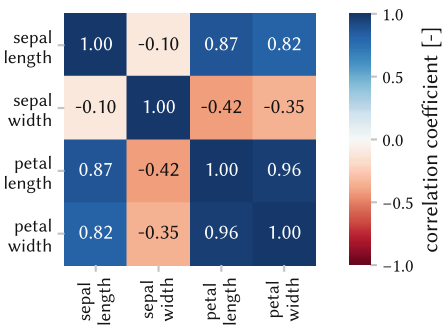
Out [2]: array([[ 1.          , -0.10378415,  0.87128294,  0.81697087],
 [ 0.87128294, -0.41521773,  1.          ,  0.9623143 ],
 [ 0.81697087, -0.35073314,  0.9623143 ,  1.          ]],
 [-0.10378415,  1.          , -0.41521773, -0.35073314],
```

The matrix can now be visualized, e.g., using `imshow` :

```
In [3]: import matplotlib.pyplot as plt
im = plt.imshow(cor, vmin=-1, vmax=1)
plt.colorbar(im)
```

Figure 8.10 shows a visually enhanced version of this plot. There, only half of the symmetric matrix is shown in order to avoid redundancy. The strong positive correlation between petal length and width implies that the longer the petal, the broader they are. On the other hand, a larger petal width sometimes implies a smaller sepal width; however, this relationship is rather moderate. Sepal width and sepal length are (almost) not correlated, i.e., if the length changes, this has, on average, either no effect on the width *or* the two features are nonlinearly related.

Fig. 8.10 Matrix of correlation coefficients for the iris dataset



8.8 Exercises

8.1 (Variance) Show that the following relationship for a random variable X holds:

$$\text{Var}[X] = E[X^2] - (E[X])^2 \quad (8.17)$$

8.2 (Correlation) Assume that for two random variables, the correlation coefficient is zero. Does this imply that there is no relationship between the two variables at all?

8.3 (Correlation) What are the correlation coefficients for data that obey the following functions: (a) $X_2 = 4 - 0.1X_1$, (b) $X_2 = X_1^2 + 1$, (c) $X_2 = 527 \cdot \log(X_1^2 + 10^3)$

8.4 (Correlation) Compute the covariance matrix and Pearson's correlation coefficient for the following random variables:

$$X = [-2, -1, 0, 1, 2] \quad (8.18)$$

$$Y = [5, 2, 4, 1, 3] \quad (8.19)$$

8.5 (Random Sampling) Calculate the value of π using random sampling. An approach to estimate the value of π is randomly sampling points inside a square of length 1 ($r = 1$) within which a quarter circle is inscribed.

1. Device a method to estimate the value of π by comparing the area of the square and circle.
2. Write a Python code to sample points randomly and calculate the value of π .
3. How many data points are needed to accurately estimate the value up to 4 decimal points.
4. Is there a strategy to improve the sampling, for example, using smaller number of data points, but performing the experiment multiple times?



8.6 (Sampling Techniques) Given below is the data from molecular dynamics simulation of Al using an EAM potential. The instantaneous temperature measured

(by applying a thermostat) at 0 pressure is given in K, and the temperature is recorded in intervals of 0.1 picoseconds.

T (K) = 600, 232, 280, 215, 429, 358, 340, 271, 219, 286, 258, 360, 298, 316, 254, 287, 250, 325, 262, 363, 312.

1. What is the mean temperature of the simulation?
2. Compare different sampling techniques:
 - a. Use “take from top” sampling and choose the first ten values and measure the average temperature.
 - b. Use systematic sampling with $n = 10$ samples and measure the average.

Reference

1. D. Lusinchi. “President” Landon and the 1936 Literary Digest Poll: Were automobile and telephone owners to blame? *Social Science History*, 36(1): 23–54, 2012. DOI <https://doi.org/10.1017/S014555320001035X>.

Exploratory data analysis is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those we believe to be there.

John Tukey (1915–2000)

American mathematician and statistician

9.1 The Why, the When, and the How

Assume that a colleague did an experiment for you and, after she was finished, provided you with a data file. Obviously, you are curious to see what is in there and how the experiment went. This is exactly the point of departure for any *exploratory data analysis* (EDA).

The next two sections start with a short introduction of techniques for performing an initial exploration of the data. The purpose is to ensure that we actually read the file content correctly and that we mainly know what we are looking at. While in a way trivial, these steps are crucial for avoiding many difficult to detect follow-up mistakes. These efforts are in a way also “explorative” steps but are not always considered as part of EDA.

The subsequent steps are then always considered part of any EDA: we analyze the data with the purpose to describe and characterize the whole dataset in the spirit of an initial inspection of the data. Among others, we are interested in finding out if there are any patterns, e.g., is the data clustered around a certain point? Are there correlations between variables? This can be either done purely descriptive, i.e., using mathematical measures such as the mean or the variance that could be then summarized, e.g., in form of a table. Or it can be done through visualization of the data itself where the challenge is to reduce the typically multi- and often even high-dimensional data to simpler representations that can be plotted in a two- or three-dimensional figure. Both approaches will be introduced below for univariate

and multivariate distributions in Sects. 9.3 and 9.4. These two methods meet when the mathematical measures are visualized themselves (e.g., plotting the distribution of mean values for various samples).

After completing the EDA, we know what the state of the dataset is: we know about the size, the “data coverage” of regions of interest, we might know about outliers, missing data, and other errors. However, EDA may go beyond characterizing distributions by simple mathematical measures that are typically found within descriptive statistics: it may even employ advanced methods such as clustering of data, dimensionality reduction, or multivariate regression analysis. Many of those methods will be introduced in later chapter in the context of *machine learning* (ML) (even though there will be no particular focus on EDA); for now we focus on the “classical” EDA approaches such as initial data exploration, descriptive statistics, and data visualization, each of which are performed prior to any further ML analysis. With this, we are then able to make a decision if, e.g., more or different datasets are required and which data analysis or ML approach might be the most suitable one for further investigating of the dataset.

9.2 Two Preliminary Steps

Before the “real” EDA starts, it is very useful to do a quick screening of the files and data and to ensure that everything is ready for the subsequent exploratory data analysis.

9.2.1 Initial Exploration of Data File(s)

The very first step, that is often not mentioned because it is not a statistical data analysis step, is a technical one: it aims at trying to understand the file format, the character encoding of the file content, or the structure of the data in the file and to see if the dataset seems to be in a good state. The easiest but nonetheless very effective way in doing this is to simply “look at the files and the data.” Figure 9.1 shows an extract from the DS-1 (Iris Flowers, Sect. 4.7) dataset. Even though without any additional information we do not know what the numbers mean, we still can observe a number of important details. First of all, we observe, that this is a text file, that we can read with a text editor; thus the data exists in form of ASCII characters and is not binary data. Next we see that each line consists of four floating point numbers with only one decimal place, followed by some text string (without spaces between the words—this might be important if we are to split the row content into individual items). Furthermore, numbers and the string are separated by commas; each row has the same structure of four numbers and a string. Altogether, there are only three different text strings. We don’t observe any unusual characters or “strange symbols” which might indicate an unusual character encoding.

In this case, we most likely already know from additional documentation that the first four numbers are the four attributes in centimeter (sepal length and width, petal

Fig. 9.1 Extract from a file containing the data of the famous “Iris Dataset”: the “==//==” denotes a number of rows that were skipped for this figure. The gray column are line numbers shown by the used text editor and are not part of the file itself

```
1 5.1,3.5,1.4,0.2,Iris-setosa
2 4.9,3.0,1.4,0.2,Iris-setosa
3 4.7,3.2,1.3,0.2,Iris-setosa
4 4.6,3.1,1.5,0.2,Iris-setosa
==//==
6 7.0,3.2,4.7,1.4,Iris-versicolor
7 6.4,3.2,4.5,1.5,Iris-versicolor
8 6.9,3.1,4.9,1.5,Iris-versicolor
9 5.5,2.3,4.0,1.3,Iris-versicolor
==//==
11 6.3,3.3,6.0,2.5,Iris-virginica
12 5.8,2.7,5.1,1.9,Iris-virginica
13 7.1,3.0,5.9,2.1,Iris-virginica
14 6.3,2.9,5.6,1.8,Iris-virginica
```

length and width) followed by the “class” (*Iris setosa*, *Iris versicolor*, *Iris virginica*). Nonetheless, through this quick look at the file content (i), we could ensure that the file is in a good state, and (ii) we have an idea how the data should be imported. Python Listing 9.1 shows an example how this could be done using `numpy`. Then,

```
1 import numpy as np
2
3 def import_iris_dataset():
4     """Read the file `iris.csv` and prepare data.
5
6     Returns the feature matrix X and target vector y.
7     """
8     data = np.loadtxt('iris.csv', dtype='str', delimiter=',')
9     X = np.array(data[:, :4], dtype=float)
10    y = np.array(data[:, 4], dtype=str)
11    return X, y
```

Python Listing 9.1 A Python function for reading the file `iris.csv`. The function returns the feature matrix and the target vector. Note the specification of the `dtype` when the two `np.array`s are created. Due to the mixed data type (numbers and strings) in the file `iris.csv`, we import all of it as strings and then create `X` as a `float` and `y` as a `string`. Section 4.7 provides more information on how to obtain the file

this function can be used to obtain the data in form of two `numpy` arrays. Next we can then print some basic information, such as the number of rows and columns:

```
In [1]: X, y = import_iris_dataset()
        print(f"X has {X.shape[0]} rows and {X.shape[1]} columns.")

X has 150 rows and 4 columns.
```

This can be quite useful information: we see that there are 150 data records of the 4 features—if we would have expected more (or less) records, then we know that something must have gone wrong during reading the data file. We then can also take a look at, e.g., the first four rows by

```
In [2]: X[:4]
```

```
Out [2]: [[5.1 3.5 1.4 0.2]
          [4.9 3. 1.4 0.2]
          [4.7 3.2 1.3 0.2]
          [4.6 3.1 1.5 0.2]]
```

Comparing this to the beginning of the data file is a good “sanity check” which helps to ensure that the importing of the file data went well.

Words of advice



Better check twice that your `numpy` variables indeed contain the data from the file that was imported. Sometimes, due to some peculiarities of the data representation in the file, you might have missed the first line, the delimiter that separates columns has a uncommon format resulting in “merging” of numbers, or the feature matrix could have been transposed. “Looking at the data” in the file and the `numpy` variables helps to avoid these problems.

As always in Python, there is not just one way of importing data files, there are in fact many of them. An alternative to using `numpy` is `pandas`¹ [3] which is one of the Python packages of the *Scientific Python Stack* and offers much flexibility for importing tabular data.

Last but not least, if you find in this initial investigation that the file does not contain tabular data but unstructured data, then you most probably have to use different ways for importing and storing the data.

9.2.2 A Quick Initial Visualization

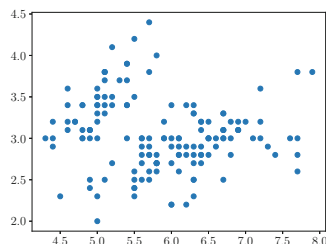
This initial visualization is only a first glimpse at the data through some basic plots. This could be, e.g., a line plot of, e.g., the total displacement vs. time, a histogram of a temperature distribution, or a scatter plot of two features of the above “Iris” example:

```
In [3]: import matplotlib.pyplot as plt
        plt.scatter(X[:,0], X[:,1])
```

There, `X[:,0]` and `X[:,1]` give a vector with the data of the first and second feature (sepal length and width), respectively. This results then in the plot shown in Fig. 9.2. The figure could be improved in several ways to reveal more or different information; however, this is not the purpose of the initial visualization, and we postpone all more sophisticated visual analysis to Sect. 9.4.

¹ <https://pandas.pydata.org/>.

Fig. 9.2 Scatter plot of the first two feature of the “Iris dataset.” Obviously, this plot lacks labels and might not be well formatted, but it serves the purpose of a quick initial visualization with just one line of Python code



As this scatter plot is a two-dimensional projection of the dataset into the plane spanned by the first two features, it cannot visualize the structure of the *whole* dataset, e.g., we do not know if the data points are distributed homogeneously along the other two directions (or, rather, features) or not. However, this plot already might be good enough to get a first impression of at least an excerpt from the data and to visually make sure that the dataset is in a good shape without any defects and that it can be reliably imported for the following “proper” exploratory data analysis.

9.3 Descriptive Statistics

Descriptive statistics is used to compute summarizing properties of distributions of data (the sample) which results in *numerical* values. Data visualization is a different tool for understanding the data but is not considered to be part of descriptive statistics.² Making predictions about the population is *not* the goal of descriptive statistics, and therefore it is not based on the mathematical foundation of probability theory as in the case of inferential statistics. Descriptive statistics always operates with characteristics that are “simpler” than the original data and that *summarize* important aspects of distributions of data, hence the other, often encountered, term of *summary statistics*. The most commonly used measures for describing and characterizing distributions of data are:

- *Measures of the frequency*: They describe how commonly specific events occur. This can be the number of data points with a certain characteristics or a percentage (e.g., “25% of all data records for the first attribute is smaller than 221.0”).
- *Measures of the central tendency*: These are the mean, the median, or the mode, all of which reveal something about the position of the “center point” of a distribution.

² Among statisticians and data scientists, this distinction is not always followed consistently, and sometimes, data visualization is considered as part of descriptive statistics as well.

- *Measures of dispersion or the variability:* These are measures such as the standard deviation, the minimum and maximum values of the variables. They reveal information about the value range or how “spread out” the data are.
- *Measures of the shape:* The shape of a distribution is described by the skewness and the kurtosis which characterize the asymmetry and the “tail” of a distribution, respectively;
- *Measures of the dependency of variables:* In case that two or more variables are considered: measures of the *statistical dependence* of variables, such as a correlation coefficient.

A number of these measures can be conveniently obtained as moments of distributions, introduced in Chap. 7.

9.3.1 Simple Descriptions of Univariate Distributions

Performing descriptive statistics of univariate data (i.e., data consisting of only one variable) is the simplest way of characterizing data. As an example consider a sample with the following measurements of the temperature in degrees Celsius: [20.5, 19.8, 20.1, 22.4, 20.9]. Here is our summary: “*The sample consists of 5 data points that have a mean value of 20.74, the minimum is 20.1, the maximum is 22.4*”. The values of the minimum and maximum (i.e., the *range* of a distribution) are the simplest way of characterizing the spread of the distribution. They are strongly influenced by extreme values, though. There are a number of other measures that describe, e.g., the shape of a distribution which will be introduced in Sect. 9.3.3 ff.

Another way of presenting a small dataset is a *frequency table*. An example is shown in Table 9.1 which is based on a number of experimental specimens which have different thicknesses. The shown table summarizes how many times a sample with a particular thickness was used by simply listing the “feature” (e.g., the measured variable) and how many times this measurement occurred. From this table we can easily see that the most common specimen has a thickness of 55 μm .

If the data is not discrete but rather continuous, then we would count the number of data points for given intervals that without gaps cover the whole range of interest.

Table 9.1 Frequency table showing the number of measurements taken from specimens of different thicknesses (only discrete values in steps of 5 μm could be measured). The information of the total sum is optional but can be useful for, e.g., computing percentages

Thickness [μm]	Frequency [–]
50	6
55	10
60	9
65	8
70	3
Sum: 36	

In analogy to what was done in the context of discrete probabilities, we can also show the relative frequencies which then should sum up to 1.

9.3.2 Simple Descriptions of Multivariate Distributions

Most of the above univariate measures can also be used for characterizing multivariate distributions: counting data in categories and computing minimum and maximum values can be directly done, but also computing various other measures of univariate distributions can be computed if the data is projected onto a single axis, similar as what would be done in the context of probabilities with a marginal distribution. Example 9.1 shows how this can be applied in practical situations.

Example 9.1 (Using “Data Projections” for Simple Data Analysis)

How can we obtain a “projection” of the dataset onto a single axis and how can we perform calculations with it?

Consider the example of the “Iris dataset” from Sect. 9.2.1 where the feature matrix $\mathbf{X}$ consisted of four columns each of which contained the data for one variable $X_1, \dots, X_4$ which makes it a four-dimensional data distribution:

```
In [1]: X, y = import_iris_dataset()
        print(f"X has {X.shape[0]} rows and {X.shape[1]} columns.")

X has 150 rows and 4 columns.
```

By only considering one variable, the other three “dimensions” are ignored, effectively projecting all data on the first variable.

```
In [2]: x1 = X[:, 0]
        x2 = X[:, 1]
        x1.shape
```

```
Out [2]: (150,)
```

which shows that x_1 is a vector with 150 records. We can now compute, e.g., the mean of the first and the second variable:

(continued)

```
In [3]: import numpy as np
mean_X1 = np.mean(X1)
mean_X2 = np.mean(X2)
print("mean of X1:", mean_X1)
print("mean of X2:", mean_X2)
```

```
mean of X1: 5.843
mean of X2: 3.054
```

In this example, we obtained two mean values, one along each of the first two “directions.” From Fig. 9.2 we would say that the mean along the horizontal direction (X_1) is around 6.0 and the mean of the vertical direction (X_2) is approximately at 3.0. This fits very well to the first two mean values that we obtained from `mean_X1` and `mean_X2` as 5.843 and 3.054, respectively.

For small datasets a *contingency table* can be used that shows the frequency of individual data categories; this type of a table is also known as *cross tab*; see Table 9.2. This representation gives an easy overview over the sample and helps to see that, e.g., specimen type C is underrepresented (5 vs. 10 or 11 data points). A contingency table only works for two-dimensional data, in rare cases also three-dimensional data can be shown by printing “slices” of the dataset, but for higher dimensional datasets or large amount of data this becomes impractical. In such cases, descriptions of conditional or marginal distributions can still be given in such a tabular structure.

Often, the relationship between two or more variables is of interest. A simple means to characterize this is the slope of a regression line (the best fitting curve). A more general and quantitative measure of the dependence between two variables is the *correlation* introduced below.

Table 9.2 Contingency table that shows the number of experiments performed with a specific specimen type and for a certain Cu content. The numbers denote frequencies; the right column and bottom row are the sums of rows and columns, respectively

Specimen type	Cu content			Total by specimen
	<0.2%	0.2–0.5%	>0.5%	
A	3	1	6	10
B	2	4	4	10
C	3	2	0	5
D	4	6	1	11
Total by Cu content	12	13	11	

9.3.3 Central Tendency: The Mean, the Median, and the Mode

The central tendency of a distribution of data tells something about where the “middle” or the “average” is located. It is able to reduce an entire dataset to a single number.

The Mean

The mean of a population is typically not known, and we almost always operate with the *sample mean*. The mean is obtained in the exact same way as the expectation value (cf. Chap. 7 and in particular Sect. 7.3.3) with the only conceptual difference that now we are operating with sample data; all theorems about calculating with expectation values (Sect. 7.1.2) such as the addition of mean values still hold.

Definition 9.1 (Sample Mean) The sample mean of the observations x_i is indicated by a line on top of the variable, $\bar{x}$, and is given by

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i . \quad (9.1)$$

There, the n instances of the sample dataset is the set $\{x_1, \dots, x_n\}$.

In case that the dataset contains more than one feature, one can compute the mean for each feature separately.

How about the Mean of a Continuous Variable?



Why don't we also show the mean for a distribution of a continuous *random variable* (RV) using integration as in Sect. 7.3.3? Because the data obtained from a file is automatically *discrete* data, we never will get a mathematical function from the file. However, in cases where a specific probability density function was fitted to a dataset, then we typically operate with mathematical functions and need to either (i) perform an integration on that or again “discretize” the data and perform a numerical integration which usually implies again summation. Later, we will also consider such a case.

Note that there are different types of means: the *arithmetic mean* from Eq. 9.1 (which is the one that we usually refer to when we speak about *the mean*), the *geometric mean* (a measure based on the product of the data) as well as the *harmonic mean* (which is based on the reciprocal of the sum of the reciprocal data), to name only the most well-known means.

The Median

The median is an alternative description of the central tendency. It is computed as summarized in Algorithm 9.1.

Algorithm 9.1: Computing the median



1. Sort all N data points;
2. Locate the middle value in the sample, i.e., count the first $N/2$ data points;
3. In case that N is an even number: take the average of the middle two observations.

There is no particular symbol that is commonly used for the median. It is also called *50-th percentile* because it splits the amount of data into two sets of equal size (other percentiles are defined analogously). The median is much less sensitive w.r.t. outliers than the mean. For the example dataset of the four numbers $\{2, 4, 6, 50\}$, the median would be 5. The median also would be 5 *regardless* if the value of the last number is 50 or 5000! An example showing how to compute the median for even and odd numbers of values is given in Example 9.2.

Example 9.2 (Mean and Median)



As an example consider the two datasets, dataset #1 consisting of $\{7, 1, 3, 6, 3, 9, 8\}$ and dataset #2 consisting of $\{9, 8, 4, 6, 5, 4, 3, 2, 9, 1\}$. Then, the sorted datasets are shown with the “middle” values marked in boldface:

- dataset #1: $\{1, 3, 3, \mathbf{6}, 7, 8, 9\}$ has the median 6 and the mean of ≈ 5.29 .
- dataset #2: $\{1, 2, 3, 4, \mathbf{4}, \mathbf{5}, 6, 8, 9, 9\}$ has the median $(4 + 5)/2 = 4.5$ and a mean of 5.1.

A typical application in materials science and physics is the analysis of data that contains extreme *outliers* (values that are very far off due to different kinds of errors or because they correspond to “rare events”), as shown in Fig. 9.3 which could be the velocity distribution of atoms in the vicinity of a surface. There, we observe that the data is highly non-symmetrical. We already saw above that mean values are strongly influenced by outliers and also here are expected to be too large. Here, the mean is by a factor 10 larger than the median which also by visual inspection seems to better represent the data.

As a general rule of thumb, one can conclude that mean values are suitable if the data distribution is approximately symmetrical, while medians are more suitable for data that is skewed and/or has a large number of outliers.

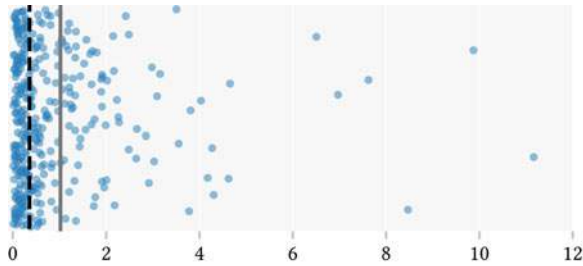
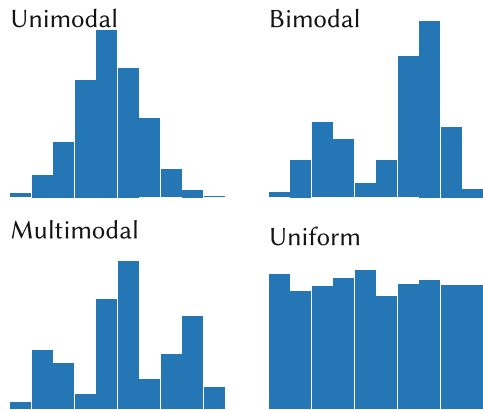


Fig. 9.3 The data (obtained from a lognormal distribution) is highly asymmetric (“skewed”) with many large outliers. The mean is 1.021 (full line), while the median is 0.358 (dashed line). By visual inspection, the median clearly seems to represent the data better

Fig. 9.4 Visualization of four distribution types that result in different types of modes. The multimodal distribution has three maxima and is therefore also called “trimodal” distribution



The Mode

The mode of a dataset is the category or value that has the highest frequency, the *modal value*. The definition of the mode is not mathematically strict because it requires that a category or value needs to have significantly more elements than the others; however, there is no clear rule how large that difference has to be. In case that each value or category occurs the same number of times, then there is no mode. In case of continuous random variables, that data first needs to be binned. Then the mode can be obtained from the frequency distribution or histogram by identifying the maxima. A visualization can be found in Fig. 9.4.

As the modal value is used to characterize the distribution in terms of the “number of peaks,” there are also a number of notions that describe these situations: a *unimodal* distribution has one peak, a *bimodal* distribution has two peaks, and any distribution with more than two peaks is summarized as *multimodal* distribution, while a distribution without distinct maxima is called a *uniform* distribution.

Example 9.3 (Mode)

For the following two dataset we first have to count the number of occurrences of different items, e.g., using a frequency table. Then the mode can be directly obtained as the number with the highest frequency(ies):

- {5, 6, 5, 7, 5, 8, 9, 5}: the mode is 5 because 5 occurs 4 times while any other number occurs fewer times.
- {5, 6, 5, 6, 5, 8, 6, 5, 6}: 5 and 6 are both modes as both occur the same number of times and more often than any other number.

There is no commonly used variable name or symbol for the mode, and most of the time it is just mentioned in text form, e.g., “the mode of the dataset is 2,” or Mode is used in the equation as well.

9.3.4 The “Spread” of a Distribution: The Variance

In Sect. 7.4.3, we already encountered the second centered moment and identified it as the variance. The variance is a measure for how strongly “spread out” a distribution is, the *dispersion* of a distribution. It is a measure that can occur in two slightly different mathematical formulations, depending on whether the population or the sample variance is considered.

Sample Variance and Bessel’s Correction

The variance, or to be more precise the population variance, is often either represented by $\text{Var}[X]$ or by σ^2 . Similar to the case of the mean, also the variance can usually not be computed for the population. Instead, we use the sample variance as an estimate for the population variance. However, the equation for the sample variance is slightly different, due to *Bessel’s correction*.

The reason why a “correction” is needed for computing the sample variance is related to the fact that we are (at least indirectly) using the sample mean, $\bar{x}$, instead of the population mean, μ . As a consequence, any value of the sample is automatically closer to the sample mean than to the population mean. Thus, for the squares of the distance from the mean values, it is

$$\sum_{i=1}^N (x_i - \mu)^2 \geq \sum_{i=1}^N (x_i - \bar{x})^2. \quad (9.2)$$

As a remedy, dividing the left term by n while dividing the right term only by $n - 1$ (this is Bessel’s correction) effectively increases the value of the right term, and therefore the approximation of the population variance becomes more accurate. Of

course, this can also be shown with mathematical rigor, but we will not show this here.

For a given discrete random variable, where all N values have equal likelihoods, the sample variance s is

$$s = \frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2 \quad (9.3)$$

The factor of $1/(N-1)$ from Bessel's correction occurs in many statistics formulations and makes them more accurate. However, for larger sample sizes, the difference to using N becomes negligible.

The following example shows how to compute the two types of variances:

Example 9.4 (Population and Sample Variance)



Assume that the population is given by the dataset $\{16, 11, 9, 8, 1\}$. The population variance is computed by the following steps:

1. Find the mean, μ_x : $\mu = (16 + 11 + 9 + 8 + 1)/5 = 9$.
2. Subtract each data point from the mean; then square the result:

$$\begin{aligned} (16 - 9)^2 &= 49, & (11 - 9)^2 &= 4, & (9 - 9)^2 &= 0 \\ (8 - 9)^2 &= 1, & (1 - 9)^2 &= 64. \end{aligned}$$

3. Add up all of the squared differences from Step 2:

$$(16 - 9)^2 + (11 - 9)^2 + (9 - 9)^2 + (8 - 9)^2 + (1 - 9)^2 = 118.$$

4. Divide Step 3. by the number of items which gives a population variance of 23.6.

Assuming that the dataset is a sample of an unknown population, we would compute the sample variance by dividing by 4 instead of 5, giving a sample variance of 29.5

Standard Deviation

The standard deviation was already introduced in Definition 7.4 in the context of random variables. In analogy, we can also compute the standard deviation for distributions as the square root of the sample or population variance.

Words of advice



The variable name “ s ” is used for the sample variance and sometimes also for the sample standard deviation. In this text we only use σ for the standard deviation; it is either clear from the context whether this is a sample or a population measure, or we explicitly state which one is used.

Example 9.5 (Measurement of the Speed Light (A. A. Michelsen, 1882))



A physics example is based on a dataset published by Stigler [5], who revisited the work done in 1882 by A. A. Michelson: Michelsen performed 5 experiment each of which resulted in 20 measurements which were used to determine the speed of light. To compare this dataset to today’s accurately known speed of light in vacuum ($299,792.5 \text{ km s}^{-1}$), one still would need to convert the data.

A visualization of the data is shown in Fig. 9.5. The bottom plot shows the data as scatter plot where each marker is slightly translucent such that overlapping marker creates a darker color. There, we observe two outliers at the left and one at the right. The top plot is a histogram that shows how many measurements fall within each bin.

The mean is computed as $299852.4 \text{ km s}^{-1}$ (it is shown by the red line in the figure), and the median is $299850.0 \text{ km s}^{-1}$. The only minor difference indicates that the distribution of data is approximately symmetric.

Fig. 9.5 The plots show 100 measurements of the speed of light by A. A. Michelsen in 1882, data taken from [5]

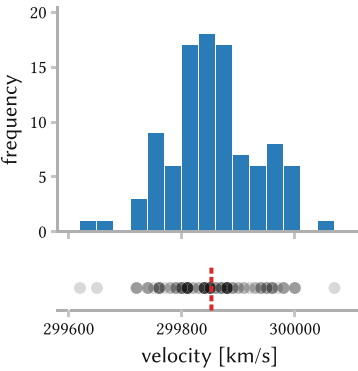


Table 9.3 Summary statistics of the Iris dataset

	Sepal length	Sepal width	Petal length	Petal width
Count	149	149	149	149
Mean	5.84832	3.05100	3.77450	1.20537
std	0.82859	0.43350	1.75965	0.76129
Min	4.30000	2.00000	1.00000	0.10000
25%	5.10000	2.80000	1.60000	0.30000
50%	5.80000	3.00000	4.40000	1.30000
75%	6.40000	3.30000	5.10000	1.80000
Max	7.90000	4.40000	6.90000	2.50000

9.3.5 Example: Summary Statistics of the Iris Dataset

Often, a typical summary statistics includes the number of records (the “count”), measures for the central tendency (at least the mean and the median), measures for the dispersion, as well as the shape of the distribution (e.g., in form of the standard deviation, the minimum/maximum values, and the 25th and 75th percentiles). These values are then computed for each feature of interest and summarized in a table such as the one shown in Table 9.3.

9.3.6 Skewness

Comparing the Mean to the Median

Comparing the mean to the median also helps to characterize if a distribution is symmetrical or not, cf. Fig. 9.6. If the mean is left of the median, then this implies that the distribution is negatively skewed. The reason is that the median splits the dataset into two sets with equal numbers of points. If the mean is left of the median, this implies that for every point in the right set of points, there is a point in the left set of points that is further away from the median. Thus the left half of the distribution stretches further away from the “middle” of the curve. A perfect normal distribution has a skewness of zero because the mean equals the median. However, in practical situations, distributions of data are never perfectly normal distributed. In these cases one can use the skewness to get an estimate of how much the distribution deviates from a normal distribution: for values of the skewness in the range of $[-2, +2]$ one can assume that the data are normally distributed. This, however, requires that the sample size is sufficiently large and has, e.g., at least ≈ 300 data points.

PEARSON’S Moment Coefficient of Skewness

The third *standardized* moment $\tilde{\mu}_3$ is the *skewness* of the distribution, which is a measure for the asymmetry of a data distribution. It is also called PEARSON’S moment coefficient of skewness and quantifies the amount and the direction of the deviation from the symmetry. It is obtained from letting $n = 3$ in Eq. (7.32) as

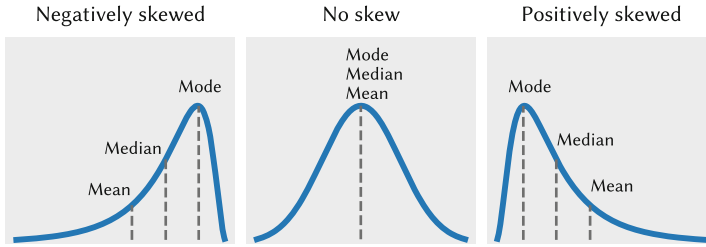


Fig. 9.6 Comparing mean, median, and mode also helps to determine the skewness of a distribution. A distribution is called “negatively skewed” if its tail points into negative direction; a perfectly symmetrical distribution has no skew

$$\gamma_1 \equiv \tilde{\mu}_3 = E \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] \quad \text{with} \quad \mu = E[X]. \quad (9.4)$$

Often, the variable γ_1 is used which is also called “Pearson’s moment coefficient of skewness.” Appendix B.3 shows an alternative formulation for Eq. (9.4). Effectively, skewness describes how much a distribution differs from a normal distribution, either to the left or to the right. Hence, the skewness value can be positive (if the tail is to the left of the maximum), negative (tail to the right of the maximum), or zero (for a symmetric distribution).

Further Skewness Measures

As a last method for estimating the skewness, we briefly introduce PEARSON’s *mode skewness* (which is also called *Pearson’s first skewness coefficient*). It is given by

$$\text{Skew}[X] = \frac{\mu - \text{Mode}}{\sigma} \quad (9.5)$$

where “Mode” is the mode. Sometimes, it is difficult to estimate the mode, for example, in cases where the difference between the maxima and the value range some noise are small.

Pearson’s second coefficient of skewness (also called *median skewness*) is an alternative measure and defined as

$$\text{Skew}[X] = \frac{3(\mu - \text{Median})}{\sigma}, \quad (9.6)$$

where “Median” denotes the median. Both equations give their results in units of the standard deviations, which are therefore dimensionless numbers.

Discussion

Measuring the skewness based on comparing the mean to the median or based on higher order moments might seem to be intuitively reasonable if one is only considering a distribution function given by a mathematical equation. However, most of the time, we are *not* using mathematical formulation for distribution function but rather the data themselves. The data might be noisy and contain outliers. As we saw above, the mean as well as all moments are very sensitive w.r.t. outliers. As all above skewness measures are either directly or indirectly based on moments, they are sensitive w.r.t. outliers as well. Therefore, in these cases, it is advisable to either first identify and remove outliers or to use measures that are not based on moments, e.g., see the approaches presented in [2].

9.3.7 The Fourth Moment: Kurtosis

Finally, the fourth central moment contains information about the “tail” of a distribution. It is given by

$$\text{Kurt}[X] = E \left[\left(\frac{X - \mu}{\sigma} \right)^4 \right] = \frac{E[(X - \mu)^4]}{(E[(X - \mu)^2])^2} = \frac{\mu^4}{\sigma^4}, \quad (9.7)$$

which is called PEARSON’s *measure of kurtosis*. A high value of kurtosis can either imply that the distribution has a “fat” tail which contains a significant part of the probability mass or a distribution is centered around the mean but has many outliers. Below, a visualization method (the box-whisker plot, Sect. 9.4.3) is introduced which helps to identify outliers.

9.3.8 Summary

This section covered many different techniques and descriptions relevant for descriptive statistics. Additionally, we had to differentiate between population and sample properties. To recall the most important variables and operators that were introduced in this section, we summarize them as follows:

Things to Remember ... about notations



Let X and Y denote two features of a dataset where the x_i and y_i are the i -th data record. Then in general we use Latin letters for sample quantities and Greek letters for population properties. In particular:

- the sample mean of a random variable X is given by $\bar{X}$, while the population mean is given by $E[X]$ or μ .
- the sample variance is s^2 , while the population variance is denoted by $\sigma^2 = \text{Var}[X]$
- the covariance between two RVs X and Y is σ_{XY} or $\text{Cov}(X, Y)$ for the population, and the sample covariance for a number of measurements is s_{xy}^2
- covariance matrix for a population and a sample is given by Σ and $\mathbf{S}$, respectively.

9.4 Data Visualization

Data visualization is an extremely powerful tool for [EDA](#) as well as for understanding and analyzing data in general (see the work by Edward R. Tufte, e.g., [6]). In fact, even in the previous section of *descriptive statistics*, a few plots were shown in order to *show* what one or the other numerical measure (such as the median) means. Additionally, we also introduced in Sect. 9.2.2 a quick type of “initial visualization,” to be used to get a very brief overview. As opposed to this, in the current section, the data visualization strategies are somewhat more involved, and some of those plots even might need some explanations so that the reader becomes familiar with how to read and interpret some of the plots.

One of the famous and very educative examples which shows why descriptive statistics necessarily needs to be complemented by data visualization is *Anscombe's quartet*. The statistician F. ANSCOMBE created in 1973 four datasets, each of which had the same statistical properties (such as mean values, variances, the correlation coefficient) as the three others [1]. The dataset is shown in Table 9.4. Upon investigation, one finds that each of the four datasets has the following properties:

- each of them has $n = 11$ observations
- the mean of each X is exactly $\bar{x} = 9$
- the mean of each Y is $\bar{y} = 7.50$ which is accurate up to two decimal places
- the sample variance of each X is exactly $s_x^2 = 11$
- the sample variance of each Y is $s_y^2 = 4.125 (\pm 0.003)$
- the correlation coefficient between any pair of X and Y is $\text{Cor}(X, Y) = 0.816$ with an exact match of the first three decimal places
- fitting a line to the data (see *linear regression*) always gives the equation $Y = 3.00 + 0.500 \cdot X$, where the two constants are up to two and three decimal places accurate, respectively.

Table 9.4 The data of *Anscombe’s quartet* given as a table similar to the one presented in the original publication [1]. The four datasets are indicated by the roman numerals I–IV

I		II		III		IV	
X	Y	X	Y	X	Y	X	Y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

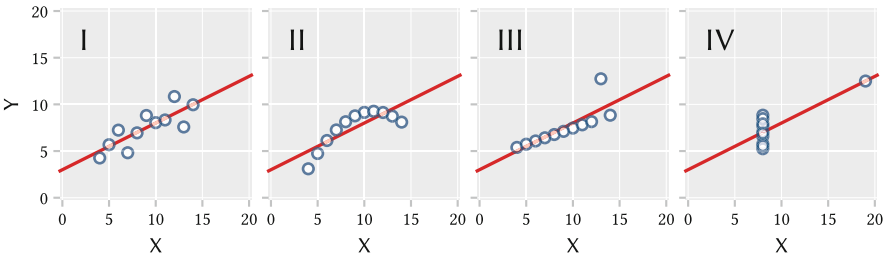


Fig. 9.7 A visualization of the data of *Anscombe’s quartet* using four scatter plots: while the datasets have nearly identical statistical properties, the data distributions are upon visual inspection entirely different. The roman numbers indicate the number of the dataset

While the coincidence of the properties for all datasets suggest that the data itself are identical, visualizing the data as in Fig. 9.7 reveals that the distributions look entirely different: the left dataset I shows roughly a linear distribution, while dataset II shows a nonlinear relation for which Pearson’s correlation coefficient is not a suitable measure. The second plot from the right, dataset III, again shows a linear behavior; however, the slope should be different from dataset I—except for the outlier with $Y_3 = 12.74$. Last but not least, dataset IV does not show any relation between the two variables, but the one outlier is sufficient to strongly impact the result, e.g., in terms of the correlation coefficient or the regression line.

9.4.1 Scatter Plots

Scatter plots are the most commonly means of data visualization. There, each datum is represented by a marker, often a circle, a star, or any other shape that clearly indicates the coordinate of the point. These plot types can be used for univariate (cf. Fig. 9.5) as well as for bivariate (shown in Fig. 9.9 as well as in Figs. 9.7 and

8.9) or even for trivariate data distributions. However, scatter plots in EDA are most commonly used for bivariate variables. Scatter plots can be also used for datasets with more features by simply showing a projection of the data into a two-dimensional plane. Beware that the marker are not interconnected by lines. If lines show up in these plots, then usually it is to indicate trends or highlight, e.g., a mean value.

We can also “annotate” such plots, e.g., by indicating the mean as a point (e.g., with a different color or a different marker type) or by showing a *regression line*³ as in Fig. 9.7.

9.4.2 Histograms

Histograms are related to scatter plots and represent a “coarsened” picture of the scatter data. The main aspect of histograms is that the whole spatial domain is subdivided into a number of sub-domains, the “bins” which are usually of equal size. Then the number of values that fall into each sub-domain is counted, which is why continuous data and not categorical data is required. The number of data points per sub-domain are the numbers that are visualized. In the limit case of infinitesimally small bins, the histogram would look like a scatter plot. Histograms are useful for observing where, e.g., maxima are located (allowing to infer if it is a uniform, bimodal, or multimodal distribution) and for getting an impression of the structure and shape of distributions (is it a symmetric distribution or is it skewed to one directions?).

Histograms for Univariate Data

Binning in the context of univariate data implies that the whole range of interest $[a, b]$ of an input variable X is subdivided into an appropriate number of n bins, i.e., *intervals*. For simplicity, we assume that all bins have the same size $\Delta x = (b - a)/N$. Bin number i is then given by the interval $[i \cdot \Delta x, (i + 1) \cdot \Delta x]$. Next, the number of points that are located in each interval are counted. The histogram contains rectangular *bars* where one side has the length Δx_i and the other side’s length is proportional to the number of data points in the bin. This visualization directly corresponds to the *frequency tables* that we encountered in Sect. 9.3.

An important parameter for a histogram is the “bin size.” In principle, it can be chosen arbitrarily; however, it has a strong impact on the visual representation, cf. Fig. 9.8, and therefore could have been accidentally chosen such that the relevant feature of a dataset might not become apparent. As a rule of thumb, the bin size should be chosen such that most of the bins contain several data. Additionally, the bin size does not necessarily have to be the same for each bin. If the frequency is divided by the total number of data, then *relative frequencies*

³ This is, in a nutshell, the line that fits best to the data, typically minimizing the sum of the squared errors.

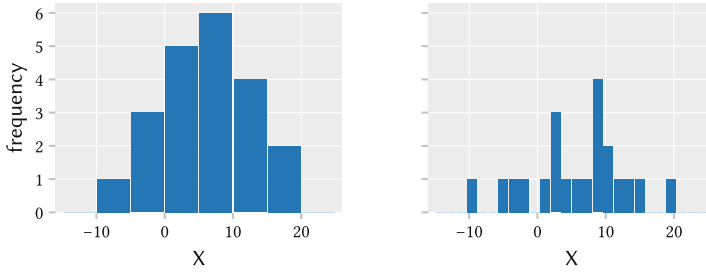


Fig. 9.8 Histograms of the values $X = [10, 3, 3, -2, -5, 9, 5, 8, 9, 4, 9, 9, 15, 11, 1, -4, -9, 13, -7, 3]$ for a bin size of 5 (left panel) and ≈ 1.5 (right panel). A too small bin size (right panel) results in many empty bins or bins with very few data. In this visualization there is a tiny gap between the bars; however, these are just for guiding the eye

are depicted. Histograms help to estimate the probability density function; *kernel density estimation* plots are an alternative visualization method that operates with smoothed data.

Words of advice



Even though *bar diagrams* look like histograms (in particular when the “bars” are slightly separated for visual purposes as in Fig. 9.8), there is a fundamental difference: a box plot always represents data of different categories, while a histogram represents a distribution of non-discrete and quantitative data where individual “bars” have a fixed order.

Histograms for Bivariate Data

In principle, bivariate variables could be visualized as a histogram in a (pseudo) three-dimensional style, i.e., two variables and the frequency are the three axes. However, it is difficult to quantitatively read the three-dimensional projection into the two-dimensional paper plane. As a better alternative, one can map the frequencies to color values (middle panel of Fig. 9.9) from which we can easily identify the first and second largest maxima. This representation also makes it possible to quantitatively compare to bivariate distributions. There are numerous applications in materials science and physics, among them are dislocation densities which, if obtained from *electron backscatter diffraction (EBSD)* measurements or simulations, are shown and compared in such a “coarse-grained” histogram representation; see, e.g. [4, 8, 9].

A variant of such a histogram quantifies the frequencies by the size of markers (a kind of scatter plot that allows scaling the marker size) as shown in the right panel of Fig. 9.9. There, the area of the marker is proportional to the frequency and not the radius. Such kind of diagram also allows to use colors to represent other aspects of the data, e.g., for the Iris dataset this could be used to indicate the four different classes of flowers.

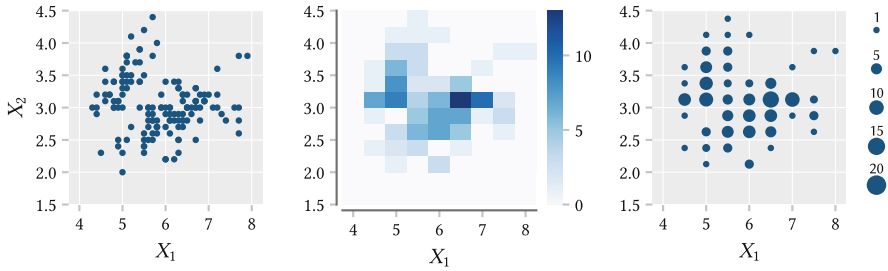


Fig. 9.9 Visualization of bivariate data (the first two features of the “Iris dataset”: scatter plot (left), 2D histogram (middle), and histogram using different marker sizes instead of colors

Even if in a scatter plot two points coincide, in a histogram plot they can be differentiated due to the different frequency values. As an variant of the two-dimensional histogram, it is also possible to use bins of hexagonal shapes. As rectangular array-like grids emphasize the horizontal and the vertical directions, hexagons can help disturb these strict patterns. They are additionally more similar to circles which means that the data aggregation is more efficient and balanced. However, as this kind of data binning is not commonly used, it might be more difficult to “read.” `matplotlib` provides the command `hexbin` for this type of plot.

9.4.3 Box-Whisker Plots

A box-whisker plot (or only: box plot) is a very compact representation of several statistical properties of a univariate distribution. The below used version was initially developed by the statistician JOHN TUKEY [7]. A main ingredient are the *quartiles* which partition the whole dataset into four subsets, each of which contains the same number of data. For example, the first quartile ending at Q_1 contains 25%-th *percentile* of the data with the lowest values. The 50-th percentile (Q_2) splits the whole dataset into two halves, one where all data have a value smaller than Q_2 and one where all values are larger than Q_2 . The second quartile (= the 50-th percentile) corresponds to the above-introduced median; the 75%-th percentile (Q_3) is defined following the same strategy. For determining all three quartiles, the same rules concerning odd and even number of data as in Algorithm 9.1 applies.

The box plot (a sketch of all relevant quantities is shown in Fig. 9.10) consists of a rectangle that covers the data from the first quartile to the third quartile. The median is shown as a line inside the rectangle; for a symmetric distribution, that line is exactly centered within the box. Often, the mean is also indicated, e.g., by a small marker. The difference between Q_3 and Q_1 defines the *interquartile range* (IQR),

$$\text{IQR} = Q_3 - Q_1, \quad (9.8)$$

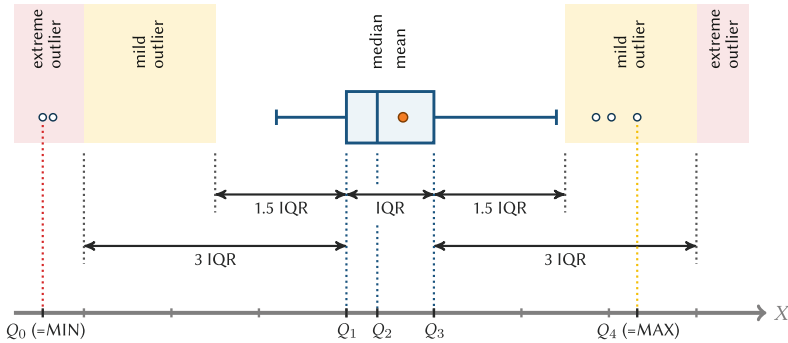


Fig. 9.10 Sketch of the elements contained in a box-whisker plot (see the text for more explanations)

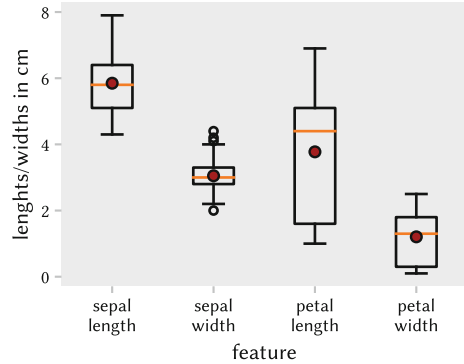
which is a length and gives the extent of the rectangle. The IQR is used in a box plot as a measure for the dispersion of a distribution. Unlike using the standard deviation or the range of the variable (i.e., the maximum and minimum values), the IQR is much less affected by outliers.

As an extension of the box plot, *whiskers* are added (which gave this plot type its name), a line that stretches away from the box starting from Q_1 or Q_3 . In the simplest version, the line then terminates at the data point with the minimum or maximum value, respectively, excluding outliers. The minimum is also denoted as the Q_0 percentile and the maximum as the Q_4 or 100th percentile. Following Tukey et al. [7] a more robust approach is to use a length of 1.5 IQR starting from Q_1 into negative direction which terminates at the smallest data point that falls into this range. Similarly for the whisker on the positive side, the whisker ends at the largest point that falls into the interval given by Q_3 and $Q_3 + 1.5 \text{ IQR}$. The fact that the whisker ends at a data point and not *exactly* after the distance of 1.5 IQR is the reason why the two whiskers may have different lengths even in cases where coinciding mean and median suggest a symmetric distribution. There are other conventions as well which is why it is important to clearly state which kind of quantity the whiskers denote. In this text we follow the “1.5 IQR” convention.

The position of $Q_1 - 1.5 \text{ IQR}$ is called *lower fence*; that of $Q_3 + 1.5 \text{ IQR}$ is called *upper fence*. The *fences* are the points past which data are considered as extreme values called *outlier*. Those might be the result of measurement errors or can be due to a new phenomenon so far not seen in the data, to name but two possibilities.

Sometimes, an additional fence is considered at $Q_1 - 3 \text{ IQR}$ and at $Q_3 + 3 \text{ IQR}$, the *outer fences*. Points beyond the outer fences are then called *extreme outlier*, points in the intervals $[Q_1 - 3 \text{ IQR}, Q_1 - 1.5 \text{ IQR})$ and $[Q_3 + 1.5 \text{ IQR}, Q_3 + 3 \text{ IQR})$ are called *mild outlier*, and the corresponding fences are called *inner fences*. Outliers are the only ones in a box plot where data points themselves and not summarizing measures are shown. All of the above quantities are condensed in a box-whisker plot as shown in Fig. 9.10.

Fig. 9.11 Box-whisker plot of the four features of the Iris dataset. The red dot shows the mean; the orange line inside the box shows the median



The box-whisker plot allows to compare several random variables and is a very compact data representation. As an example, Fig. 9.11 shows all four features of the Iris dataset. From this plot, we can easily observe that the value range of the sepal lengths and the petal widths do not overlap. We also see that the sepal length, unlike the petal length, is approximately symmetric because the median and the mean coincide. The sepal width contains a few outliers, but compared to the petal length, its distribution is very narrow. Thus, box plots are very information-rich and very well able to visualize some of the important numerical measures of distributions.

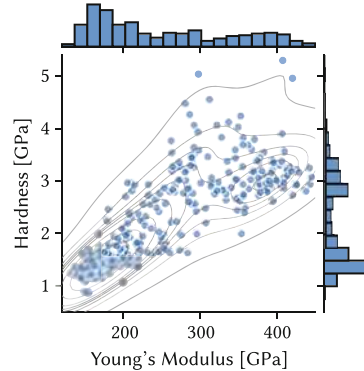
9.4.4 Advanced Exploratory Plot Types

The above-introduced plot types are the most important plot variants that are used in EDA. There are a few less commonly and more specialized plot types which were skipped—mainly because many of them require more advanced statistical analysis approaches, such as the *kernel density estimation* (KDE) plot or the so-called Andrews curves. There are, however, also commonly plot types that consist of a combination of two or more of the above-introduced plot types which will briefly be introduced in the following.

Scatter Plot with Marginal Histograms

A compact way of visualizing bivariate data (or a respective projection) is to show the data as a scatter plot together with two histograms of the marginal distributions (i.e., one-dimensional “projections” of the data along one direction). Figure 9.12 shows this for a part of the dataset MDS-5 (Nanoindentation). It becomes very well visible that the hardness distribution is a bimodal distribution, while this is not so obvious for the Young’s modulus. This is difficult to see in the scatter plot, even though we already made the marker slightly translucent such that overlapping markers result in darker points, helping to estimate density variations. We have also added contour lines from a *kernel density estimation* (KDE) which estimates the distribution using a number of Gaussian kernels and which also helps to visualize

Fig. 9.12 Two-dimensional scatter plot of the MDS-5 (Nanoindentation) dataset in which additionally the marginal distributions are shown as one-dimensional histograms



the structure of the data. Of course, we could continue to add more and more information into the figure, such as the mean, median, and mode of the bivariate distribution and of the marginals; however, in most cases such a plot has the purpose to highlight a few specific aspects, and it is advisable not to “overload” such a plot.

Scatter Plot Matrix with Histograms

In those situations where a dataset contains more than two features, we already introduced the scatter plot as a means to visualize a projection of the data onto two selected features. This kind of approach can be done for all possible combinations and results in a *scatter plot matrix*, where each row and each column is one particular combination of two features. In Fig. 9.13 a scatter plot matrix is shown for the whole dataset “DS-1 (Iris Flowers, Sect. 4.7),” where different colors highlight different species. Scatter plots that are located on the main diagonal of the matrix use the same features for both coordinates, and thus, all data points fall on a line. Usually, this does not reveal much information. Instead, often histograms are shown for this feature instead. As the matrix is symmetric, the scatter plots above the diagonal can be removed to avoid redundancy. With this, we have a very compact, two-dimensional representation of a whole, multivariate dataset. For example, one can observe that the *Iris versicolor* and *Iris virginica* are the two classes that overlap most, while the *Iris setosa* is rather separated from the other two classes. We also can easily see the ranges of the different variables (i.e., the interval from minimum to maximum), and by looking at the scatter plots and the histograms, we get an idea about the spread and correlation of the different variables. As an extension of this plot, type one could also plot the individual distributions of all three classes in each histogram which can reveal information about the intersection of the different classes as well as on the amount of data per class. Yet another extension is to replace histograms by KDE plots; this is done by default, e.g., by the Python package *seaborn* which is part of the *Scientific Python Stack*.

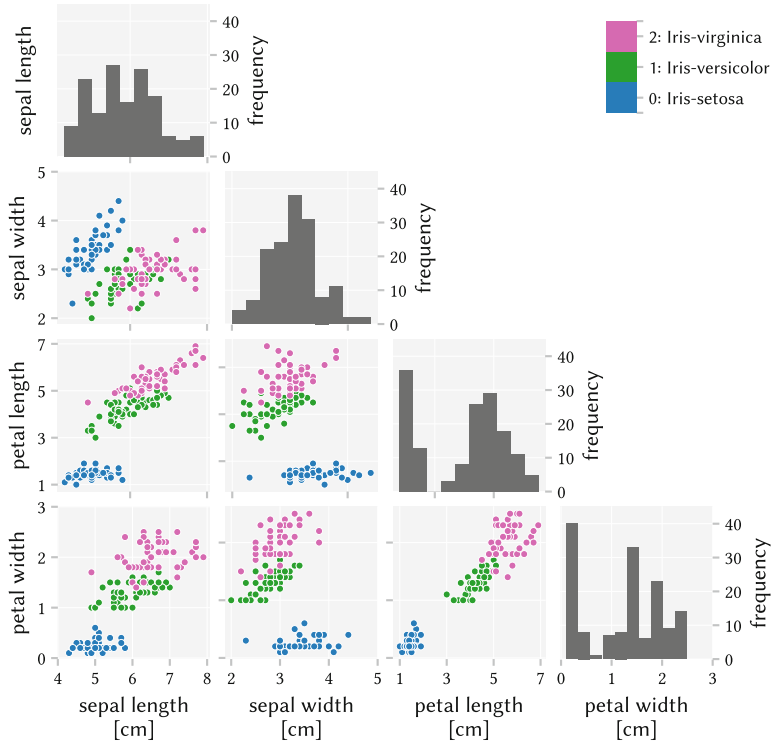


Fig. 9.13 Scatter plot matrix of the Iris dataset. Additionally, histograms are shown on the main diagonal

9.5 Exercises

9.1 (Box-Whisker Plot) After the length of dislocations in a pileup observed during an in situ TEM experiment was analyzed, the following line lengths in nanometer were found: 47, 49, 56, 59, 61, 63, 66, 69, 69, 70, 70, 70, 71, 71, 71, 72, 73, 75, 75, 76, 76, 78, 79, 81.

Calculating the following quantities: minimum, maximum, median (Q2), first quartile (Q1), and third quartile (Q3) as well as the IQR. In a second step, create a box plot, and compare it with your calculated values to ensure that you (or the plotting code) did not make any mistakes.

9.2 The following DSs contain the grain size distributions of an iron alloy which has been subjected to two different heat treatments.

1. Visualize the data.

Table 9.5 Grain size distribution. Heat treatment up to 1100 °C

Relative frequency (%)	Grain size (μm)
2.1	40
9.5	60
20.7	80
33.0	100
20.8	120
7.6	140
5.1	160
1.2	180

Table 9.6 Grain size distribution. Heat treatment up to 1400 °C

Relative frequency (%)	Grain size (μm)
1.3	40
2.0	60
4.5	80
6.2	100
10.0	120
15.6	140
15.9	160
13.5	180
9.1	200
9.5	220
5.5	240
5.0	260
1.9	280

2. Calculate mean and median. Hint: be careful with the median of a histogram; use cumulative frequencies (Tables 9.5 and 9.6).

9.3 Materials design is a discipline that seeks to improve specific properties of the material, such is the case of polymer design according to the molecular weight distribution. Visualize the following data and comment on the modal values:

```
In [1]: logM = np.array([2.5, 3.0, 3.5, 4.0, 4.5,
                        5.0, 5.5, 6.0, 6.5, 7.0])
distM_1 = np.array([0.02, 0.05, 0.1, 0.31, 0.45,
                    0.55, 0.49, 0.4, 0.35, 0.05])
distM_2 = np.array([0.01, 0.1, 0.14, 0.33, 0.12,
                    0.09, 0.32, 0.46, 0.12, 0.05])
distM_3 = np.array([0.03, 0.29, 0.25, 0.11, 0.09,
                    0.26, 0.4, 0.02, 0.39, 0.09])
```

9.4 Create a box plot to analyze the data from the dataset “MDS-4 (Properties of Chem. Elements).” For each properties (such as the atomic radius or the electronegativity), there should we one box plot. What can you tell about the distribution shapes and about outlier?

References

1. F. J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27 (1): 17–21, 1973. DOI <https://doi.org/10.1080/00031305.1973.10478966>. URL <https://www.tandfonline.com/doi/abs/10.1080/00031305.1973.10478966>.
2. G. Brys, M. Hubert, and A. Struyf. A robust measure of skewness. *Journal of Computational and Graphical Statistics*, 13(4): 996–1017, 2004. ISSN 10618600. URL <http://www.jstor.org/stable/27594089>.
3. W. McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.
4. S. Sandfeld and G. Po. Microstructural comparison of the kinematics of discrete and continuum dislocations models. *Modelling and Simulation in Materials Science and Engineering*, 23(8): 085003, 2015. DOI <https://doi.org/10.1088/0965-0393/23/8/085003>.
5. S. M. Stigler. Do robust estimators work with real data. *Annals of Statistics*, 5: 1055–1098, 1977.
6. E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, 2 edition, 2001. ISBN 978-0-9613921-4-7.
7. J. W. Tukey et al. *Exploratory data analysis*, volume 2. Reading, MA, 1977.
8. D. Wallis, L. N. Hansen, T. Ben Britton, and A. J. Wilkinson. Geometrically necessary dislocation densities in olivine obtained using high-angular resolution electron backscatter diffraction. *Ultramicroscopy*, 168: 34–45, 2016. ISSN 0304-3991. DOI <https://doi.org/10.1016/j.ultramic.2016.06.002>. URL <https://www.sciencedirect.com/science/article/pii/S0304399116300821>.
9. C. Zhang, H. Song, D. Oliveros, A. Fracziewicz, M. Legros, and S. Sandfeld. Data-mining of in-situ tem experiments: On the dynamics of dislocations in CoCrFeMnNi alloys. *Acta Materialia*, 241: 118394, 2022. DOI <https://doi.org/10.1016/j.actamat.2022.118394>.



Commonly Encountered Distributions in Materials Science and Engineering

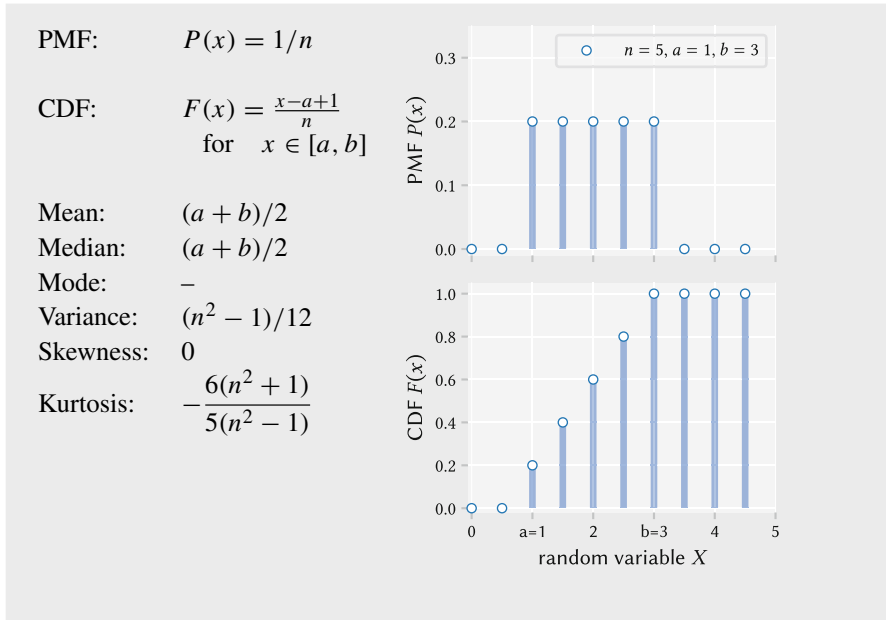
10

10.1 About the Following Discrete and Continuous Distributions

All different types distributions can be subdivided into two broad classes: discrete and continuous distributions. The first half of this chapter introduces the most commonly used discrete distributions, characterized by their *probability mass function* (PMF) and *cumulative distribution function* (CDF). These are in particular: the discrete uniform, Bernoulli, Binomial, Geometric as well as the Poisson distribution. They will be introduced in Sects. 10.2–10.6.

The second half considers continuous distributions, given by their *probability density function* (PDF) and CDF. One of the most important distributions is the normal distribution (including the bivariate and multivariate normal distribution), for which a number of useful mathematical relations can be derived. Further notable distributions are the lognormal, exponential, and the logistic distribution. These will be introduced below in Sects. 10.7–10.13.

10.2 Discrete Uniform Distribution



Additional Details One of the most well-known discrete distribution is the discrete uniform distribution. There, all n values $x_1, \dots, x_n$ have the same likeliness $P(x) = 1/n$ to be observed. In the “canonical formulation,” the distribution would start at $x = 0$, and the values of the random variable would all be integer numbers. For this, formulation requires only a single parameter, n , to describe the shape of the distribution. n is the number of discrete values of the random variable X .

The formulation shown above is a generalization, where (i) the random variable is not restricted to only integer values and (ii) the distribution is shifted. For this formulation, three parameters are required: the lower bound a , the upper bound b , and the number of values n .

It is easy to recover the canonical form, which is non-zero in the interval $[1, n]$, by letting $a = 1$ and $b = n$, resulting, for example, in a mean value of $(n + 1)/2$.

A commonly used notation for drawing random numbers from the interval $[a, b]$ is

$$X \sim U(a, b) \quad (10.1)$$

Note, however, that U denotes a uniform random distribution which is a continuous distribution.

Examples and Application Examples include throwing a *fair*, six-sided dice for which $x_i = 1, \dots, 6$ and $P(x_i) = 1/6$ for all i . Other examples are tossing a coin or drawing a random card from a deck of cards. Furthermore, whenever an integer number, e.g., as index for an array needs to be drawn from a range of numbers with equal likelihood, this distribution would be used. In materials science and engineering, Monte Carlo simulations are based on this type of random sampling. An example is the dataset MDS-2 (Ising Model), where the site of the magnetic dipole to flipped is chosen from a uniform random distribution of all integers from $[0, N^2)$ where N is the size of the system.

Python/SciPy As the `scipy` library does not contain the discrete uniform distribution, we show how to plot the distribution and how to draw uniformly distributed discrete random numbers.

A minimal example for plotting the distribution:

```
In [1]: import matplotlib.pyplot as plt
import numpy as np

In [2]: n = 5          # number of values
a, b = 1, 3          # lower and upper limit of interval

x = np.linspace(a, b, n)
pmf = 1 / n * np.ones_like(x)
cdf = np.cumsum(pmf)

In [3]: fig, ax = plt.subplots(nrows=2, sharex=True)
ax[0].bar(x, pmf, width=0.1)
ax[1].bar(x, cdf, width=0.1)
ax[0].set(ylim=(0, 0.5), ylabel="PMF")
ax[1].set(ylim=(0, 1.05), xlabel="X", ylabel="CDF")
# plt.plot() # required, if not in a jupyter notebook
```

There, the function `np.ones_like()` creates a `numpy` array that has the same shape as the argument `x`. Specifically, multiplying with the factor $1/n$ creates an array with $n = 5$ elements all of which have the value $1/n$.

A minimal example for drawing samples from the distribution uses the (legacy) `numpy` function `randint`:

```
In [1]: import numpy as np

In [2]: np.random.randint(a, b + 1, 15)

Out [2]: array([3, 3, 2, 2, 3, 1, 1, 2, 2, 1, 2, 3, 1, 1, 3])
```

10.3 Bernoulli Distribution

$$\text{PMF: } P(X) = \begin{cases} 1-p & \text{if } X=0 \\ p & \text{if } X=1 \end{cases}$$

$$\text{CDF: } F(X) = \begin{cases} 0 & \text{if } X < 0 \\ 1-p & \text{if } 0 \leq X < 1 \\ 1 & \text{if } X \geq 1 \end{cases}$$

$$\text{Mean: } p$$

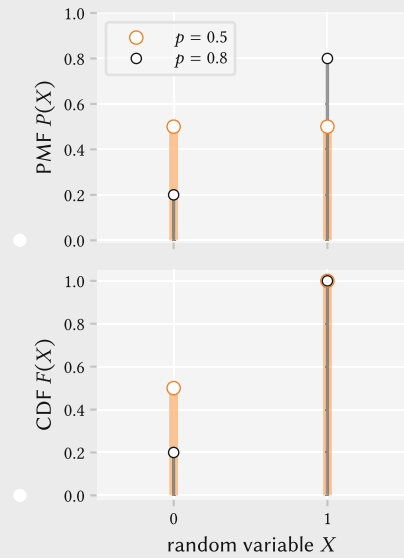
$$\text{Median: } \begin{cases} 0 & \text{if } p < 1/2 \\ [0, 1] & \text{if } p = 1/2 \\ 1 & \text{if } p > 1/2 \end{cases}$$

$$\text{Mode: } \begin{cases} 0 & \text{if } p < 1/2 \\ [0, 1] & \text{if } p = 1/2 \\ 1 & \text{if } p > 1/2 \end{cases}$$

$$\text{Variance: } p(1-p)$$

$$\text{Skewness: } (1-2p)/\sqrt{p-p^2}$$

$$\text{Kurtosis: } \frac{1-6p(1-p)}{p(1-p)}$$



General Description The BERNOULLI distribution is the simplest of all discrete distributions and is sometimes colloquially called the “yes/no distribution.” It represents a random process that has only two possible outcomes during each trial—the *Bernoulli trial*. The outcomes are typically called “Success” and “Failure” and are represented by the integer numbers $X = 1$ for Success and $X = 0$ for Failure. The probabilities of obtaining one or the other outcomes may be different: “Success” is expected with the probability of p (because $P(X = 1) = p$), while “Failure” occurs with the probability $q = 1 - p$ (because $P(X = 0) = 1 - p$). An alternative formulation of the above PMF for the case that the random variable X takes only the value 0 or 1 is

$$p(x) = p^x q^{1-x}, \quad x = 0, 1. \quad (10.2)$$

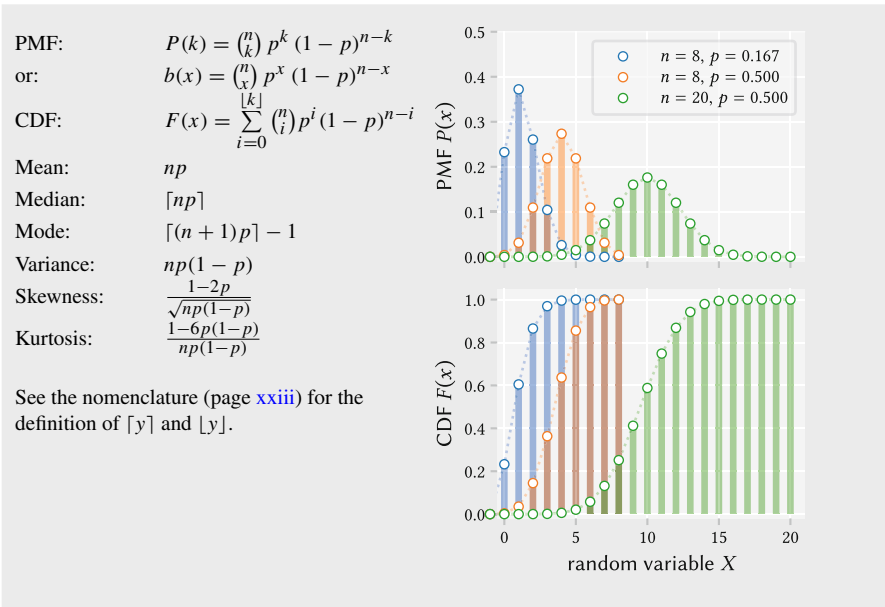
The Bernoulli distribution is also sometimes called “point binomial distribution.”

Examples For the coin toss, the two values would be Heads and Tails with equal probability, $p = 0.5$. Getting a 1 or 2 from a fair dice could be value 1 with a probability of $p = 2/6$, while 3, 4, 5, and 6 are the second value with a probability of $4/6$. But even throwing two dies and defining Success as getting two 1s is an experiment that can be represented by this simple statistical model.

A number of more complex distributions contain the BERNOLLI distribution as a basic building block such that the BERNOLLI distribution can be considered as a special case if there is only one experiment. Among those are the binomial distribution in Sect. 10.4 (which gives the probability for a certain number of “successes” during a number of Bernoulli trials), the negative binomial distribution introduced in (giving the probability of a number of failures during several BERNOLLI trials), and the geometric distribution from Sect. 10.5 (predicting the probability of a certain number of “Failures” until the first success occurs).

Application in Science, Engineering, and Machine Learning An engineering example is the failure of a specimen which, in the simplest case, also can be modeled by such a binary distribution. In *machine learning* (ML), decision trees are based on sequences of probability-based binary decisions, modeled by BERNOLLI distributions.

10.4 Binomial Distribution



General Description The binomial distribution is one of the distribution types that is based on Bernoulli trials and the Bernoulli distribution. More specifically,

there are n mutually independent Bernoulli trials, each of which can take the outcomes *Failure* or *Success*. Then, the binomial distribution gives the probability of observing, e.g., the outcome k times Success in a during performing n Bernoulli experiments. In each individual experiment, the probability for Success is p , and, as a consequence, the probability for Failure is $1 - p$. Such a distribution may result from random sampling of a finite population where each drawn sample is returned (“with replacement”) or from a population that is infinitely large. In the latter case, it does not matter if the drawn sample is returned or not.

The equation of the PMF can be easily understood: recall, that $\binom{n}{k}$ gives the number of possible combinations C_k^n that consist of k items, taken from a set of n items with replacement (cf. Definition 5.5). Here, each of the k Success outcomes occur with a probability of p , while the $n - k$ Failures have a probability of $1 - p$ each. Often, the binomial probability distribution is also denoted by $b(x) = \binom{n}{x} p^x (1 - p)^{n-x}$. In other words, the random variable counts the number of successes in n independent trials (“sampling with replacement”) of a random experiment where the probability of success is p . The shape of the distribution is given by the two parameters n and p . Due to its definition, it can also be represented as the sum of n independent Bernoulli random variables with parameter p such that the resulting random variable Y is given by $Y = \sum_{i=1}^n X_i$. A consequence is the “compound probability,” where, for computing the probability of getting *at most* k successes, we may add all the different success options using the individual binomial distributions. For example, it is $P(x \leq 3) = P(x = 1) + P(x = 2) + P(x = 3)$. For obtaining the probability of getting *at least* k successes, we use the binomial distribution function and add together all relevant success options. Last but not least, it might seem to be a strong limitation that we only differentiate between Failure and Success events. However, it turns out, that this is sufficient to cover a large range of different cases.

Examples The mean of the distribution is simply np and gives the number of expected successes μ in n Bernoulli trials which is in line with our intuition, e.g., if each Bernoulli trial has a probability of $p = 0.5$, then for 100 trials we would expect on average to get 50 Successes, which is exactly $n \cdot p = 100 \cdot 0.5 = 50$.

Application in Science, Engineering, and Machine Learning Binomial distributions are often used in situations where a specimen or device is tested whether it is still intact or whether it is faulty. When investigating specimens from a production batch, it is important that the batch is sufficiently large because otherwise the above requirement (infinite population) would be violated. In such cases, the hypergeometric distribution would have to be used instead.

Example 10.1 (Probability of Faulty Specimens)

A certain type of metallic samples are very prone to cracking. About 25% of them show small fracture networks after the surface treatment. Altogether 15 samples undergo a visual inspection. Before they should be shipped to various labs, the following questions arise:

What are the values of the PMF?

It is $P(X = 0) = 0.25$ for a faulty specimen from which follows that $P(X = 1) = 0.75$ for an intact specimen.

What is the probability that more than 2 specimen have cracks?

$$P(X > 2) = 1 - P(X = 0) - P(X = 1) - P(X = 2) = 1 - 0.236 = 0.764.$$

What is the expected value of broken specimens and their standard deviation?

Starting from the definition of the expectation value, we sum up products of frequencies and probabilities, resulting in

$$E[X] = 0 \cdot 0.01336 + 1 \cdot 0.06682 + 2 \cdot 0.15591 + \dots + 15 \cdot 0 = 3.75$$

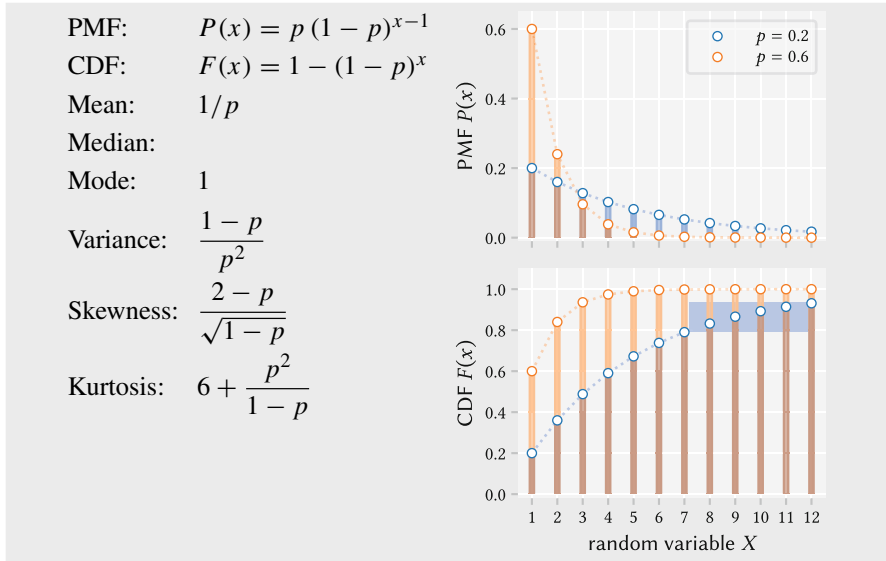
With this, the variance follows as

$$\begin{aligned} \text{Var}[X] &= E[X^2] - E[X]^2 \\ &= (0^2 \cdot 0.01336 + 1^2 \cdot 0.06682 + 2^2 \cdot 0.15591 + \dots + 15^2 \cdot 0) - 3.75^2 \\ &= 16.875 - 3.75^2 = 2.813 \end{aligned} \tag{10.3}$$

The standard deviation s is the $s = \sqrt{\text{Var}[X]} = 1.677$.

10.5 Geometric Distribution

Assuming that the values of the random variable are $x \in \mathbb{N}$ (i.e., positive integer numbers excluding zero), it is



General Description The geometric distribution is a type of discrete probability distribution which gives the probability that, in a Bernoulli trial (cf. Sect. 10.3), the first time that the event “Success” occurs required k independent trials before “Success.” As in the Bernoulli trial, the probability of success of an individual trial is p , and the probability of failure is $1 - p$. The geometric distribution represents the probability that after x trials for the first time “success” occurs. For example, in the above figure, the probability that after three trials “success” occurs is $P(x = 3) \approx 0.1$ if the probability for “success” is $p = 0.2$.

The above equations for the PMF and CDF assume that the random variable k is an integer number $k \in \mathbb{N}$ where $\mathbb{N}$ denotes the set of all positive integer numbers. An alternative formulation exists where x does not start at 1 but rather at 0 (the number of trials up to and including the first “success”). The geometric distribution is a special case of the negative binomial distribution.

Examples A typical example is related to the number of defective specimens: assume that we know that 3% of all samples or specimens have a defect. We can then determine the probability of finding 0, 1, 2, or more specimens that are non-defective using the geometric distribution. For example, the probability that we find two good samples before a defective sample occurs is obtained from $k = 3$ and $p = 0.03$ as $P(k = 3) = 0.03(1 - 0.03)^{3-1} = 0.0282$: in about 1 out of 30 “experiments” (where the single experiment consists of looking at samples until we find a defective sample), there are two good samples before we find a bad one.

This distribution is often also called “failure time distribution.”

Application in Science, Engineering, and Machine Learning The geometric distribution can be encountered in Markov chain models which are used for making predictions in the field of statistical mechanics or engineering physics, e.g., when representing a random walk as in the context of Brownian motion of atoms or molecules.

Python/SciPy The above example can be also easily computed using SciPy:

```
from scipy import stats

p = 0.03 # probability of failure
k = 3    # number of times until the first defective specimen occurs

print('probability for finding a defect after 2 non-defective specimens:',
      stats.geom.pmf(k, p))
```

10.6 Poisson Distribution

PMF:

$$P(x) = \frac{\mu^x \exp(-\mu)}{x!}$$

CDF:

$$F(x) = \exp(-\mu) \sum_{i=0}^{\lfloor x \rfloor} \frac{\mu^i}{i!}$$

Mean:

μ

Median:

$\approx \lfloor \mu + 1/3 - 0.02/\mu \rfloor$

Mode:

$\lceil \mu \rceil - 1, \lfloor \mu \rfloor$

Variance:

μ

Skewness:

$\frac{1}{\sqrt{\mu}}$

Kurtosis:

$\frac{1}{\mu}$

See the nomenclature (page [xxiii](#)) for the definition of $\lceil y \rceil$ and $\lfloor y \rfloor$. In those cases where x is an integer number (e.g., the number of events), $\lfloor \cdot \rfloor$ can be removed.

General Description The Poisson distribution plays an important role in many different scientific fields. It is obtained from the binomial distribution in case that the sample size is very large, i.e., $n \rightarrow \infty$, where, at the same time, the probability for Success tends to zero, $P \rightarrow 0$, such that nP stays constant. A vanishing probability for the event Success means that this event happens only very rarely. This is why the Poisson distribution is the distribution of choice whenever it comes to *rare events*. The main application is to help to count such events.

In general, Poisson distributions model events that are typically occurring at discrete points in time. There, the underlying assumption is that any event occurs statistically independent of any previous event. Examples for such cases are the number of cars passing by a checkpoint or the number of phone calls received. These examples (and those, presented below in the “Applications” paragraph) have in common that all events represented by the random variable are the result of a *Poisson process*. As this is an important concept, we now introduce it briefly.

Poisson Process One of the most important random processes¹ is that of the Poisson process, which is also called *Poisson point process*. This describes situations in which random events (i.e., “points”) occur in space and in time. There, the occurrence of one event does not affect the probability of other points or events, which is also called the *memoryless property*. The most important properties of such random processes can be summarized as follows:

- Any two subsequent events in time occur independent from each other.
- In any time interval, the average number of events that occur is proportional to the size of the interval; there is no dependency on where or when these events occur.
- The probability to find exactly one event in an time interval $(t, t + \Delta t)$ is $\lambda(\Delta t)$; the probability to find two or more events in this interval is zero.

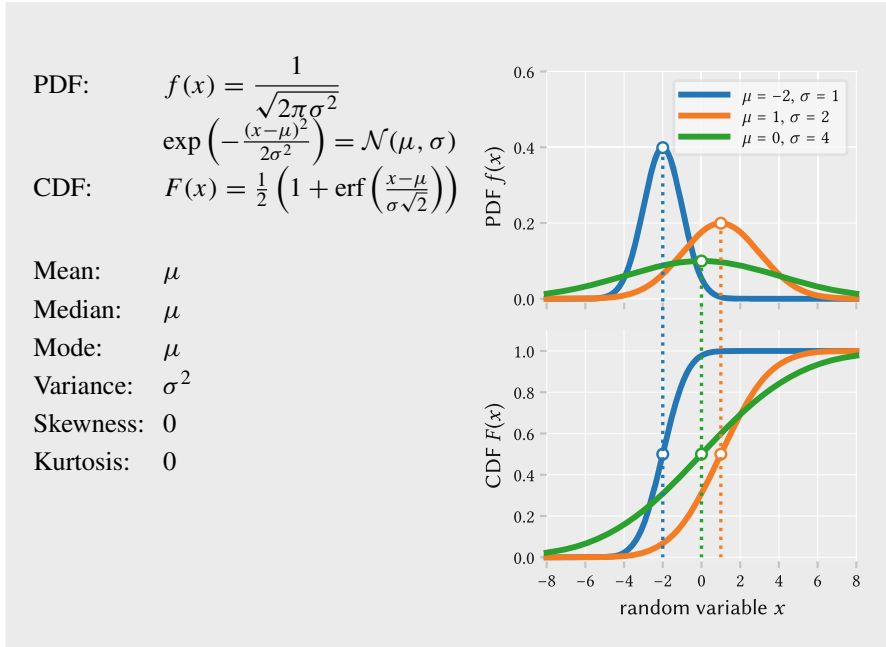
The variable $\mathbf{X} = (X_1, X_2, \dots)$ denotes the inter-arrival time of events, i.e., the time Δt between two events. For a given time span, the number of occurring points is taken as the random variable, which is why the Poisson process is in fact a counting process. The distribution of the time between any two subsequent points is given by the Poisson distribution.

Application in Science and Engineering A surprisingly large number of applications exists: whenever it is about counting the number of independent events occurring during a certain random process, then the odds are that it is related to Poisson distribution. For example, the number of scratches per area found on a specimen’s surface or the number of radioactive particles per volume follows a Poisson distribution. Poisson processes are also at the heart of percolation processes, as encountered in, e.g., fracture processes.

¹ Recall that a *random process* is a mechanism that generates random events with certain properties.

This distribution is also used to find the probability of a rare event to happen during a particular period of time, a certain length, area, or volume. An example would be a particle detector with the purpose of counting events.

10.7 Normal Distribution



A normal distribution is given by two parameters, the location μ and the scale σ . It is often abbreviated by the calligraphic “N”, $\mathcal{N}(\mu, \sigma)$. μ is also the mean or, in the context of probabilities, the expectation value, σ is the standard deviation, and σ^2 is the variance. The normal distribution is also known as *Gaussian distribution* because it consists of the *Gaussian function*, i.e., $f(x) = a \exp\left(-\frac{(x-b)^2}{2c^2}\right)$. In the above equation for the cumulative distribution function, erf is the error function defined as

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp(-t^2) dt, \quad (10.4)$$

which is a sigmoid function frequently occurring in statistics as well as in deep learning as an activation function. The PDF can also be characterized in terms of

moments: the 0-th moment is the total probability ($=1$). The first raw moment is μ ; the first central moment is 0. The second non-central moment is $\mu^2 + \sigma^2$, the second central moment is σ^2 .

Generally speaking, if your data is symmetrical, bell-shaped, centered, and unimodal, then it is very likely that a normal distribution is given. The normal distribution is very well understood; many data distributions that we encounter follow a normal distribution but not all of them do (e.g., the IQ's of people follows a normal distribution, while their income fits an exponential function).

The normal distribution is ubiquitous, and therefore it also hides in many mathematical aspects of inferential statistics and the calculation of probabilities. Many of their methods assume that a normal distribution is given. This implies that if the data is *not* normally distributed, then one has to be very careful since the methods could lead to wrong conclusions. A good “counter-example” is the *Gaussian naive Bayes* classification method—a ML method that requires the training data to be normal distributed; however, the method turns out to be surprisingly robust even in cases when the data is not normal distributed.

The normal distribution has a large number of properties; here we only list some of the most important ones:

- The “width” of the normal distribution is defined as the distance between the two points where the function values are $\text{MAX}(f) \cdot 1/\sqrt{e}$
- $\approx 68\%$ of the total area under the Gaussian curve (which represents the probability) lies in the range $[\mu - \sigma, \mu + \sigma]$. In other words, there is a 68% chance for obtaining a value of x that is located within one standard deviation from the mean μ
- The two parameter μ and σ can be used to shift and scale the Gaussian distribution. This results in the “standardized” form of the distribution:

$$f(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}.$$

Application in Science and Engineering Many properties in nature are normal distributed, such as people's height, their IQ scores, or body temperatures. Furthermore, it can be observed that the properties of many samples if plotted in a histogram take the shape of the normal (or Gaussian) distribution. We already encountered this in the context of the *central limit theorem*, Sect. 8.6.

Python/SciPy Python Listing 10.1 shows how the PDF and CDF can be obtained from Python's `scipy` library.

```

1 import numpy as np
2 from scipy import stats
3 import matplotlib.pyplot as plt
4
5 x = np.linspace(-8, 8, 200)
6 mu, sigma = -2, 1
7 rv = stats.norm(loc=mu, scale=sigma)
8
9 plt.plot(x, rv.pdf(x), label='PDF')
10 plt.plot(x, rv.cdf(x), label='CDF')
11 plt.legend()

```

Python Listing 10.1 A minimal example for plotting the PDF and CDF of a normal distribution

Drawing, e.g., 50 random numbers that follow the probability given by a normal distribution can be then done with `rv.rvs(50)` (after the random variable has been create by `rv = stats.norm(loc=mu, scale=sigma)`; see Python Listing 10.1).

10.8 Bivariate Normal Distribution

Parameters: standard deviation σ_1, σ_2 ,
and mean μ_1, μ_2

PDF:
$$f(x_1, x_2) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{(x_1-\mu_1)^2}{2\sigma_1^2} - \frac{(x_2-\mu_2)^2}{2\sigma_2^2}\right)$$

 $= \mathcal{N}(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2)$

CDF: (usually constructed numerically)

Mean: (μ_1, μ_2)

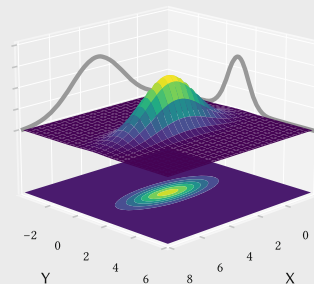
Median: (μ_1, μ_2)

Mode: (μ_1, μ_2)

Variance: σ_1^2 and σ_2^2 along
 x_1 and x_2 direction

Skewness: 0

Kurtosis: 0



The bottom plane shows the projection into the x - y - plane; the two line plots show the contours of the marginal distributions.

The bivariate normal distribution is a normal distribution in two random variables. It is a special case of the general multivariate normal distribution, introduced in Sect. 10.9, but it can be still written out without having to use linear algebra and the covariance matrix. The main idea is that using the identity $\exp(a + b) = \exp(a)\exp(b)$, the bivariate normal distribution can be decomposed into a product of univariate normal distributions:

$$\mathcal{N}(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2) = \mathcal{N}(x_1; \mu_1, \sigma_1) \mathcal{N}(x_2; \mu_2, \sigma_2) \quad (10.5)$$

Therefore, for general aspects we refer the reader to the respective sections on the univariate and multivariate normal distribution. Here, we only explain aspects pertaining to this bivariate distribution.

Additional Details In science, this function is of some importance as it is a function in two variables, which therefore can still be easily visualized. For example, its isocontour lines can be used to highlight the quartile of all normally distributed data points that are located within a certain region. The isocontour lines can either be plotted (e.g. using `matplotlib` this can be done using the function `matplotlib.contour`), or these lines can be mathematically computed; see below.

Isolines of a Bivariate Normal Distribution Visualizing bivariate normal distributions can, e.g., be done by showing lines that connect points of equal values. In ***Appendix B.4.3, it is derived that the points that have the same values c are located on contour lines of elliptical shape. As a result the two semi-axes of the ellipse are given by

$$a = \sqrt{2\sigma_1^2 \ln\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)} \quad \text{and} \quad b = \sqrt{2\sigma_2^2 \ln\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)}. \quad (10.6)$$

The major semi-axis is given by the larger of the two values, $\max(a; b)$, while the minor semi-axis is $\min(a; b)$. With these abbreviations the equation governing the elliptical isolines is

$$\frac{(x_1 - \mu_1)^2}{a^2} + \frac{(x_2 - \mu_2)^2}{b^2} = 1, \quad (10.7)$$

where μ_1 and μ_2 shift the center of the ellipse along the two direction. Thus, this equation describes all constant values c of the PDF $\mathcal{N}(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2) = c$.

As a demonstration, we now create a dataset, for which we will show how an elliptical isoline can be computed. The dataset is obtained by sampling a normal distribution, $X_1, X_2 \sim \mathcal{N}$, which is defined through the mean and the 2×2 covariance matrix.

```
In [1]: import numpy as np
        from scipy.stats import multivariate_normal
```

```
In [2]: true_mean = np.array([5, 3], dtype=float)
        true_cov = np.array([[2., 0.], [0., 0.3]], dtype=float)
        rv = multivariate_normal(mean=true_mean, cov=true_cov)
        X = rv.rvs(size=2000)
```

We then compute the sample mean, which is used to define the center of the ellipse, and the sample covariance matrix, which is used to obtain the standard deviations.

```
In [3]: sample_mean = np.mean(X, axis=0)
        sample_cov = np.cov(X, rowvar=False)
        mu1, mu2 = sample_mean
        sigma1, sigma2 = np.sqrt((sample_cov[0,0], sample_cov[1,1]))
```

The next two cells implement the above equations for the ellipse:

```
In [4]: def ellipse(x1, mu1, mu2, a, b):
        y = np.sqrt((1 - ((x1 - mu1) ** 2) / (a ** 2)) * (b ** 2))
        x1 = np.hstack((x1, x1[::-1], [x1[0]]))
        return x1, mu2 + np.hstack((y, -y, y[0]))
```

```
In [5]: c = 0.01
        fac = np.log(1 / (2 * np.pi * c * sigma1 * sigma2))
        a = np.sqrt(2 * sigma1 ** 2 * fac)
        b = np.sqrt(2 * sigma2 ** 2 * fac)

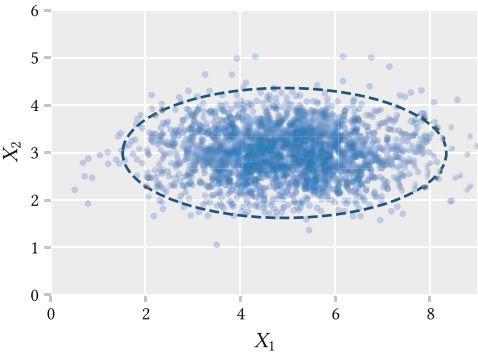
        x1 = mu1 + 0.999 * np.linspace(-a, a, 200, )
        x1, x2 = ellipse(x1, mu1, mu2, a, b)
```

There, the `np.hstack` functions are used to create data of the upper and lower half of the ellipse in a format suitable for plotting.

```
In [6]: fig, ax = plt.subplots()
        ax.plot(X[:, 0], X[:, 1], 'o', mfc='C0', mec='none', alpha=0.3)
        ax.plot(x1, x2, '--')
        ax.set_aspect('equal')
        ax.set(xlim=(0, 9), ylim=(0, 6), xlabel='X1', ylabel='X2');
```

The resulting figure is similar to the one shown in Fig. 10.1. Note that the covariance matrix of the dataset `x` might have small, non-zero off-diagonal entries. This would imply that the ellipse, that fits best to the data, would be slightly rotated. This is, for simplicity, ignored here. The next subsection introduced the most general

Fig. 10.1 Elliptical contourline of a bivariate normal distribution. The parameter of the normal distribution function was obtained from computing the mean and the covariance matrix of the dataset shown as blue markers



representation of multivariate normal distributions and will further elucidate the link to the covariance matrix.

10.9 Multivariate Normal Distribution

PDF:

$$f(\mathbf{X}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}} \frac{1}{(\det(\boldsymbol{\Sigma}))^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{X} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{X} - \boldsymbol{\mu})\right) = \mathcal{N}(\mathbf{X}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$$

Parameter:

covariance matrix $\boldsymbol{\Sigma}$
vector of mean values $\boldsymbol{\mu}$
dimensionality D

Mean:

$\boldsymbol{\mu}$

Median:

$\boldsymbol{\mu}$

Mode:

$\boldsymbol{\mu}$

Variance:

e.g., $\text{Var}(X_i, X_i)$ from covariance matrix

Skewness:

0

Kurtosis:

0

This is a distribution in multiple random variables. The notation benefits particularly from the vector and matrix notation (e.g., $\boldsymbol{\mu}$ is a vector of the mean values along all directions, and $\boldsymbol{\Sigma}$ is the population covariance matrix). In all generality, the multivariate normal distribution is not just a superposition of normal distributions along the two EUCLIDian coordinate axes but additionally might be subjected to

a rotation. Using the covariance matrix as introduced in Sect. 8.7.2 is a particular concise representation.

As for the univariate normal distribution, the calligraphic letter $\mathcal{N}$ is used to indicate the normal distribution, e.g., $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$. The above formulation is not just a formulation in arbitrary dimensions (i.e., arbitrary number of random variables), but it also allows to represent normal distributions in transformed coordinate systems, which, e.g., in two dimensions can be a rotation for the distribution about its center.

10.10 The Relation Between Covariance Matrix and Multivariate Normal Distribution

At the heart of the multivariate normal distributions is the covariance matrix which is introduced in details in Sect. 8.7.2. It represents correlations between variables in an average sense and thereby elegantly connects data samples from a normal distribution with some of their statistical properties that are represented through the shape of the distribution.

In many ML methods such as the principal component analysis, the Gaussian naive Bayes method, or the Gaussian mixture model, normal distributions play an important role; this is the reason why we dedicate a later part to this subject. In the following we will explore some aspects of the covariance matrix that are strongly related to concepts from linear algebra as introduced in Appendix A; these will be used to show that the eigenvectors of the data covariance matrix are the directions of the axes of the corresponding ellipse or ellipsoid.

For simplicity, we consider the case of only two random variables, $X_1, X_2 \sim \mathcal{N}(X_1, X_2; \mathbf{S}, \boldsymbol{\mu})$, where $\mathbf{S}$ is the *data covariance matrix*²

$$\mathbf{S} = \frac{1}{n-1} \bar{\mathbf{X}}^T \bar{\mathbf{X}}, \quad (10.8)$$

where $\bar{\mathbf{X}}$ is the centered data matrix which has zero mean values. $\mathbf{S}$ is then a 2×2 matrix with the components

$$\mathbf{S} = \begin{bmatrix} S_{11} & S_{12} \\ S_{12} & S_{22} \end{bmatrix} \quad (10.9)$$

where the diagonal terms are the squared standard deviations along the two Euclidean axes, and $\mathbf{S}$ is symmetric. For this 2×2 matrix, we can compute the eigenvalues from the characteristic polynomial Eq. (A.58) which results in

² Reminder: the data covariance matrix $\mathbf{S}$ is an estimate of the “true” covariance matrix $\boldsymbol{\Sigma}$. The latter characterizes the distribution function from which the data was sampled, cf. Eq. (8.9).

$$\lambda_{1/2} = \frac{\text{tr}(\mathbf{S})}{2} \pm \sqrt{\frac{\text{tr}^2(\mathbf{S})}{4} - \det(\mathbf{S})} = \frac{S_{11}^2 + S_{22}^2}{2} \pm \sqrt{\frac{(S_{11} - S_{22})^2}{4} - S_{12}^2}, \quad (10.10)$$

where $\det(\mathbf{S}) = S_{11}S_{22} - S_{12}^2$. Without loss of generality, we assume that $\lambda_1 \geq \lambda_2$. Following Appendices A.3.1 and A.3.2, we then also can obtain eigenvectors $\mathbf{u}_1$ and $\mathbf{u}_2$:

$$\mathbf{u}_1 = \begin{bmatrix} 1 \\ (-S_{11} + \lambda_1)/S_{12} \end{bmatrix} \quad \text{and} \quad \mathbf{u}_2 = \begin{bmatrix} -1 \\ (S_{11} - \lambda_2)/S_{12} \end{bmatrix}, \quad (10.11)$$

which in the following we assume to have unit length, i.e., they are scaled as $\mathbf{u}_1 \leftarrow \mathbf{u}_1/\|\mathbf{u}_1\|$ and $\mathbf{u}_2 \leftarrow \mathbf{u}_2/\|\mathbf{u}_2\|$. With the help of Eq. (A.62) the data covariance matrix $\mathbf{S}$ can be diagonalized:

$$\mathbf{D} = \mathbf{P}^{-1}\mathbf{S}\mathbf{P} \quad (10.12)$$

where $\mathbf{D}$ is a diagonal matrix that has the eigenvalue on its main diagonal

$$\mathbf{D} = \text{diag}(\lambda_1, \lambda_2) = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \quad \text{with} \quad \lambda_1 \geq \lambda_2, \quad (10.13)$$

and the columns of $\mathbf{P}$ consist of the two unit eigenvectors

$$\mathbf{P} = [\mathbf{u}_1 \ \mathbf{u}_2]. \quad (10.14)$$

According to the spectral theorem Appendix A.3.4, these eigenvectors are mutually orthogonal, and hence, $\mathbf{P}$ is an orthogonal matrix with $\mathbf{P}^{-1} = \mathbf{P}^T$. It follows that $\mathbf{D}$ governs the possibly anisotropic “stretch” of a vector while $\mathbf{P}$ governs a rotation of a vector. This holds also for a bivariate normal distribution: $\mathbf{S}$ transforms this distribution by a stretch and a rotation. The angle of rotation in this two-dimensional example can be obtained by comparing the elements of $\mathbf{P}$ with a rotation matrix: there, it is $\theta = \cos^{-1}(P_{11})$ where P_{11} denotes the element in the first row and column of $\mathbf{P}$.

This result can be generalized: the covariance matrix is decomposed into orthogonal and diagonal matrices by means of eigenvalues and eigenvectors. This can be used for finding the isocontours of arbitrary, multivariate normal distributions: the isocontour lines are formed by ellipsoids in three dimensions or by hyper-ellipsoids in case of more than three dimensions. In the case of two dimensions, the isocontours are *isolines* forming ellipses as discussed in Sect. 10.8. Those points located on the contour line or (hyper-)surfaces, respectively, are the lines of constant density values c and therefore have equal likelihood to be drawn. The ellipsoids are given by

$$(\mathbf{X} - \boldsymbol{\mu})^T \mathbf{S}^{-1} (\mathbf{X} - \boldsymbol{\mu}) = c^2 \quad (10.15)$$

Mahalanobis Distance ?

Note that the quadratic term on the left hand is called the *squared Mahalanobis distance*. The Mahalanobis distance is a useful and intuitive measure: a value of two indicates a distance of two standard deviations away from the mean.

The ellipsoids are centered at $\boldsymbol{\mu}$, and their axes are parallel to the directions of the eigenvectors of $\mathbf{S}$. In the bivariate special case, the major and minor semi-axes have the lengths $c\sqrt{\lambda_{1/2}}$ where λ_1 and λ_2 are the two eigenvalues of $\mathbf{S}$.

Example The following example demonstrates how the above results can be used to show how the covariance matrix determines details of the distribution. We start by creating a data set from sampling a given normal distribution:

```
In [1]: import numpy as np

true_mean = np.array([3, 1], dtype=float)
true_cov = np.array([[0.8, 0.3],
                    [0.3, 0.2]], dtype=float)
X = np.random.multivariate_normal(true_mean, true_cov, size=800)
```

The prescribed covariance and mean are the “true” quantities of the population. Next, the sample mean and sample covariance are computed from the data—for a large enough sample size, they should be only slightly different from the true values and serve as an estimate for them. Additionally,

```
In [2]: sample_cov = np.cov(X, rowvar=False)
sample_mean = np.mean(X, axis=0)
```

Now we can compute eigenvectors. For this, the numpy function `eig` is used which returns eigenvalues and -vectors:

```
In [3]: eigenvec1 = np.linalg.eig(sample_cov)[1][:, 0]
eigenvec2 = np.linalg.eig(sample_cov)[1][:, 1]
```

We have now obtained eigenvectors that are aligned with the axes of an ellipse that would represent isocontour lines of the PDF. The sample data along with the directions of the two eigenvectors can now be visualized:

```
In [4]: import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.set(aspect='equal', xlim=(-1, 7), ylim=(-1.5, 3.5),
       xlabel='$X_1$', ylabel='$X_2$')
ax.scatter(X[:,0], X[:,1], s=10, alpha=0.3, edgecolors='none')

xa, xb = np.linspace(-1, 1), np.linspace(-0.2, 0.2)
ax.plot(xa + sample_mean[0],
        eigenvec1[1] / eigenvec1[0] * xa + sample_mean[1], c='0.3')
ax.plot(xb + sample_mean[0],
        eigenvec2[1] / eigenvec2[0] * xb + sample_mean[1], c='0.3')
```

The resulting plot is shown in Fig. 10.2. There, the two dashed lines show the directions of the eigenvectors; they intersect at the sample mean, which is observed to be very close to the true mean. From the eigenvectors, we can compute the rotation θ of the data resulting from the cross-covariance terms as $\theta = \tan^{-1}(u_{1y}/u_{1x})$ where u_{1x} is the first component of the first eigenvector and u_{1y} is its second component. We observe that the direction of the largest and smallest “spread” of the data coincide with the directions of the eigenvectors; a fact that we will heavily rely on in Sect. 15.2, when we derive the method of *principal component analysis* (PCA) from this.

Note that this is just a basic visualization where we compute the inclination of the two lines from the components of the eigenvectors. Here, the length of the two lines is chosen for visual purposes and does not reflect, e.g., the magnitude of the eigenvalues.

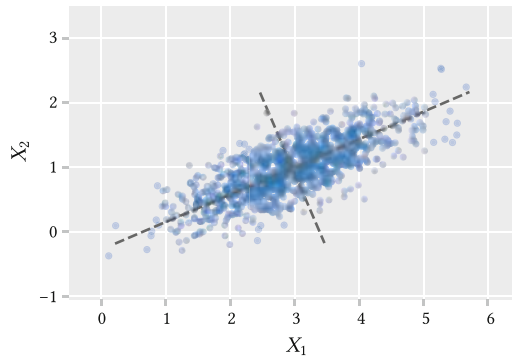
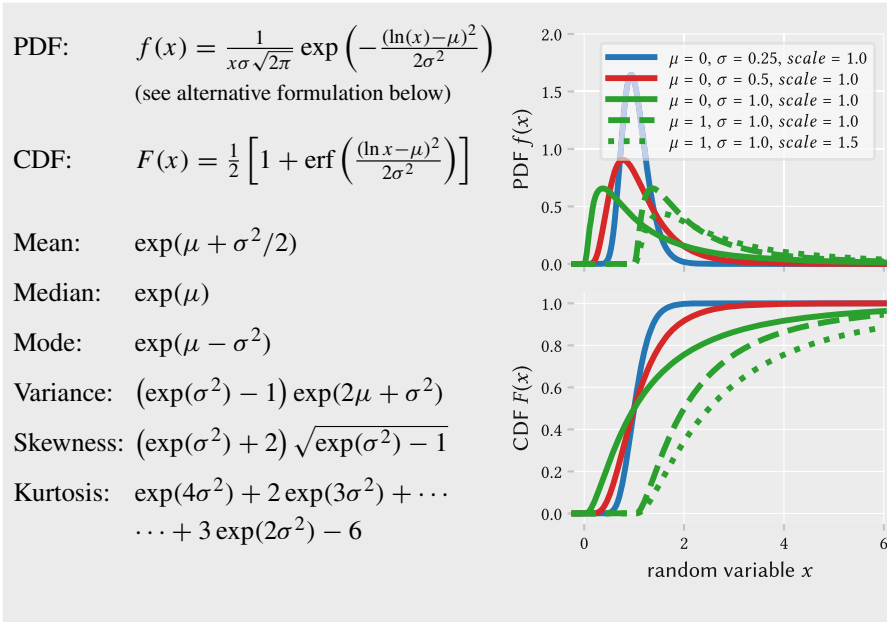


Fig. 10.2 The markers show samples from a given bivariate normal distribution. The directions of the eigenvectors of the data covariance matrix are indicated by the two dashed lines. The eigenvectors are aligned with the semi-axes of the ellipse which represents the isovalues of the corresponding PDF. Note that the lines’ lengths are not up to scale

10.11 Lognormal Distribution



General Description A lognormal distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. As a consequence, if the random variable X is log-normal distributed, then $Y = \log(X)$ has a normal distribution. Likewise, if Y follows a normal distribution, then $X = \exp(Y)$ is characterized as a log-normal distribution. As opposed to the normal distribution, the lognormal distribution is strongly skewed and has a long tail; additionally, it is defined for positive value of the random variable only. In situations that are related to waiting times between events, this is useful as time cannot become negative.

Words of Advice

In case that your data exhibit a truly fat tail, then the lognormal description is not appropriate to represent your data. Rather, power law-type distributions are more suitable as, e.g., used for describing strain burst statistics of dislocations in intermittent plasticity [1, 2].

There is an alternative formulation for the lognormal PDF

$$f(x; \tilde{\mu}, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2} \frac{(\ln(x/\tilde{\mu}))^2}{\sigma^2}\right) \quad (10.16)$$

which is obtained by the transformation of $\mu = \log(\tilde{\mu})$. The advantage of this formulation is that it becomes visible that $\tilde{\mu}$ is the scale of the distribution, which can be useful information in view of a particular physical problem. Additionally, this formulation has the beneficial aspect that the argument of the log function is dimensionless. A convenient side effect of the fact that the standard deviation shows up in the formulation of the mean is that the equation can be easily used to estimate the two distribution parameters from a dataset: from computing the mean follows μ , and with this we obtain σ from the mean of the lognormal distribution; no further “curve fitting” is required.

Python/SciPy SciPy has an implementation of Eq. (10.16) where the above $\tilde{\mu}$ is the location of the function. The first cell creates a lognormal random variable “object” that in the second cell can then be used to create the PDF. Additionally, we also create 800 lognormal distributed random numbers:

```
In [1]: import numpy as np
        from scipy.stats import lognorm

        mu = 1
        sigma = 0.85

        # create a continuous lognorm RV
        rv = lognorm(sigma, mu)

        # generate lognorm distributed random values
        r = lognorm.rvs(sigma, loc=mu, size=800)
```

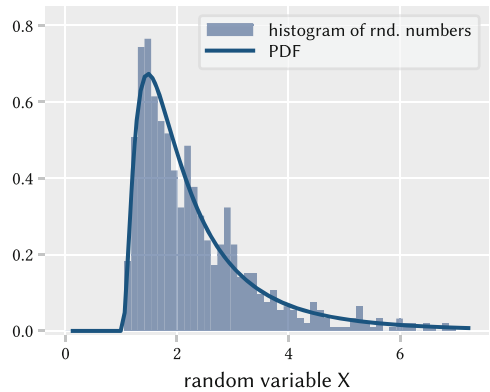
Plotting the PDF and a histogram of the random numbers, we observe that they agree (note, the parameter `density=True`, used to ensure that the total area equal to one):

```
In [2]: fig, ax = plt.subplots(dpi=200)
        x = np.linspace(lognorm.ppf(0.01, sigma),
                        lognorm.ppf(0.99, sigma), 100)
        ax.hist(r, bins=np.linspace(0, 7, 60), density=True);
        ax.plot(x, rv.pdf(x))
```

The resulting plot is similar to the one shown in Fig. 10.3.

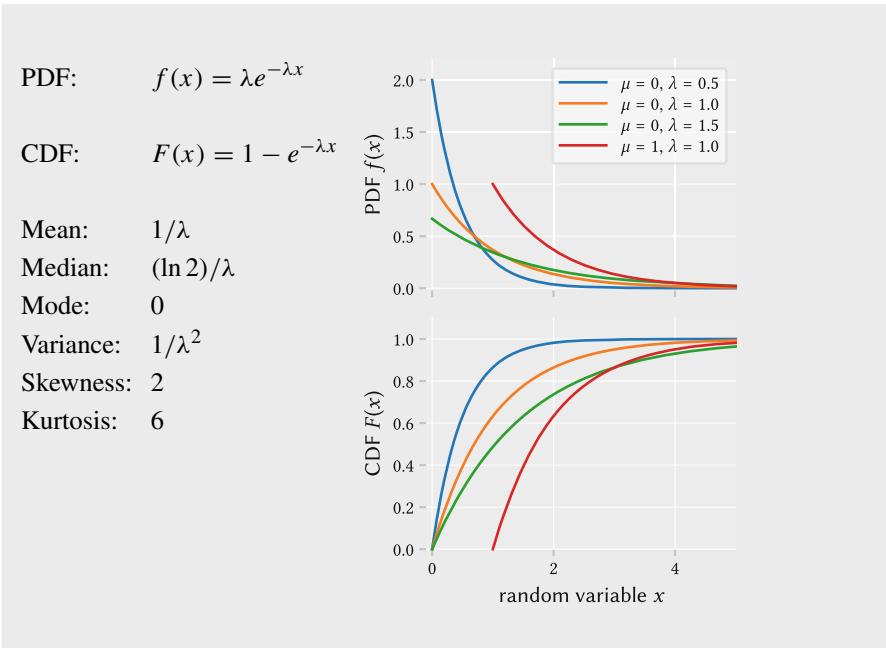
Application in Science and Engineering The lognormal distribution can be encountered in many engineering and scientific applications. For example, it is used to model the lifetime of specimens during fatigue conditions, the particle size distribution of aerosols or metal powders, or the reliability of electronic devices.

Fig. 10.3 Continuous PDF and histogram of lognormal distributed random numbers



Furthermore, many growth phenomena are also well approximated by a lognormal distribution, such as grain growth. A related distribution is the Weibull distribution.

10.12 Exponential Distribution



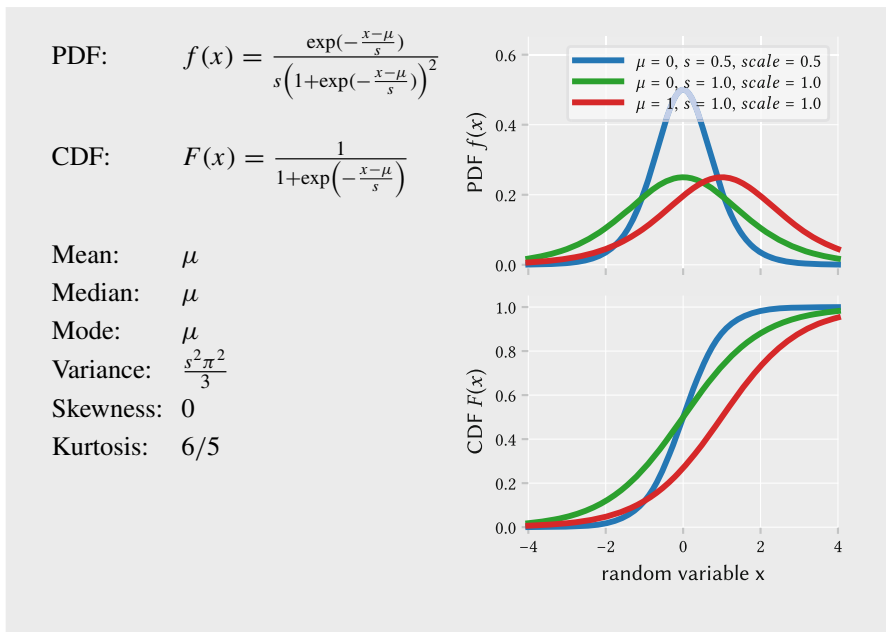
General Description We encountered the exponential distribution already in the context of the POISSON point process where it is the probability distribution of

the time between subsequent events. There, this function is also described as being “memoryless”: any successful event should not influence the outcome of another successful event, or the waiting time until an event occurs does not depend on how much time has elapsed already. There exist only two distribution types that are “memoryless”: the only distributions that are “memoryless” are exponential distributions of non-negative real numbers and the geometric distributions of non-negative integers.

Python/SciPy See the Python example for the lognormal distribution in Sect. 10.11. All that needs to be done is to replace `lognorm` with `expon` and adjust the respective parameters accordingly.

Application in Science and Engineering Typical applications are failure, lifetime prediction, and reliability engineering (also see the respective section of the Poisson distribution, Sect. 10.6).

10.13 Logistic Distribution



General Description The logistic distribution looks somewhat similar to a normal distribution—it also is symmetric and looks bell-shaped—however, it has a larger variance as well as a larger kurtosis. The latter is a consequence that the logistic

distribution has a heavier tail than the normal distribution. The **CDF** is the *logistic function*, which appears in logistic regression and feedforward neural networks.

Python/SciPy See the Python example for the lognormal distribution in Sect. 10.11. All that needs to be done is to replace `lognorm` with `logistic` and adjust the respective parameters accordingly.

Application in Science and Engineering This distribution occurs in semiconductors and in metals in the context of electron transport properties. There, it has the same functional shape as the derivative of the Fermi function. However, the most important application is in the context of machine learning methods.

References

1. M.-C. Miguel, A. Vespignani, S. Zapperi, J. Weiss, and J.-R. Grasso. Intermittent dislocation flow in viscoplastic deformation. *Nature*, 410(6829): 667–671, Apr. 2001. DOI <https://doi.org/10.1038/35070524>.
2. Q. Rizzardi, P. M. Derlet, and R. Maaß. Intermittent microplasticity in the presence of a complex microstructure. *Phys. Rev. Mater.*, 6:073602, Jul 2022. DOI <https://doi.org/10.1103/PhysRevMaterials.6.073602>.

Part III

Classical Machine Learning

Introduction and General Concepts of Machine Learning and Data Science

11

Errors using inadequate data are much less than those using no data at all.

Charles Babbage (1791–1871)

English mathematician, engineer, and inventor

11.1 The Definition(s) of Machine Learning

The notion of *machine learning* (ML) is often used in a somewhat mystifying manner, and often no difference is made between the notion of ML and *artificial intelligence* (AI); ML often is synonym with a computer code that is able to perform in a super-human fashion and that is able to make intelligent decisions based on some data which, e.g., the code or machine might even have collected itself. Clearly, neither is any of this possible today nor a definition that is helpful for our understanding how machines learn and why this might be useful for data analysis and data mining in science and engineering.

When we take a look at the different types of algorithms and the vast ranges of applications that are relevant to the natural sciences or engineering, it becomes clear that a systematic approach is needed, starting at the very beginning. What better approach could there be than simply starting with a definition of ML?

As it turns out, writing down such a definition is not completely trivial because there is not *the* generally agreed-upon definition of what ML is. It is, however, quite illustrative to see some of the commonly quoted definitions:

Some Definitions and Descriptions of Machine Learning



The field of study that gives computers the ability to learn without being explicitly programmed.

ARTHUR SAMUEL (1959) coined this quote—originally in a slightly different version in [6]

A computer program is said to learn from experience E with respect to some class of task T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

TOM M. MITCHELL (1997) in the book “Machine Learning” [2]

The purpose of machine learning is finding patterns through the use of computer algorithms. Searching for patterns in data is a fundamental one and has a long and successful history.

CHRISTOPHER B. BISHOP (2011) in the book “Pattern Recognition and Machine Learning” [1]

Machine Learning is about extracting knowledge from data. It is closely related to statistics and optimization but is very focused on prediction.

ANDREAS C. MÜLLER (2017) in “Introduction to Machine Learning with Python” [3]

These are four quite different definitions. ARTHUR SAMUEL coined the first definition—originally in a slightly different version in the context of playing checkers against a computer [6]. His definition emphasizes that the computer can learn, i.e., become better at something. Mathematically speaking this involves measuring the performance, e.g., by a mathematical function. In the next chapter, the *cost function* is introduced for that purpose, which might already be known from mathematical optimization theory. The fact that performance gain happens without *being explicitly programmed* for the task at hand signifies that learning is data-driven and not only algorithmically determined, e.g., by writing down a list of highly specific instructions.

TOM. T. MITCHELL's well-known definition takes this one step further by formalizing the previous definition. Additionally, as a new aspect, the notion of performance is introduced. However, this definition might need a bit of explaining: *experience E* is related to processing some given (training) data; the occurrence of a specific *task T* indicates that the learning process is tailored to a particular problem, e.g., the prediction of temperature based on humidity and the number of the calendar week. The *performance P* quantifies how well the program performs, which is related to either an error or a performance measure, as above. For example, this could be the mean value of all deviations between a “correct” value and the value predicted by the algorithm. Machine learning is then the process, when the performance for a particular problem is getting better as more and more input data are provided, which is what “experience *E*” means. This definition is also closely oriented to how humans learn. For example, when we were still young, we were shown numerous images of cats and dogs. The more images we were shown, the better we were able to differentiate between dogs and cats—without explicitly being taught what makes a dog a dog.

At first glance the definition by CHRISTOPHER BISHOP focuses mainly on data-mining aspects: finding *patterns*, i.e., structure in data. One might think that while this is an important task for ML, it neglects other, equally important tasks, namely, training of the ML model (the “learning” process) and prediction. However, being able to automatically detect “patterns”, i.e., relations between seemingly unrelated data, is already a really big challenge: when the Babylonians looked at the sky and could identify the paths of the planets, they detected a pattern in the motion. An algorithm, which is able to perform such a task independently, clearly would seem quite advanced. A “pattern” can also mean different aspects of a cat's anatomy, which define how to identify a cat. Thus, “pattern” is a very broad notion that fits well with many different machine learning problems, while the training process is still of no great importance here.

The definition by ANDREAS C. MÜLLER is written from a perspective of information science: recall the DIKW pyramid, Sect. 2.3, which is a visualization of the concept, in which the value of the (raw) data is increased by turning it into *knowledge*, e.g., by extracting structural information about the data. The definition also includes that ML is intrinsically a multidisciplinary topic as it draws heavily on statistics and mathematical optimization. However, the aspect of making predictions is much more important than in (descriptive) statistics focusing on the description of the characteristics of data distributions. This definition also includes the second key aspect of ML in addition to making predictions, i.e., making inferences: based on data processing, the goal is to find out what the underlying relations and the “inner workings” are that created the data. Such an “underlying relation” could be, e.g., for the case gravitational force, a quadratic time-position relation.

In this book we will adopt the following definition, which captures a number of aspects from the above definitions:

Definition 11.1 (Machine Learning). Machine learning is the process in which ...

1. a computer program learns to find and extract patterns in given data;
2. the program makes predictions on new but somewhat related data, based on what was learned in step 1;
3. the prediction accuracy in step 2 increases, the more data the program is shown in step 1.

The detection of *patterns* within data refers to the identification of structure in data. Such a structure is, e.g., to identify that all data roughly follow a linear trend, which could be represented by a straight line segment. The “essence” of what the machine learned consists of the two points that govern the line segment (or the inclination and intercept). This is extremely condensed data compared to the possibly hundreds or thousands of original data points. Finding such a line is what a machine learning *regression* algorithm does.

Another example would be to detect regions of points that belong to the same “class,” for example, in a microscopy image, we might want to differentiate between two phases such as austenite and martensite. The ML algorithm is supposed to automatically find these contiguous regions of pixels and decide if they belong to the first phase or to the second one. In one of the simplest approaches, the geometry of the regions could be given by two rectangles. Hence, in our example, we describe each of the two phases by a geometrical shape, defined by only two points. This is again a very strong reduction of the original dataset (the image pixel). This is done by machine learning *classification* algorithms: they assign labels (i.e., different categories, cf. Sect. 6.1) to input data and thereby *classify* them.

In both cases, the resulting *model*, i.e., the regression line or the class labels, can help to understand aspects about the underlying physical phenomena that created the data. From *understanding* results something that exists independently of the dataset: knowledge.

The aspect 2. in this definition tells us that ML is a tool for making predictions: if the program learned from a few data points how a stress-strain curve should look like (again, that would be the pattern), it would be able to predict the stress value for a new strain value that was not in the dataset.

At the same time, 3. implies that learning is the process of ingesting *training data*, during which the performance of a specific task improves. For example, adding more and more stress-strain data pairs should cover more and more regions of the stress-strain curve so that predictions eventually become very reliable.

11.2 How and What Do Machines Learn?

The question “How and what do machines learn?” has, at least from a bird’s eye view, already been answered in the previous section.

The remaining question is still: how does a program represent the relation between output and input? There are several ways to do this, and all of them strongly depend on the kind of output that an algorithm is supposed to provide.

11.2.1 The Machine Learning Model

The basic structure of a ML model is shown in Fig. 11.1. A machine learning model takes input data, does something to it, and produces output data. During the training process, some details of the model (the model parameters) are adjusted such that the output data matches the expectations (e.g., some already known output data). Once we decide that the model is well-trained, we can use it to make predictions, i.e., provide new input data and obtain the corresponding output data.

This is also the reason for introducing the simple equation $y = f(x)$ in Sect. 3.3 as *the* fundamental equation for this text. As we always operate with several data records, our preferred notation is

$$Y = h(X) \tag{11.1}$$

for a single input and output variable. Also, we prefer to call the model function as h , because in the next chapter, this will represent a ML “hypothesis.” For vector-values input and output variables, the above equation reads

$$\mathbf{Y} = h(\mathbf{X}) \quad \Leftrightarrow \quad (Y_1, \dots, Y_k) = h(X_1, \dots, X_n) . \tag{11.2}$$

Obviously, a lot of details can be hidden inside the model h . Sometimes it is even impossible to determine how the model works. Such a model is called a *black box model*: a model that takes some input and somehow turns it into output, while the internal “mechanisms” of this data processing are entirely unknown.

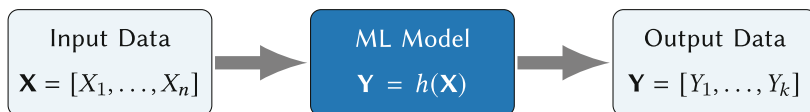


Fig. 11.1 A machine learning model is, from a bird’s eye view, very simple: it takes input data, does something to it, and produced output data. During the training process, the model is adjusted such that the output data matches the expectations (e.g., already known output data)

Clearly, if there is a black box model then there is also a white box model. And indeed, a *white box model* is a completely transparent model, of which the entire internal mechanisms are visible and could be, e.g., mathematically written down. A simple example would be a mathematical function h that maps any variable x to a point on a line given by the inclination m and the intercept y_0 :

$$h : X \mapsto Y = m \cdot X + y_0 . \quad (11.3)$$

The two parameter m and y_0 are the ones that need to be determined during the training process. Often, this is indicated by writing the function as $h(X; m, y_0)$, as introduced in Sect. 3.1.

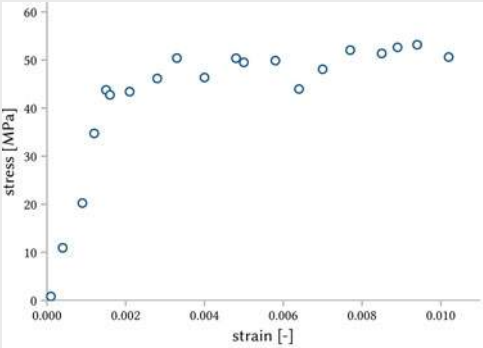
In the next subsection, one of the simplest possible learning algorithms is introduced as an example of a “model” that does not explicitly rely on a mathematical but rather on an algorithmic formulation.

11.2.2 A First Learning Algorithm: Instance-Based Learning

Assume the simple strategy that a program learns by simply “memorizing” the training data. To make predictions for a new instance, the memorized data is searched for the training instance that is most similar to the new instance, and this is used as the predicted datum. This procedure is called *instance-based learning* (or sometimes also *memory-based learning*). An example for a nearest-neighbor approach to regression is shown in the table in Example 11.1.

Example 11.1 (Instance-Based Learning of Stress-Strain Data)

A tensile test was performed on a specimen during which the axial force and displacement were measured (cf. dataset MDS-1 (tensile test)). Uniaxial strain and stress data were computed as shown in the table—these are the training instances. This is an example of only one input feature, i.e., the strain, and one output, i.e., the stress. Assume that we want to make predictions for a new input value, e.g., a strain value of $\varepsilon = 0.0042$. The nearest input value in the training dataset is the record ($\varepsilon = 0.0040$; $\sigma = 46.36$), and hence, the prediction of this instance-based learner is $\sigma = 46.36$.



The scatter plot displays the relationship between uniaxial strain (x-axis, ranging from 0.000 to 0.010) and stress (y-axis, ranging from 0 to 60 MPa). The data points, represented by blue circles, show an initial rapid increase in stress with strain, followed by a plateau and then a slight decrease at higher strain values. The data points correspond to the values in the table provided.

strain [–]	stress [MPa]
0.0001	–1.84
0.0004	10.93
0.0009	20.24
0.0012	34.73
0.0015	43.78
0.0016	42.75
0.0021	43.42
0.0028	46.16
0.0033	50.41
0.0040	46.36
0.0048	50.37
0.0050	49.51
0.0058	49.88
0.0064	43.97
0.0070	48.08
0.0077	52.05
0.0085	51.36
0.0089	52.63
0.0094	53.18
0.0102	50.61

Quantifying the meaning of *most similar* is straightforward, if all input values have the same unit. In such a case, the use of common distance metrics works well. It is then straightforward to compute the distance d between the new input data $\hat{x}$ and the input data of *all* training instances, e.g., as the Euclidean norm (which in 1D is just the norm of the difference between the training input variable and the given input variable). If the input data consists of n variables or features and $x = [x_1, \dots, x_n]$ is the input data for a specific training instance, the Euclidean norm for the new input data is

$$d = \sqrt{(x_1 - \hat{x}_1)^2 + \dots + (x_n - \hat{x}_1)^2}. \quad (11.4)$$

Computing these distances of the new input data to *all* training instances and finding the minimum value gives the best fitting data point that is then returned as the prediction for the new point. This methodology looks for the *nearest neighbor*.

The link to the *k*-nearest neighbors method



An extension of this method is the *k*-nearest neighbors (**kNN**) classification algorithm, which uses the *k* closest neighboring points to determine the predicted value. For example, in case of $k = 3$ neighboring, the result could be taken as the average value of these three points. This method will be discussed in more detail in Chap. 14.

The strategy of computing distances, however, is not always straightforward, as not all features (input variables) have the same relevance; they, therefore, might require some weighting. Furthermore, computing the Euclidean distance only works if all input data have the same units. For mixed units, a feasible approach is to perform dimensionless scaling and then, based on the dimensionless numbers, compute the distance between the new and the given input data. The scaling, however, necessarily includes a degree of arbitrariness. Knowledge of the physical nature of the data might give a good hint for determining reasonable scaling/weighting values; generally, this is a key problem in instance-based learning.

A characteristic of instance-based learning is that during the learning process, training data is only stored. Effectively there is no training process of the model, and no further processing of the data is done at this point. This is the reason why instance-based learning is also called *lazy learning*: only when it comes to making predictions, computations (e.g., evaluation of the distance metric) need to be performed, which can be a costly and time-consuming process. An advantage of lazy learning is that it is very easy to adjust the model to new training data: they are simply added to the already stored training data. However, data storage itself can be challenging if the datasets become larger.

Comparing this to the example of the function that represented data as a straight line in the previous subsection, a drawback of instance-based learning becomes visible: there is no explicit function as a representation of the underlying knowledge. The knowledge consists simply of the agglomerated data, while no further abstraction or complexity reduction of the data was performed.

11.3 Introduction of the General Machine Learning Workflow

We have now obtained some first impressions of how ML models work, and we have also learned a number of important notions. We will now broaden the picture and take a look at the whole ML workflow as shown in Fig. 11.2.

Step I is to obtain data, e.g., from experiments, simulations, or various online resources. This step ends when we are able to somehow read and import the data into the data analysis program, e.g., such that everything is ready for using Python or numpy with the data.

Subsequently, the preprocessing of the data in **step II** begins. This may consist of a number of chained manual or automatic processing steps. The preprocessing is supposed to remove any problems that may exist in the dataset (such as missing data, duplicates, etc.), but also to mathematically transform the data (e.g., by scaling or through dimensionality reduction) into the dataset that can then be directly used for the ML training process.

In **step III**, the type of ML model can be selected to be used for the subsequent training of the ML model. When choosing an appropriate model, it is important to consider, for example, whether the data distribution consists of continuous data for which a trend is to be learned, or whether it is categorical data that is to be divided into different classes.

As a prerequisite for learning in **step IV**, the (numerical) parameters of the model implementation and the model parameters must be selected. Parameters that are closely related to a model are also called *hyperparameters*. Then, the training process can be performed, which sometimes takes less than a second, but can also take several hours or even days (e.g., for large datasets with images with high resolution).

The resulting trained model has to be tested in various ways in **step V**, such as cross validation (see Sect. 16.2) to ensure that the model, in fact, learned generic features and can be used for new data as well. This is a particularly important step as this is often the only guard against an unsuccessful training process. If the “error analysis” (cf. Sect. 11.7 for an overview over error measures and how to compare datasets) reveals problems, then we should go back and repeat the training process

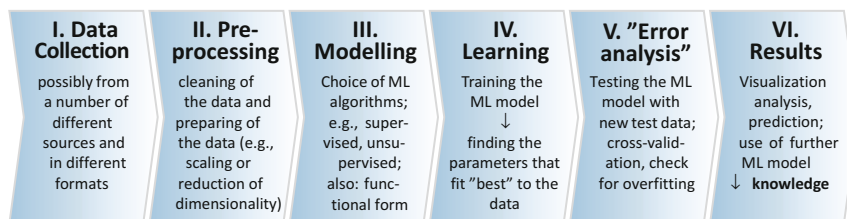


Fig. 11.2 The machine learning workflow consists of six main steps, starting with the data collection and ending at the exploitation of the results

of step IV, e.g., with more data or with different training parameters. Or we even might have to go back to step III and choose a different model.

Finally, in **step VI** we can use the results to make predictions or detect new patterns in the dataset or use the trained model to infer something about the underlying physics. All of this is ultimately served to increase our knowledge of the physics problem.

The following sections will discuss some of these steps in more detail.

11.4 Data Collection

Data collection can be a simple process, such as when an experiment has been performed, the data has been recorded and documented, and then only the appropriate files need to be imported.

However, in such a case data collection can also become very complex when the experiment needs to be tailored to the requirements of the **ML** method to be used. A typical bottleneck is not only the quality of the dataset (extremely noisy data can make **ML** training difficult), but often also the quantity, since not all experimental or simulation methods provide sufficient data. In these cases, one has to fuse datasets, e.g., from different experiments, possibly even performed by different people and on different machines. Very often this also requires additional work, e.g., for reading files in different formats or to understand the structure of the files in the first place.

Where did the data come from?



Even if an obtained dataset is well documented and ready to be used, it still might be helpful to recall how the data was acquired, e.g., during an experiment: At the beginning of this process, there is the physical phenomenon or property to be measured (e.g., temperature or force). The sensor (e.g., a thermistor or strain gauge) then samples this data in a continuous or time-discrete manner. In the next step, this is typically converted into an electrical signal, which might then get further processed by hardware or software (e.g., the signal might be filtered or amplified). In the final step, the signal is then converted into a digital representation, which is typically the point of first contact, when it comes to, e.g., statistical analysis of data.

Another way of collecting data is to search (e.g., in online resources) for training datasets created by someone else. Then, the data usually needs to be downloaded or queried, e.g., from a web page, web server, or repository. The data format can then be converted into a format that can be used directly by the data analysis software involved.

Thus, data typically used in statistical analysis or machines already have a long history of various types of conversions and processing; it is (almost) never the “raw”

data that we work with. It is also clear that any such dataset may contain a number of errors or variations.

11.5 Data Preprocessing

The goal of *data preprocessing* or *data preparation* is to transform the collected data into something that can be used directly for statistical analysis or ML. This is particularly important if the data was not created for the purpose of data mining or statistical analysis. We assume that the data has already been obtained and is now contained in a computer program in the form of a data structure (e.g., an array). Preparing the data is a task that can sometimes take a lot of time. A number of tasks are performed during this stage.

11.5.1 Data Cleaning

Data cleaning removes all unwanted artifacts, inconsistencies, and errors from the dataset so that the data is then ready for further processing or analysis. This can take a lot of time as it sometimes requires manual processing and simple browsing of the data. However, the benefit of such a well-prepared dataset for statistical analysis or ML is significant, as the quality and reliability of the results usually depend heavily on the quality of the input data. In computer science, this was condensed in the notion “garbage in, garbage out” (GIGO), or in other words, one cannot expect meaningful results if the input data is of poor quality. What exactly “bad quality” means depends on the particular case and/or analysis method and has not been globally defined. These are the key steps in data cleaning, some of which can be performed before or together with the *exploratory data analysis* (EDA) (cf. Chap. 9):

- *Repair structural problems:* This refers to problems due to different spelling (e.g., variables might be written in British and American English), typos, or wrong formatting (e.g., using a comma instead of a decimal point or using a wrong date/time format). The first column in Fig. 11.3a shows such an inconsistency in the time format used. Structural problems can be caused by the person, who is working with/creating the data, or be caused by errors during transferring or recording of the data. Some of these problems already might have been resolved, when file formats and metadata were validated.
- *Remove duplicate data:* Duplicate data could be due to data accidentally obtained during data collection, e.g., when during an experiment several readings were noted and one/some of them were noted twice. Another such situation can occur when during recording of a video due to a wrong sampling rate a slightly varying number of duplicates (repeated frames) are created. Lines 5 and 6 in Fig. 11.3a are an example for duplicates (and contain additional structural problems as well).

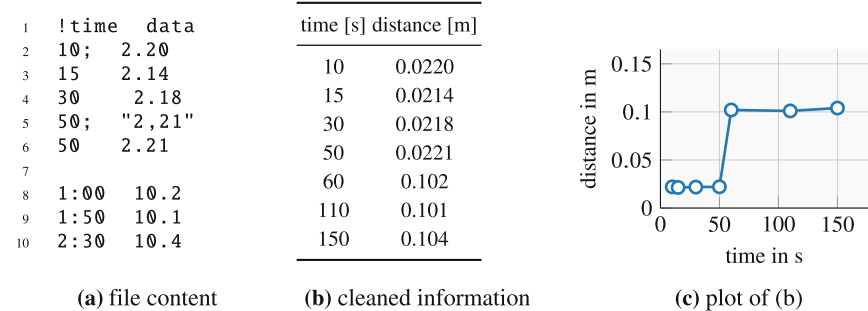


Fig. 11.3 Data preparation exemplified for a file containing column data. **(a)** raw file content (line numbers are not part of the file content); **(b)** cleaned up and “interpreted” data; **(c)** visualization of the data points. The data in **(a)** has a number of problems starting with the unclear column names, various delimiters between columns (semicolon and space) duplicate data, and inconsistent number formats. The processed data in **(b)** additionally contains knowledge of the units of the measurement. Visualizing the data during [EDA](#) as in **(c)** helps to ensure that the data is in a consistent state

- *Remove irrelevant observation:* This step refers to data or measurements that have been included in a dataset but are not needed. For performance reasons (file reading too slow) or data handling reasons (dataset too large) during reading, such data should be excluded. Or they need to be removed so that they are not automatically included in the data analysis. This might concern data that, for example, had been recorded *before* the actual experiment started. For practical purposes, removing irrelevant observations can make a lot of sense, e.g., when we want to combine data from different sources. By including only the relevant observations, it becomes easier and less error prone to merge such datasets into one file or array.

11.5.2 Outlier Detection

Outliers are data points that strongly deviate from what is expected from the dataset. Outliers can be related to a defective measurement device, any other disturbance during recording the dataset or writing the file, or due to human errors. However, they also might be the signature of *rare events*—which might be worth investigating. Generally, outliers can be sorted into three classes: (i) *Point outlier* or *global outlier* are those that are far away from all other data in the dataset. (ii) *Contextual* or *conditional outlier* are found within the dataset, and their values significantly deviate from all other points *in the same context*. For example, they may appear as noise. (iii) Collective outliers are encountered, when an entire subset of the dataset collectively behaves differently from the rest. Figure 11.4 shows a visualization of the three types of outliers using the dataset MDS-1 (tensile test).

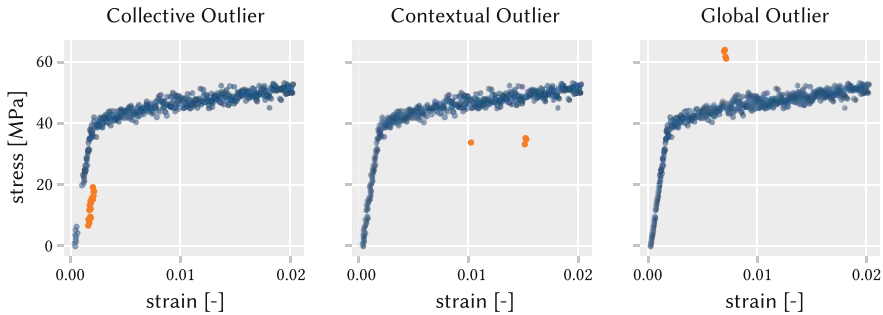


Fig. 11.4 The three main types of outliers (shown in orange) for stress-strain data with uncertainties. Refer to dataset MDS-1 (tensile test) for further details of the data

Outlier detection can be a very difficult task. Statistical methods can help to find outliers, e.g., by assuming that the data follows a certain kind of distribution; everything outside that distribution then is an outlier. Even ML methods are used (e.g., cluster analysis methods, which will be introduced later). During the data preparation stage, *unwanted* outliers should be removed carefully as it is not always clear, if they are indeed outliers or part of the dataset. As for data cleaning, also the removal of unwanted outliers benefits from and can be done during EDA.

11.5.3 Data Transformation

Data transformation is a topic one could easily dedicate several chapters to. It can be approached purely from the point of view of the data, not caring about the underlying problem. While this is perfectly fine, data transformation also offers the possibility to take some of our domain knowledge into account: for example, knowing that the data in a certain dataset is supposed to be normally distributed, because this is what the laws of physics postulate, might serve as a suggestion for how to scale the data properly.

The main motivation for data transformation is that many analysis algorithms or methods for data comparison work better when the respective datasets are directly comparable, when all values fall into a certain range, or when the investigations are performed in a certain (possibly different) coordinate system (e.g., a double logarithmic plot easily reveals if data follows a power law). The most commonly used methods for data transformation are introduced below.

Shifting

Shifting the data has the effect that all data are moved collectively to a new location. For example, for the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1 \dots N}$, which consists of N points in two-dimensional space, a shift by Δx along the x -axis can be mathematically written as

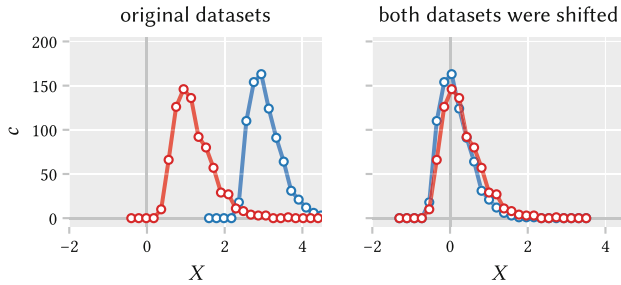


Fig. 11.5 Data shifting moves the data “as a whole.” Aligning all peaks at $x = 0$ makes it easier to, e.g., inspect if the shapes of the curves or distributions are similar

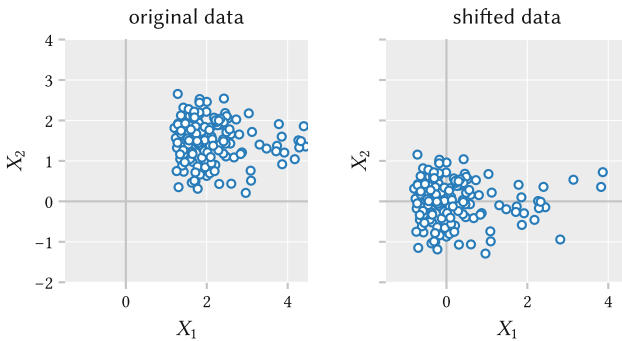


Fig. 11.6 Data shifting in two dimensions: this plot shows the positions of the data points, i.e., the pairs $\{(x_i, y_i)\}$ —or the distribution of two discrete random variables x and y . Thus, each point either has a value c_i attached or the points are be part of a “distribution”

$$\{(x_i, y_i)\}_{i=1\dots N} \mapsto \{(x_i + \Delta x, y_i)\}_{i=1\dots N}. \quad (11.5)$$

The shape of the distribution is preserved. Shifting is useful when, e.g., time-dependent measurements are to be compared, and each recording starts at a different point of time. Shifting each dataset, so that unique or clearly visible features of all datasets align, makes it possible to compare them more easily. Figure 11.5 shows an example of two datasets that were shifted, so that their maxima are located at the same x -coordinate. Similarly, bivariate or multivariate distributions can also be shifted, either along only one direction or simultaneously along multiple axes. An example of a bivariate distribution is shown in Fig. 11.6.

Data shifting typically refers to applying an offset to the input variables only. However, adding a constant value to the output variable might also be reasonable: if the dataset consists of temperature values $T((x, y))$ sampled at discrete positions $\{(x_i, y_i)\}$, shifting the data in space by $(\Delta x, \Delta y)$ as well as adding an offset to the temperature (e.g., converting from degrees Celsius to Kelvin) can both be reasonable.

Words of advice

For data that are part of a *probability* distribution function, shifting the probability value is not reasonable, while shifting the values of the random variables might be. Then, the expected value is shifted accordingly, but, e.g., the standard deviation stays the same.

Scaling

Whenever the data of two variables have very different magnitudes, a number of numerical problems may arise (e.g., numerical rounding errors when very large and very small numbers are added). Additionally, as with data shifting, also scaling is a useful preparation step for comparing functions or distributions. In Sect. 11.7, we introduce measures for quantitative comparison of the differences of datasets. Some of these measures such as the EUCLIDEAN distance are sensitive w.r.t. the absolute magnitude of the data. Figure 11.7 shows an example where the data was scaled along the first coordinate.

Non-dimensionalization is a type of scaling based on physical quantities and removes the physical units, which can help to simplify equations and to reveal their structure, e.g., in terms of the speed of light or in terms of internal “length scales” (see, e.g., [8]).

Scaling typically implies a linear mapping of the data or values to a “more suitable” range, i.e.,

$$X \mapsto \lambda X, \quad (11.6)$$

where λ is a constant. This constant can be chosen, e.g., as the dispersion of a distribution, and thereby also acts as a means to remove the physical units. Linear scaling of a random variable X to $\tilde{X}$ has the following implications for the mean μ and the standard deviation σ :

$$\tilde{X} = \lambda X, \quad \mu_{\tilde{X}} = \lambda \mu_X, \quad \sigma_{\tilde{X}} = \lambda \sigma_X. \quad (11.7)$$

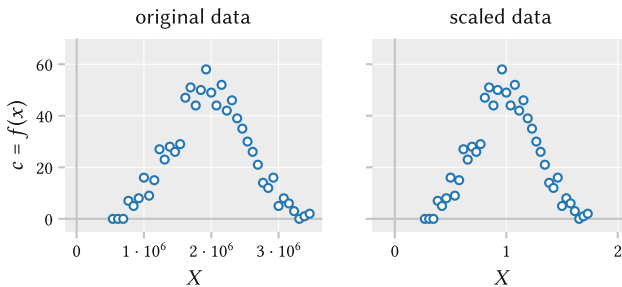


Fig. 11.7 Scaling of the x -coordinates of the data points. Here, the data in the right panel are transformed, so that the maximum is approximately at 1, while in the left panel the maximum is at $2 \cdot 10^6$

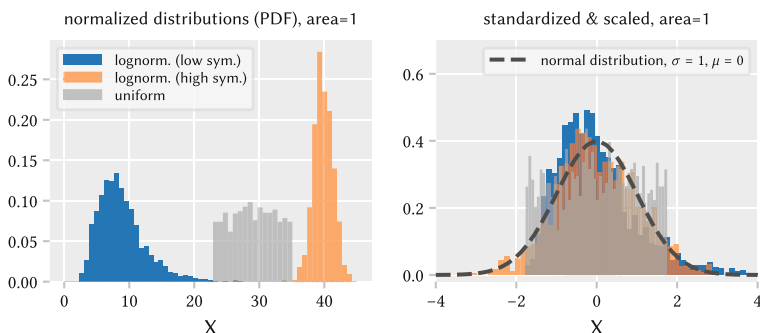


Fig. 11.8 Standardization of data distributions: left are three different distributions that are standardized and scaled, so that each of them has a total area of 1, shown in the right panel

Nonlinear scaling is often used to transform a nonlinear problem into a linear one. In the context of [ML](#), we will use such a scaling in [Sect. 13.1.3](#).

Standardization

Quite naturally, also combinations of scaling and shifting can be used. *Standardization* is such a combination. It changes the data, so that the distribution's standard deviation is 1 and the expectation value is 0. Mathematically, standardization is given by the following transformation:

$$X \mapsto \frac{X - \mu_X}{\sigma_X}. \quad (11.8)$$

A benefit of this transformation is that the shapes of different datasets become directly comparable because they are now defined on the same “scale,” given by mean and standard deviation. Data that originally was not normally distributed is still not normally distributed after the transformation. This is demonstrated in [Fig. 11.8](#).

11.5.4 Other Transformations

A number of other transformation types exist, some of which are not well defined. Among those is the commonly used notion of *normalization* which can have different meanings:

- either or both of linear scaling and shifting (e.g., so that all values are $\in [0, 1]$)
- plain standardization
- “converting” data such that it is approximately normally distributed (and not just normalized!), e.g., using the so-called Box-Cox power transform; see below
- scaling so that the area below a distribution is 1

Words of advice

Be careful and try to understand what is really meant when someone says “normalization” in the context of data transformation as this notion is not unambiguously defined.

Further transformations and commonly used notions are:

- *centering* or *mean centering*: The goal of data *centering* is to ensure that the dataset has a mean value of zero after scaling. This can be achieved by subtracting the mean of the data

$$X \mapsto X - \mu_X, \quad (11.9)$$

which can be generalized to multiple dimensions in the obvious way. We encountered this scaling in one of the previous chapters in the context of the covariance matrix.

- *range scaling* denotes a combination of scaling and shifting.
- *min/max scaling*: A combination of scaling and shifting is also denoted as *min/max scaler*. It is defined as the following map:

$$X \mapsto \frac{X - \min(X)}{\max(X) - \min(X)}, \quad (11.10)$$

which is, e.g., sometimes used during preparation of image training data for deep learning methods.

- *quantile transformer* and (*Box-Cox*) *power transforms*: These two transformations transform data into distributions that are similar to Gauss function distributions. Often, this requires changing the shape of the distribution.

A different class of data transformation methods are those that go beyond simple mathematical operations, for example, the methods of *principal component analysis* (PCA), which will be introduced in detail in Sect. 15.2. This is a method that maximizes the variance of data along a direction and thereby is able to reduce the number of variables by simultaneously optimizing the information loss. It involves rotation and translation of the data. Another ML specific transformation is the *one-hot encoding* where class labels are turned into lists of zeros with only a single one. For example, assume that a dataset consists of triangles, circles, and squares. Then, we could assign the following values: circle = [0, 1, 0], square = [0, 0, 1], and triangle = [1, 0, 0]. This is often for classification tasks in neural networks; see Chap. 17.

11.6 A Taxonomy of Machine Learning Models

The third step in the *Machine Learning Workflow*, Fig. 11.2, is concerned with the choice of the **ML** model used for the data analysis. In this part, we restrict ourselves to “classical” **ML** models, which are those that were derived in the spirit of statistics and mathematics. This separates them mainly from *deep learning* (**DL**) (and related) models, which are rather inspired by the information processing and interaction of biological neurons. The latter models are investigated separately in Part IV of this book.

Choosing the most suitable **ML** model for a particular data analysis or **ML** experiment can pose a challenge due to the large number of different available models. For instance, the Python library *scikit-learn* already contained at the time of writing this book more than 100 different **ML** models [4]. Therefore, in the following we will introduce the fundamental types of **ML** ranging from regression and classification through dimensionality reduction and clustering. An overview is given in Fig. 11.9. In the remainder of the book, most of the **ML** methods shown will be introduced.

Taking a look at Fig. 11.9, we see that the first categorization is whether learning takes place in a supervised, unsupervised, or reinforcement learning manner.

Supervised learning denotes a training process that relies on both the data of the feature variable, x_i , as well as on *labels* or *target variables*, y_i .

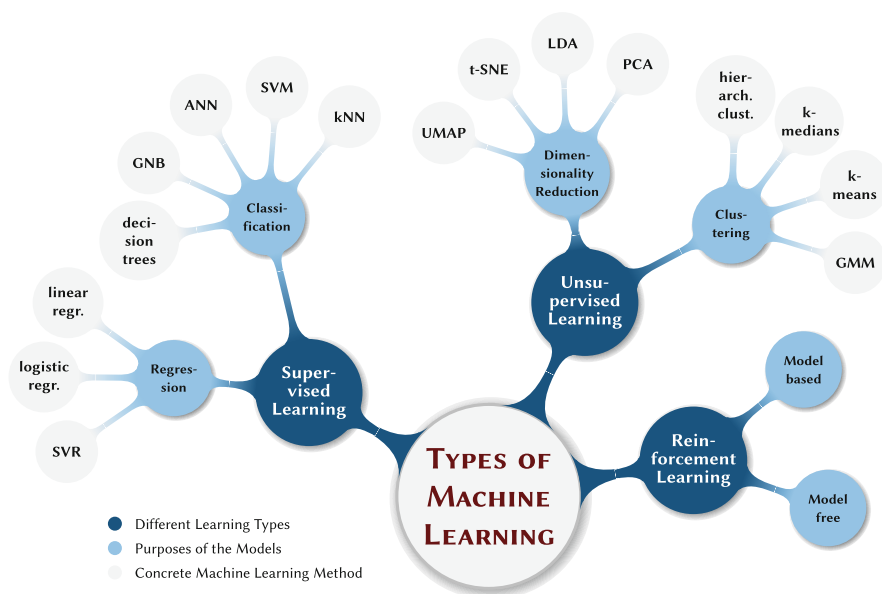


Fig. 11.9 Taxonomy of some of the most important machine learning model types

The points $\mathcal{P} = (x_i, y_i)$ are the *training examples*, presented to the ML model based on which the model is supposed to learn. An important algorithmic aspect is a feedback loop in the form of an error measure that measures the performance of the model. Essentially, the ML model “tries” (using an error minimization strategy) a number of different model parameters, makes predictions y_i^{pred} for the given feature values x_i , and compares them with the given labels y_i . The labels serve as the supervision, as they tell the algorithm, if a particular prediction was accurate or not. *Regression* (Sect. 11.6.1) and *classification* (Sect. 11.6.2) are the most famous problem types considered in supervised learning; we have already briefly discussed them above.

Unsupervised learning is a typical data mining and data exploration strategy.

It does not require supervision, i.e., no labels y_i are given, only the data of the feature variables are used as input. How or what could a model learn only from the input variables? The goal of such methods often is to detect patterns in a dataset and to understand the structure within potentially high-dimensional data. This can be achieved by, e.g., *clustering* methods, which group similar points together. Methods for *dimensionality reduction* are another type of unsupervised learning methods. These methods aim at reducing the possibly thousands of dimensions of a dataset to a significantly smaller number of features. A brief overview is given below in Sect. 11.6.4

A third class of learning methods is *reinforcement learning*. Such models are based on “agents” that interact with an environment during trial and error. Their actions are “reinforced” through the concept of “rewards” and “punishments.” The agent’s goal is to maximize the number of rewards. This model strategy is important for robotics, but also for playing strategy games (e.g., winning the game “Go” was a great accomplishment [7]). Generally, reinforcement learning can be divided into model-based learning and model-free learning, while the former is particularly efficient if a physical model can be used. At the same time, this also might constrain the model. To date, there are only a few applications of reinforcement learning in physics and materials science, which is why they will not be discussed in more detail here. An introduction is, e.g., given by Russell and Norvig [5].

11.6.1 Overview of Regression

In ML the goals of *regression* are twofold and exhibit a (at least at first glance) small but important distinction to what is done in statistics:

- The first goal is to *predict numbers* for a given numerical input, based on existing data, e.g., from measurements or computations (the so-called training data). This is a data-driven approach to approximate a (typically unknown) mathematical function and use it to make predictions.
- The second goal is *inference*, which typically implies that we want to find out something about the underlying relation between input and output data, e.g., in form of finding the most suitable mathematical formulation.

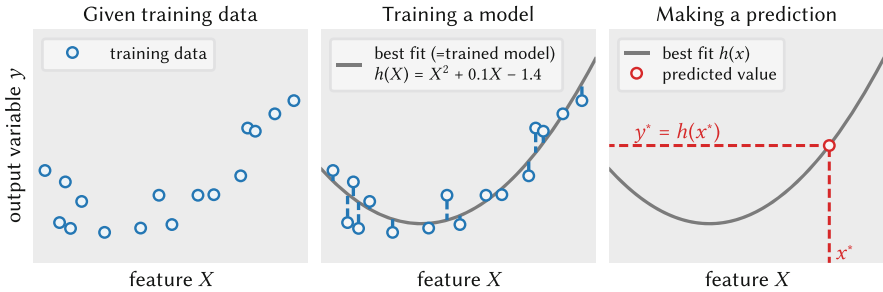


Fig. 11.10 Regression requires training data (left) and a function h . The coefficients of h minimize the deviations between the data and $h(x)$ (middle panel) allowing to make predictions with the trained model h : for an arbitrary, new input value x^* an output y^* is predicted (right)

Figure 11.10 illustrates the concept of ML regression. Given is training data, pairs of features X and output values y . Next, we decide that the *model* h is a quadratic function. Why? Because we, for example, already might know from the underlying physics that the data should roughly obey such a relation, or because often, in the spirit of Occam’s razor (Sect. 1.2.5), a simple model is the most useful one, to start with. But at the end of the day, this is simply *our* decision.

The *model parameters* or *weights* of the ML model are the coefficients of h . They will be determined through the training process. The trained model (the *hypothesis*) consists of the functional form (the quadratic function) and the three parameters. This can now be used for making predictions, e.g., of the output value at position x^* , which is obtained by evaluating the function $y^* = h(x^*)$. The trained model can also be used for *inference*, e.g., analyze where the function’s slope is zero or interpret the function parameters in terms of the underlying physics problem.

But is this all not just plain curve fitting? Where *exactly* is the “machine learning?” In ML, curve fitting is an important technique, but the ML aspects are contained in the focus on making predictions or inference and *not* just on curve fitting. Also, ML regression comes with a large toolbox of methods and concepts, e.g., for figuring out if the training dataset is sufficiently large or for ensuring that our models have indeed notable predictive capability.

11.6.2 Overview of Classification

In ML classification, the goal is to predict different categories of data. The input, i.e., the features, consists of numerical data, while the output denotes the class or label of a data point. Thus, classification is another supervised learning method. Once the ML model is trained, it is able to predict, for a given input X , what the corresponding class Y would be. Data of different classes are separated by the *decision boundary*; finding the decision boundaries is the purpose of training. Figure 11.11 shows an example of two features and a target variable that represents three different classes,

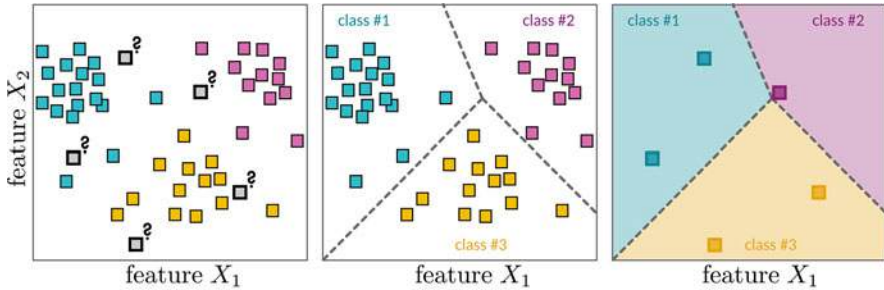


Fig. 11.11 Supervised classification assigns a label to each point. Here, the label is a color and reveals whether a point belongs to class #1, #2, or #3. For the colored points in the left panel, the label is already known (the training data). From those, the decision boundary (dashed lines in the middle panel) is obtained. The algorithm can now predict the class number for new data, e.g., the gray points with the question marks in the left panel. This results in the labels symbolized by the color of the markers in the right panel

indicated by the three different colors. Classification tasks are probably *the* type of ML problems that we are most familiar with: already since our early childhood, we sorted the world of animals into cats, dogs, bunnies, or mice. The features used in classification are usually interpretable quantities. For example, in the case of bending of a cantilever beam, one feature might be temperature; the other one might be the deflection of the beam; the label is then “beam has cracks” or “beam is intact.”

Although making *predictions* is probably the most important aspect for classification tasks, the aspect of *inference* is also of some relevance (e.g., “where do the decision boundaries meet?”).

11.6.3 Overview of Clustering

Clustering of data is an unsupervised ML method. The learner needs to accomplish the task of deciding, which of the given data points belong into the same group. Figure 11.12 sketches the result of such a cluster analysis. Looking for the nearest neighbors is one commonly used approach. However, it is not always straightforward to decide which metric to use for measuring distances in the feature space. A different approach is used in density models, which search for those regions in the feature space that would result in the densest connected regions.

Typically, these methods are not able to automatically identify the number of clusters, and we have to make an informed decision, e.g., based on our physical insight into a specific problem. Such a subdivision of a dataset into different clusters can also be used to provide labels for a subsequent supervised learning methods. Another application is the detection of outliers.

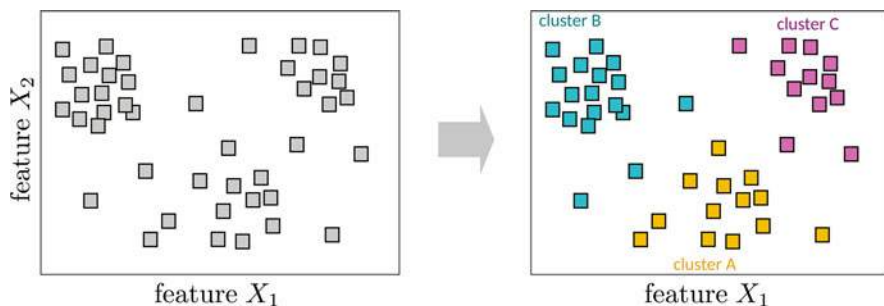


Fig. 11.12 A clustering ML model took the coordinates of the points as input and, based on that, groups points together (here into three different clusters) indicated by the three different colors. Unlike in classification in Fig. 11.11, the algorithm does not have any “examples” for the three different clusters

11.6.4 Overview of Dimensionality Reduction

Dimensionality reduction denotes a process, during which high-dimensional data (i.e., datasets that have a large number of features) is reduced to low- or at least lower-dimensional data. The main objective of these methods is to preserve as much of the original information as possible, or at least, to reduce the original dataset to the information of interest. For a simplistic example of a dimensionality reduction, let us imagine we are looking at the silhouette of an object projected by a torch on a wall. We can see enough to make some conclusions regarding some geometrical aspects that we are interested in. But we clearly also lose information regarding the three-dimensional structure of the object.

High-dimensional data can be problematic for a number of reasons. One reason is the *curse of dimensionality*, cf. Sect. 2.4, which states that for a comparable “data density” while increasing the dimensionality of the feature space, the amount of data has to be increased more than linearly. As this affects also, e.g., supervised learning methods, dimensionality reduction can often be found in the form of a preprocessing step of a dataset. Another application domain is *data visualization*: visualizing more than two or three dimensions is practically impossible. Therefore, reducing a feature space to two or three dimensions, as some of the algorithms do, is a very useful approach to allow visualization of complex datasets. Last but not least, some datasets are simply too big and storage becomes an issue, while dimensionality reduction might be able to help.

It is important to understand how these ML algorithms work and in which way they alter the data, because after a dimensionality reduction, some information is always removed; and we have to ensure that this is not the relevant information.

Table 11.1 Names for input and output variables used in ML. The boldface names are the ones commonly used in this text

X or $X_1, X_2, \dots$	Y or $Y_1, Y_2, \dots$
input variable	output variable
feature variable	target variable
feature	label
independent variable	dependent variable
explanatory variable	response variable
predictor variable	predicted variable
regressor or covariate	regressand

11.6.5 The Zoo of Names for Input and Output Variables

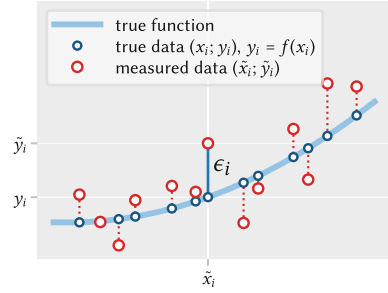
We conclude this section by saying a few words about the representation of data for machine learning problems. The main formal aspects have already been introduced in Sect. 3.3, while in this chapter we have so far focused on the huge variety of different methods. Some of these methods have their roots in statistics; others have grown from completely different disciplines. Consequently, this also brings along a large variety of different names for variables. Most of the time, we will use *feature* in this text, when referring to an input variable, while in some instances we call the output variables also *target* variable. In particular in the context of classification, we use *label* when referring to the output variable; in the literature *label* is also used for the output of regression problems.

To be able to easily read the ML and data science literature, we summarize the names that are most commonly found in Table 11.1. In this text, we avoid the notion of *independent* variables, because this might suggest that these variables are statistically independent of each other. Most of the time, input variables are in fact *not* statistically independent in ML. A notion that seems to be missing is that of the *attribute*. Usually, the attribute denotes the input variable; however, sometimes there are input and output attributes, which is why we avoid it altogether. Typically, the names for the input and output variables come as pairs, as highlighted in Table 11.1. It is indeed a zoo of names ...

11.7 Error Measures for Numerical Data

Quantifying the agreement between individual observations and datasets is key to quantifying the performance of a ML model. There are two fundamentally different classes of measures: The first one considers numerical data and compares the values of pairs of observations and then “somehow” sums these contributions up. This is the subject of the remainder of this section. The second type measures the similarity between sets of items, which is important for classification problems and will be introduced in Sect. 11.8. Both measures have in common that ultimately the error (or agreement) is quantified by a single number.

Fig. 11.13 Measuring data is affected by errors. The measurements are an estimate for the “true” data, and the deviation between true and measured data is the error ϵ_i



11.7.1 Introduction to the Problem

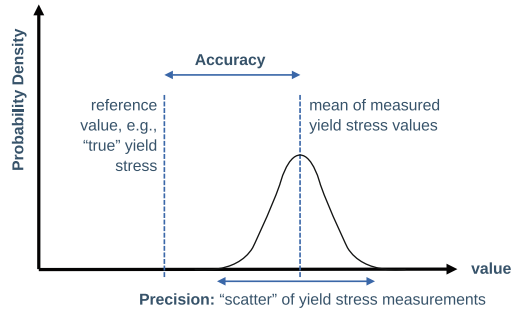
In the following, we assume that a “true” function exists, e.g., a physical law from which we are able to obtain “true data” or the data (x_i, y_i) for $i = [1, n]$ that are located on the line in Fig. 11.13. This is our reference dataset. In the context of statistics, the set of all possible points located on the line corresponds to the population, and the points on the line represent the expected values. However, if the purpose is not to obtain an estimate for the population, then the reference data does not necessarily have to be obtained from a true function, and it can be *any* dataset. A second dataset are measured data $(\tilde{x}_i, \tilde{y}_i)$, which are affected by deviations or disturbances ϵ_i from the reference data:

$$\epsilon_i = y_i - \tilde{y}_i . \quad (11.11)$$

The deviations are called *residuals* or *errors*, depending on the context, as we will see in the next chapter. For now, we will call ϵ_i an error.

In science and engineering, there are two types of errors; see Fig. 11.14: *precision* is the degree by which repeated measurement give the same result. If the distribution of measured values is broader, then the scatter in the data will be larger; thus, repeated measurements will be further away from each other. *Accuracy*, on the other hand, tells us how close some measurements are, on average, w.r.t. the true or reference value. Systematic errors often impact accuracy, e.g., if for all temperature measurements we read the thermometer from an acute angle, so that our reading is always 0.2°C too high, the accuracy is this constant offset of 0.2°C . Any measurement can be accurate, but is not automatically precise and vice versa. The goal is to have measurements (or in the context of **ML**: predictions) that are both accurate as well as precise. In the context of **ML**, we will encounter this two types of errors again. Then, they are called *bias* instead of accuracy and *variability* or variance instead of precision.

Fig. 11.14 Visualization of the two types of errors, i.e., accuracy and precision, based on the example of yield stress measurements



11.7.2 Mean Absolute Error

The most straightforward error type is the *mean absolute error* (MAE) which in the context of ML and numerical optimization is also called *L1 loss*. It is the mean of all n absolute errors:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \tilde{y}_i| = \frac{1}{n} \sum_{i=1}^n |\epsilon_i|. \quad (11.12)$$

The MAE is a simple error measure that ranges from 0 to $+\infty$ and averages the vertical distance between points.

The error is in units of the investigated variable and a scale-dependent measure, which is an advantage compared to other measures introduced below. Therefore, one has an intuitive understanding of the magnitude of a particular MAE value (if it is “large” or “small”).

This type of measure is used for time-series analysis. However, in general MAE is inferior to the *mean squared error* (MSE), which is introduced next. One of the reasons is that the MAE is not differentiable at 0. Furthermore, each measurement contributes without weighting and proportionally to the respective deviation ϵ_i . If the norm is removed, we end up with the *mean error*.

In Python, the MAE can be easily computed for the one-dimensional numpy arrays `y_ref` and `y` by `MEA = np.mean(np.abs(y_ref - y))`, where `np` as always is an alias for the numpy package.

11.7.3 Mean Squared Error

The MSE is sometimes also called *mean squared deviation* (MSD) or, in the context of ML and numerical optimization, also *L2 loss*. It is one of the most commonly used error measures and defined as the average of the *squared* deviations:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2 = \frac{1}{n} \sum_{i=1}^n \epsilon_i^2. \quad (11.13)$$

Due to the square, the **MSE** is always larger or equal to zero, and it is symmetric with regard to the sign of the deviation ϵ_i . A lower value implies a smaller deviation. Its values are in units of the squared variable of interest, which can make the intuitive interpretation more difficult, as compared to the **MAE**.

Due to the square, the **MSE** is a second moment of the error about zero, and as such, it quantifies both the precision (or variance) as well as the accuracy (or bias). As another convenient property, outliers have a disproportionately large influence on the value of the **MSE**. Or, in other words, if large errors are particularly undesirable in an algorithm that involves the **MSE**, then the square effectively penalizes such values. Finally, it will turn out that the square is also very useful in the context of error minimization because when a first derivative or a first gradient needs to be calculated in order to find the minimum, the resulting equations using the **MSE** are much simpler than any other error measure. The Python implementation is not too different from the implementation of the **MAE**: `MSE = np.mean((y_ref - y) ** 2)`.

A very useful property is the following decomposition

$$\text{MSE} = \text{Var}(\hat{X}) + \text{Bias}(X, \hat{X}), \quad (11.14)$$

where $\hat{X}$ is the measured variable and X is the true or reference variable. As a consequence of this relation, it follows that if the bias is zero, the **MSE** and variance are identical.

11.7.4 Root Mean Square Error

The standard deviation behaves to the variance as the *root mean squared error* (**RMSE**) does to the **MSE**:

$$\begin{aligned} \text{RMSE} &= \sqrt{\text{MSE}} \\ &= \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n \epsilon_i^2}. \end{aligned} \quad (11.15)$$

Similar to the **MSE**, also the **RMSE** is always greater than or equal to zero, and a lower value implies a smaller deviation. As for the **MSE**, outliers have a disproportionate influence. The main difference to the **MSE** is that the **RMSE** is measured in units of the variable of interest. Nonetheless, comparing two such error values with one another is difficult due to the nonlinearity that stems from the squared term.

The **MAE** can be used as an upper and lower bound for the **RMSE**, and it can be shown that:

- $\text{MAE} \leq \text{RMSE}$ holds
- if all errors have the same magnitude then it is even $\text{MAE} = \text{RMSE}$
- $\text{RMSE} \leq \text{MAE} \times \sqrt{n}$, where n is the number of records.

11.7.5 Coefficient of Variation

To be able to compare RMSE values, it is possible to perform a *normalization*, e.g., either with the minimum/maximum range of the variable, with the interquartile range, or with the mean value of the variable. In the latter case, the resulting measure is called *coefficient of variation (CV)*:

$$\text{CV} = \frac{\text{RMSE}}{\bar{X}}. \quad (11.16)$$

Words of advice



Note that the standard deviation divided by the mean of a variable X , σ_X/μ_X is also called coefficient of variation. Furthermore, *cross validation*, to be introduced in Chap. 16, is also often abbreviated as CV. To avoid ambiguities, one can also write the coefficient in Eq. (11.16) more explicitly as CV_{RMSE} or as $\text{CV}(\text{RMSE})$.

11.7.6 Root Mean Squared Log Error

As a last and somewhat more exotic error measure, we briefly introduce the *root mean squared logarithmic error (MSLE)* which is defined as follows:

$$\text{MSLE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log(y_i + 1) - \log(\hat{y}_i + 1))^2}. \quad (11.17)$$

While the RMSE has the character of an absolute error and scales like the variable, the MSLE has the character of a relative error. The logarithmic scaling of the values strongly reduces the influence of extreme values. However, variables need to be scaled, so that they lie in the definition range of the log function. As an example consider the following two cases in which two values, y and $\hat{y}$, are given that differ by 10%:

- $y = 100, \hat{y} = 90 \rightarrow \text{MSLE} = 0.0153, \text{RMSE} = 10$
- $y = 10,000, \hat{y} = 9000 \rightarrow \text{MSLE} = 0.0153, \text{RMSE} = 1000$

While the **RMSE** is scaled according to the scale of the variable, the **MSLE** as a relative measure remains unchanged because it is insensitive w.r.t. the scale of the error. The **MSLE** is not symmetric, which shows in the different error values for under- and overestimation: assume that $y = 1000$. Then for a lower value of $\hat{y} = 600$ it is **MSLE** = 0.510 and **RMSE** = 400 while for $\hat{y} = 1400$ it is **MSLE** = 0.33 and **RMSE** = 400. Therefore, there is a larger penalty if y is underestimated than if it were overestimated. In many applications this is reasonable, as there might be a given limit value that shall not be exceeded, e.g., a radiation dose, while it is entirely harmless to stay below the limit value.

11.8 Similarity Measures for Classification Problems

In the following, we first introduce several measures that are used for pixel-wise comparison in image data before we then introduce the “cross-entropy loss function” that is used as an “error measure” in classification problems. A materials science example, based on which all measures are compared, is given in Sect. 11.8.6.

11.8.1 From True Positives to False Negatives

Binary classifications problems require a decision whether a data point belongs to one class or to the other one. For example, deciding if a specimen is defective or not is a binary classification problem; “not defective” could be represented by zero and “defective” by one. This type of problem is also encountered in *binary segmentation* tasks, in which the **ML** model has to decide if a pixel belongs to a specific object or not. In such cases the following four measures are important:

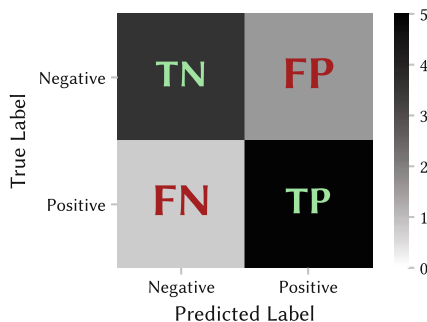
- The *true positives* (TP) is the number of all data points that have positive labels (i.e., they have the value 1) for both the predicted target variable and the true target variable.
- A *true negative* (TN) is just the opposite: if both the true value and the predicted value are 0, this adds to the number of true negatives.
- The *false positives* (FP) is a prediction of 1, while the true value is supposed to be 0.
- A *false negative* (FN), on the other hand, is a data point that is predicted as 0, while the true value is 1.

As an example, we take a look at the following two arrays:

```
In [1]: import numpy as np
Y_pred = np.array([0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0], dtype=int)
Y_true = np.array([0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0], dtype=int)
```

Fig. 11.15 Confusion matrix for a binary classification.

The gray shades indicate the number of TN, TP, etc. If all data are correctly predicted, there should be only values along the diagonal from top left to bottom right



As we are going to compare the arrays element-by-element, there is one important aspect to consider: comparison between numbers using the equal sign, such as `Y_pred == 1`, should be done only for integer numbers: hence the `dtype=int` above.

```
In [2]: TP = np.sum((Y_pred == 1) & (Y_true == 1))
        TN = np.sum((Y_pred == 0) & (Y_true == 0))
        FP = np.sum((Y_pred == 1) & (Y_true == 0))
        FN = np.sum((Y_pred == 0) & (Y_true == 1))

        print(f'TP: {TP}, FP: {FP}, TN: {TN}, FN: {FN}')
```

```
Out [2]: TP: 4, FP: 1, TN: 5, FN: 2
```

The comparison `Y_pred == 1` results in an array that contains `True` and `False`. The operator `&` performs an element-wise logical AND which again results in an array of `True` and `False`. Summation of the elements of such arrays implicitly converts `False` to 0 and `True` to 1, effectively counting the number of `True` values. All four quantities can be computed in this way.

For floating point numbers, this is not a well-defined operation. Figure 11.15 visualizes the four numbers in form of a matrix which is commonly called *confusion matrix*. A confusion matrix can also be used to visualize the performance for multi-class classification problems; the different class labels define the horizontal and vertical axes.

11.8.2 The Jaccard Similarity or Intersection Over Union

Jaccard Similarity

Assume that for two sets of data, A and B , we want to quantify how *similar* these sets are. The JACCARD *similarity* (or the JACCARD *similarity coefficient*) between A and B is given as the number of elements that are contained both in A and B relative to the total number of different elements (i.e., the elements that are either in A or in B). Without duplicates, these are sets, cf. Sect. 3.2.1). The JACCARD similarity J is given by

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}, \quad (11.18)$$

where the number of elements of a set A is given by $|A|$. As a direct consequence, two identical sets have $J(A, B) = 1$ due to the equality $A \cap B = A \cup B$. The other extreme is that two sets do not have any element in common. Then it is $J(A, B) = 0$. For all other cases it is $0 < J(A, B) < 1$.

The Jaccard similarity works well as long as the two sets have approximately the same size, $|A| \approx |B|$. If one of them has significantly more elements than the other, the denominator of Eq. (11.18) is always larger than the intersection of the two sets and J will be small.

Intersection Over Union

In the context of [ML](#), the JACCARD similarity coefficient is mostly encountered in the context of computer vision problems, in particular for *object detection* and *image segmentation* tasks. It is then encountered as *intersection over union* ([IoU](#)).

Object Detection and Image Segmentation



Object detection typically seeks bounding boxes, rectangles that fully enclose an object, while image segmentation separates the pixels of an image into, e.g., “object” (black) and “background” (white). More details are given in [Part IV](#).

For measuring how well a [ML](#) algorithm predicts a particular object in an image, we compare the number of pixels belonging to the predicted object, A , to those of the “true” object, B which are assumed to be known (the “ground-truth” given by the labels). Mathematically, this is nearly identical to Eq. (11.18), but for [IoU](#) we usually do not refer to sets. Instead, it is often directly phrased in terms of number of pixels (even though this is not strictly necessary).

Definition 11.2 (Intersection over Union). The coefficient of intersection over union ([IoU](#)) is given for two objects of an image, A and B by

$$\text{IoU}(A, B) = \frac{\text{number of pixels belonging to objects } A \text{ and } B}{\text{number of pixels of the union of both objects}}, \quad (11.19)$$

Visually, it can be interpreted as shown in [Fig. 11.16](#), where, for didactical reasons, we show also objects of different sizes. As only the relative number of

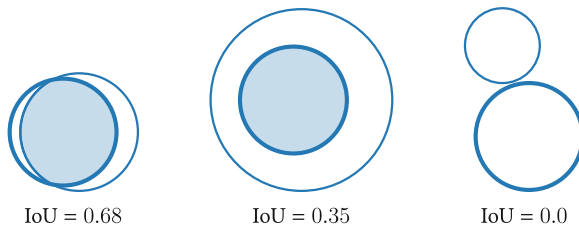


Fig. 11.16 Visualization of different cases of the **IoU**. The intersection are those items that the two sets have in common (indicated by the blue filled area); the union is the contour of all items. Dividing the former number by the latter number gives the **IoU** value, which is represented as the fraction of the blue shaded area w.r.t. the total area of both circles. In an image, the areas would be the sets of pixels

pixels or the areas are considered, the measure is scale-invariant, i.e., scaling both directions of an image with the same factor does not change the IoU value.

Often, it is convenient to use an alternative equation for computing the IoU, which is based on counting true positives (TP), false positives (FP), and false negatives (FN); refer to Sect. 11.8.1 for a general introduction to these quantities. In the context of images, a true positive is, e.g., a pixel that is correctly predicted to belong to an object; a false positive is a pixel that is predicted to belong to an object, but in fact is part of the background or of another object; a false negative is a pixel that was predicted not to belong to the object of interest, but in fact *does* belong to that object. The IoU coefficient can then be expressed as

$$\text{IoU} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}}. \quad (11.20)$$

As the number of true negatives does not show up in this equation, the formulation is not symmetric. This implies that unlike in Definition 11.2, interchanging the true object and the detected object would yield different results.

11.8.3 The Dice Coefficient

The *Dice coefficient* is another measure for comparing geometrical areas of, e.g., images. It is defined as

Definition 11.3 (Dice Coefficient). Given are the areas of two objects A and B . For these, the Dice coefficient is given by

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|} \quad (11.21)$$

The Dice coefficient is also called *F1 score*. In the definition, the denominator gives the sum of the total area of A and B , which is different from the intersection of the areas of the two objects (the latter is smaller or equal than $A + B$). As for the JACCARD metric, the range of possible values is $[0, 1]$ with 1 denoting a perfect match. The Dice coefficient can be written as well in terms of TP, FP, and FN:

$$\text{Dice} = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}. \quad (11.22)$$

As for IoU, the number of true negatives does not occur, and the equation is not symmetric. This implies that unlike in one of the above definitions of intersection over union, the true object and the detected one cannot be interchanged without changing the result.

11.8.4 Precision and Recall

As two last measures for classification, we introduce the precision and recall. The *precision* is the number of TP divided by the total number of elements that belong to the positive class:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad (11.23)$$

Precision indicates how *precise* the ML model was. This is a useful measure if for a particular problem, false positives are more important than false negatives, or if we cannot calculate FN or FP.

Recall (also called *sensitivity*), on the other hand, is the number of TP divided by the total number of elements that belong to the positive class:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (11.24)$$

This number tells us how many of the true positives the model was in fact able to predict as TPs.

In both cases, there is an “asymmetry” between FP and FN. This could be of relevance, e.g., when wrongly predicting that a sample is fractured has different consequences than wrongly predicting that a fractured sample is not defective.

11.8.5 Categorical Cross-Entropy

We briefly introduce an entirely different concept and formulation that is commonly used for classification and that also can be used for multi-class tasks. In typical classification tasks, one has to compare distributions of pixels, which, e.g., might be the label for a specific object. To do this based on the *distributions* of values,

the *cross-entropy* can be used. Assume that the y^{pred} are the predicted class labels, while the y^{true} denote the ground truth. The categorical cross-entropy “loss” $\mathcal{L}$ (in the next chapter we will introduce the “loss” as a kind of error) is given for a multi-class problem of N classes as

$$\mathcal{L} = - \sum_i^N y^{\text{true}}_i \log(y^{\text{pred}}_i) \quad (11.25)$$

where i is the number of classes. In case of a binary classification, this reduces to

$$\mathcal{L} = -y^{\text{true}} \log(y^{\text{pred}}) + (1 - y^{\text{true}}) \log(1 - y^{\text{pred}}) . \quad (11.26)$$

Note that the y are considered as distributions (here distributions of pixel values).

11.8.6 Example: Binary Segmentation of a TEM Image

In this example, we investigate a [TEM](#) image of a dislocation. The goal is to create a “mask” that captures only the pixels that are representing the dislocation. The [TEM](#) image is shown in the left panel of Fig. 11.17. The dislocation consists of 222 pixels out of a total of 4096 pixels contained in the image. The “true mask” shows the reference mask that was created by a domain expert: by looking at the image, those pixel that belong to the dislocation were manually colored in black (this is what is meant we someone speaks about “hand-labeling”). Note that there is no unique solution, and two different persons most likely will create two different masks. Mask A, B, and C show three different predictions for the dislocation. They were artificially constructed, but, in general, would be the output of a [DL](#) model. Mask A and B show a dislocation line that is fragmented; mask B and C contain additionally a “v-shaped” artifact. For these three masks we obtain the metric scores shown in Table 11.2.

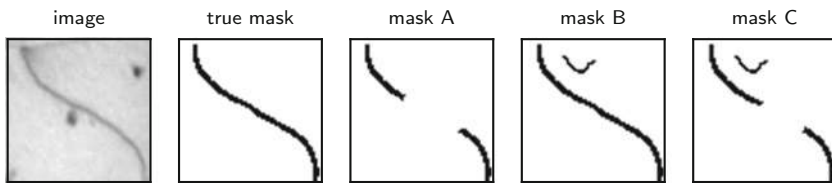


Fig. 11.17 [TEM](#) image of a dislocation (the dark curve) for which the “mask,” i.e., those pixel that belong to the dislocation, is wanted. Shown is the “true” mask, that serves as the reference, together with three different predictions

Table 11.2 Resulting metric scores for the predictions shown in Fig. 11.17

Prediction	IoU	Dice	Precision	Recall	TP	FP	TN	FN
Mask A	0.68	0.81	1.00	0.68	151	0	3874	71
Mask B	0.88	0.94	0.88	1.00	222	28	3846	0
Mask C	0.67	0.80	0.85	0.75	168	28	3846	54

In summary, the following can be observed:

- Mask A does not contain any FP, which implies that each pixel that was identified as a “dislocation pixel” is indeed part of the dislocation. In such a case, one can expect the precision to be 1.0 as found for this case. However, we can see that not all dislocation pixels were correctly identified, as a significant section of the dislocation is missing. This does not reflect correctly in the precision, but we can observe this error in the recall which is only 0.68. The corresponding ML model is said to have a higher bias or a tendency to under-predict. Therefore, despite perfect precision, the model’s overall performance is not ideal due to the lower recall.
- In contrast, the predicted mask B exhibits a different bias. It contains all dislocation pixels but additionally also a part of the background (showing as FP). We observe that now the recall is 1.0, which implies that the model has successfully identified all true dislocation pixel. However, this comes at the expense of precision, which is reduced to 0.88 due to the model predicting parts of the background as dislocation, which increases the number of false positives. By comparison, we find that the Dice score has improved significantly and is above 0.94 now. This can be interpreted such that the model has now quite a balanced performance in terms of precision and recall. However, the high Recall score indicates that the model tends to over-predict.
- The last prediction, mask C, contains both false negatives and false positives. It is therefore a combination of the two previous predictions.

The results show both under- as well as over-prediction, which results in the (approximate) balance between Recall and Precision. Despite this, both the IoU and Dice scores suggest comparable performance as compared to mask A, which is clearly not the case.

In particular the last case underlines the limitations of using only a single metric for model evaluation. It’s critical to take into account both types of errors (false positives as well as false negatives) to faithfully represent the model performance.

Things to Remember About Similarity Measures for (Binary) Classification★

In summary, we can see that even though a number of metrics are available, each provides different information. Very often, it is not feasible to rely on only one of them for judging the quality of a prediction. Using all, or at least some, of them together gives a reliable estimate of the quality of the predictions.

11.9 Exercises

11.1. Is it possible that the mean squared error is zero? What would that imply?

11.2. Given are the following true and predicted data:

y^{true}	3.2	4.4	6.7	11.0	14.6	18.7
y^{pred}	2.8	4.3	6.8	11.5	15.0	16.7

- Compute the *mean absolute error* (MAE), the *mean squared error* (MSE), the *root mean squared error* (RMSE), and the *root mean squared logarithmic error* (MSLE).
- If the fourth predicted value changes from 11.5 to 21.5, how does that affect the different error measures?

11.3. In a research project, a large dataset of SEM images is available. The SEM images show metal surfaces, and we are interested in analyzing and finding defects in them. It is time-consuming and labor-intensive to manually annotate and highlight defects in each image. Which class of machine learning techniques (supervised or unsupervised) would be most appropriate to analyze and identify different types of defects (and existence or lack thereof) in these SEM images?

11.4. In a research project, a large dataset of TEM images for different nanoparticles is available. The nanoparticles come in different sizes and are located in different regions in the image. A domain expert has annotated the image to mark the pixels which encompass each nanoparticle to separate them from the background. The goal is to use the dataset and the annotation to train a model that learns to automatically identify and locate nanoparticles in new datasets. Which class of machine learning techniques (supervised or unsupervised) would be most appropriate to analyze and identify nanoparticles in each TEM image?

11.5. In a dataset of ceramic materials, the recorded properties for each sample include processing parameters and material parameters such as temperature, pressure, time, and the corresponding hardness value. A researcher wants to find out the relationship between the processing conditions and the resulting hardness value. Is this problem better suited to be treated as a regression or a classification problem? Explain why.

11.6. In the research problem in Exercise 11.3, imagine that we have the resources to annotate each SEM image and specify the type of defect (i.e., cracks or pores) that is present. We then want to use machine learning to train a model that learns this and can be used to identify defect types in new datasets. What kind of problem is this treated as, classification or regression?

References

1. C. B. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2011. ISBN 978-0-387-31073-2. URL <https://www.springer.com/gp/book/9780387310732>.
2. T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997. ISBN 0070428077.
3. A. Müller. *Introduction to Machine Learning with Python*. O'Reilly, 2017. ISBN 978-1-449-36941-5.
4. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
5. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597.
6. A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959. DOI <https://doi.org/10.1147/rd.33.0210>.
7. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016. DOI <https://doi.org/10.1038/nature16961>.
8. M. Zaiser and S. Sandfeld. Scaling properties of dislocation simulations in the similitude regime. *Modelling and Simulation in Materials Science and Engineering*, 22(6):065012, Aug 2014. DOI <https://doi.org/10.1088/0965-0393/22/6/065012>.



A First Approach to Machine Learning with Linear Regression

12

If you don't know where you are going, any road will get you there.

Lewis Carroll (1832–1898)

English author and mathematician

12.1 The Roots of Regression Analysis

12.1.1 A Historical Perspective

Regression analysis is probably *the* flavor of *machine learning* (ML) analysis that is most closely and most obviously related to the field of statistics. Regression originates from statistics and is a “statistical toolbox” with the purpose of estimating the relationship between one or more independent variables—the inputs or features—and one or more dependent variables, the outputs. It is instructive to take a look at the history of regression analysis.

The origins of statistical regression date back to the late nineteenth century and Francis Galton (1822–1911), a truly multidisciplinary scholar who did research in the field of statistics, geography, biology, sociology, and meteorology, to name but a few. Among others he investigated the diameters of sweet peas Galton [6]. His goal was to understand which and how these characteristics were inherited from one generation to the other. He designed a statistical experiment, and for the analysis he introduced the linear regression technique where he approximated averaged data by a line and used this for drawing conclusions. A summary of the experiment and the data analysis is given in Example 12.1.

Another important step toward today's regression analysis was taken by KARL PEARSON (1857–1936) who also strongly influenced developments in physics, theology, and psychology. He was one of the pioneers of statistics and is known in particular for his *correlation coefficient*. In 1901 he introduced the

“regression line” that he obtained from a least square error estimate [5]. Ever since then, regression analysis became what we know it today (including the analysis of regression coefficients and the relation to the p -value) and turned into an important tool for statisticians.

It should be noted that apart from being excellent scientists and in particular pioneering statisticians, both GALTON and PEARSON also believed in “eugenics” and the concept of “race hygiene” in Germany under Hitler [4], which casts a slight shadow of the history of statistics.

Example 12.1 (The Origins of Linear Regression: Galton’s Pea Experiment)

The British anthropologist and meteorologist Francis Galton, a cousin of the famous Charles Darwin, studied a vast range of topics (see, e.g., [2]) but became particularly well-known for his work on inheritance. In 1875 he investigated how genetic variations from mother plants become manifest in daughter plants by an experiment: he gave 700 sweet pea seeds to each of seven friends who had to plant them, grow, and finally harvest a new generation of peas. Sweet peas are rather easy to grow and self-fertilize which results in quite a controlled experiment as there is no influence from *two* parents to be considered (Fig. 12.1).

To analyze his data Galton used a “scatter plot” of the sizes of the daughter peas versus those of the mother peas. The data is shown in Fig. 12.2 where markers of different sizes indicate how many times a particular combination of parent/daughter seed size was encountered (however, instead of markers Galton simply wrote the frequency as a number).

He then computed for each of the seven parent seed sizes the median value¹ of all corresponding daughter seed sizes and drew a straight line through these points—the first regression line was born! The slope is with ≈ 0.3 much below 1 from which follows that *the diameter of the daughters are closer to the median (or the mean) than the parent diameters were*. For a slope of 1, parent and daughter diameters are the same. A horizontal line indicates is no correlation, i.e., the daughter does not inherit the size from the parent.

Galton coined the notion “regression toward mediocrity” which today is in statistics known as “regression to the mean.”

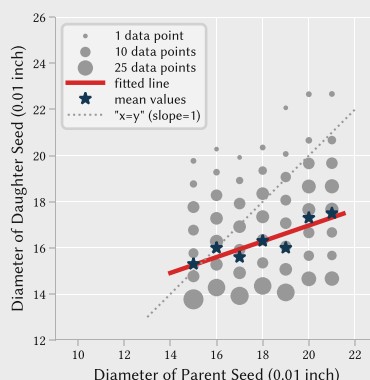
¹Galton, probably for practical reasons, did not use the mean values which would have been directly related to a least-square fit.

(continued)

Fig. 12.1 Sweet peas (or English garden peas) were used by F. Galton for his experiment on inheritance properties. The image shows peas in a pod. (© User:Atomicbre/Wikimedia Commons/CC-BY-SA-3.0)



Fig. 12.2 Size of the daughter seeds vs. size of the parent seeds as observed by F. Galton



12.1.2 Statistics Versus Machine Learning

Eventually, statisticians' regression analysis fused into today's machine learning regression—or, depending on the point of view, the field of ML borrowed some of these statistician tools. There is no clear point of time that one could mark as the birthday of machine learning regression. The history of regression makes it sometimes difficult to decide if we are talking about the statisticians' regression or the machine learning regression and even more so as the methods look nearly identical on a first glimpse.

Depending on the context, regression can have two slightly different meanings or objectives, which can make it difficult to see where and what the ML part is. These are the two different approaches which are explained using *linear* regression (i.e., finding the best fitting line):

- **Linear regression in the spirit of statistical analysis:** This is tantamount to fitting a line to given data (a procedure that also can be generalized to higher dimensions). “Fitting a line to data” translates into finding the line that fits best to the data. Putting this into a more mathematical language it implies to either analytically or numerically minimize some error.² There, the error typically consists of the sum of all squared deviations between the given data and the line, Sect. 11.7. Why to *square* the values? It turns out that for almost all minimization strategies, this is a surprisingly good choice that simplifies the respective formulations—even more than just using the sum of the absolute values would do (also see Sect. 11.7). This fitting procedure is called *least square* (LSQ) approach. The overall goal is to obtain an as small as possible error between line and given data. As opposed to the ML regression, this approach does not require any training or testing data.
- **Linear regression in the spirit of machine learning:** This approach has a different goal while at the same time still using some methods from statistics. Now, the goal is to make *predictions*, and the goal is *not* to obtain the best fit to given data. Predicting implies that the method can give a result for new data that has not been involved in the fitting processes. For this, the dataset is split into a training and a testing dataset (and sometimes also into a validation dataset as discussed later in Sect. 16.2). Then, the objective is to fit the line only to the training data. So far, we do not know how the model performs with new data which is why the error will be calculate with the *testing* data. The purpose of this ML regression model (or of *any* ML model) is to obtain a best performance on the test data.

Unfortunately, the differences are not always made clear in the (online and offline) literature and might leave the ML novice a bit confused. This becomes even worse when someone claims that splitting the data into training and testing sets is what they have always done to *evaluate the quality of the fit*. Was this then line fitting or machine learning? You can find this out by asking what the purpose was: if the purpose was to “prove” that there is a linear relationship between variables with a certain degree of significance and, e.g., obtain the inclination of the line and possibly call this “Young’s modulus,” then this was data fitting together with hypothesis-testing, which clearly belongs into the toolbox of statistics. If the goal was to create a “model” that represents generic aspects of some dataset and that is used to make predictions for new data that was not already used for *training* the model, then ML is at work.

For example, in GALTON’s experiment, the result of the linear regression was the slope of the line which was used for comparing it to a “reference slope” of 1, from which then a conclusion about the correlation of the size of the daughter to

² Also, the human eye is surprisingly good at finding a good fit of a line to given data—at least as long as the data is not too “unusual.” In any case, looking at a fitted line and the data is certainly a good “sanity check”!

the size of the parents could be drawn. *If* the purpose of the line fitting procedure would have been to predict the size of new daughter seeds based on a given size of a parent seed (e.g., someone gave me a pea seed and asks how big the child seed will become), then we need a model (e.g., a mathematical function) that takes a parent as input and returns a daughter size as output. Here, the model would be $y = mX + y_0$ where the input (or “feature”) is X , the output (or “target”) is y , and the model parameter are the slope m and the intercept y_0 . *Training the model* results in fixing these two parameters (here, $m = 0.34$ and $y_0 = 10.1$). *Making predictions* means to take a new value of $\hat{X}$ (note the hat indicates that this is unknown data) and evaluate the model $\hat{y} = h(\hat{X})$. The particular, parameterized function h is just an approximation, i.e., our “hypothesis,” as it will be introduced below. The true, “nature-given” function f is still unknown.

In the context of ML regression, there are a few different notions and concepts that are not always clearly separated. In the next section, we will define those and explain their interrelation.

12.2 General Concepts and Important Terminology

We will now go into more detail and introduce all aspects and tasks that are required for ML regression. In ML there is a whole zoo of different notions and community-dependent vocabulary. Therefore, we will also in this section introduce all used notions such that misunderstandings can be avoided or at least easily be spotted.

As already briefly touched upon in the previous section, there are three, general tasks during supervised regression which are shown in more details in Fig. 12.3. There, the first step is the choice of a specific *model class* which determines the form of the mathematical function (e.g., a polynomial of 4th order) of the ML model. The not yet determined parameter w_i are the *model parameter*, which we also sometimes write more explicitly as

$$h(X_1, \dots, X_N; w_1, \dots, w_n) \quad \text{or} \quad h_{w_1, \dots, w_n}(X_1, \dots, X_N) \quad (12.1)$$

There, the X_i are the N variables for which we assume that they play a role in describing the data. The choice of the model is important as it determines, e.g., its complexity and the type of behavior that can be represented. Nonetheless, it is a decision that *we* (as the modeler) have to make. As often, a good starting point is to use an as simple as possible model that still captures most of the aspects of interest of a dataset—another situation which reminds us of Occam’s razor from Sect. 1.2.5.

The second step is to find the parameters of the model function h (e.g., the coefficients w_i of the polynomial of choice) such that the function approximates the given training data well. This is typically an iterative process where the parameters are adjusted and the resulting training error is monitored simultaneously. The *training error* can be simply the sum or squared sum of all deviations between the target value and the predicted value by h (with the current set of parameters w_i)

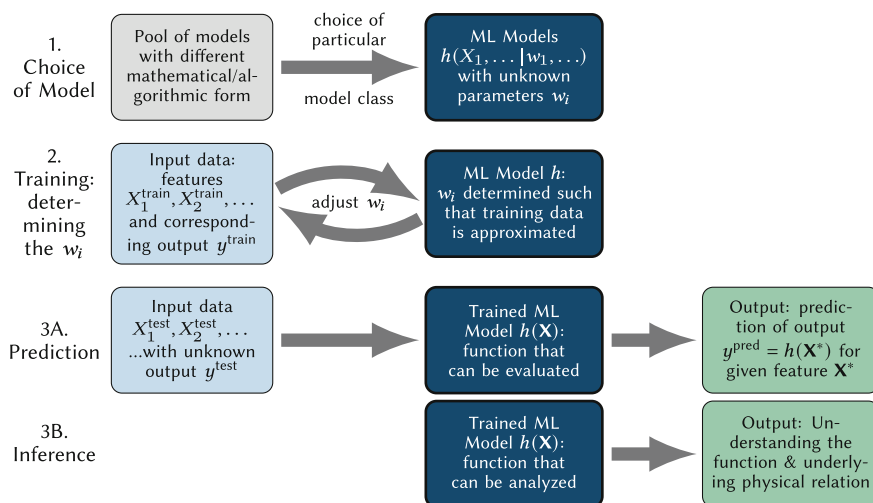


Fig. 12.3 Regression consists of three parts: (1) Selecting a suitable model, e.g., a polynomial function; (2) training the model, i.e., finding a suitable set of parameters $w_1, \dots, w_m$ (e.g., the coefficients of the polynomial) based on a given dataset consisting of input features and the output values for each data point; (3) making inference or predicting with the model h where for each input data the output value will be predicted

$$\text{training error} = \frac{1}{N} \sum_{i=1}^N (y_i - h(x_i))^2, \quad (12.2)$$

which we will discuss in more details below.

Once a set of model parameters was found that minimize the error between training data and curve, the model is trained and can be used either for making predictions (“what is the y value for a giving input $X_1, X_2, \dots$?”) or for inference, i.e., analyzing the general input-output relationship.

In the following, the most important notions and concepts for ML regression analysis are introduced. This is one of the fundamental sections that should not be skipped as the introduced concepts are also of relevance to later chapters.

As regression has its roots in statistical modeling and “parameter fitting,” this sometimes leads to some confusion concerning the relation between regression, inter- and extrapolation and “parameter fitting.” Figure 12.4 visually shows the difference and relation between them, and the next subsections will discuss these in detail.

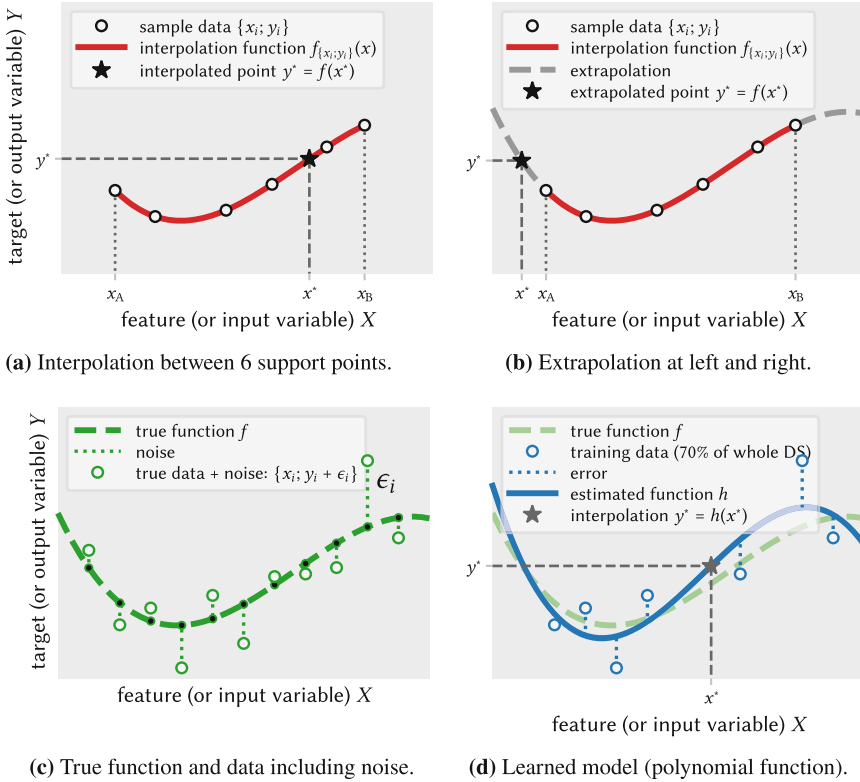


Fig. 12.4 Concepts and notions encountered in the context of machine learning regression: **(a)** interpolation between given data points with a polynomial; the curve goes exactly through these points. For any arbitrary point $x^* \in [x_A, x_B]$, one can get an interpolated value y^* . **(b)** uses the obtained polynomial to extrapolate the curve and thereby to reach points outside the range of the given sample dataset. The dashed line in **(c)** visualized a different concept than the ones shown in the previous panels. It represents the “true” function, e.g., a physics law, while the markers show measurements of that function which are affected by noise and inaccuracies. The plot shown in **(d)** shows a regression line h (in blue) that was obtained from the training data, here consisting of 70% of the data shown in **(c)**. The estimated function h minimizes the deviations to the training data and serves as an approximation of f

12.2.1 Interpolation

Interpolation is a type of function approximation where a number of data points are given, and we seek a function that passes exactly through *all* of the given points; see Fig. 12.4a. These points are also called “support points.” Having to enforce that the function passes exactly through all points is a strong restriction and only makes sense if the support points are not affected by noise.

If a polynomial is used as interpolant and N points are given, then the degree of the polynomial is $N - 1$. As an example, consider two points ($N = 2$): they define

a straight line, i.e., a linear function $y = ax^1 + b$ which has the polynomial degree of 1. Other commonly used interpolation functions are linear functions (as a special case of a general polynomial) or (piecewise) cubic splines. `numpy` offers an easy to use implementation based on the `Polynomial` class:

```
In [1]: from numpy import linspace
        from numpy.polynomial import Polynomial

        # fit a polynomial of degree 3 to the given support points
        poly = Polynomial.fit(x=[1, 3, 4, 7], y=[2, 5, 4, 8], deg=3)

        # get 20 interpolated y-values in the interval [1, 7]
        x_interp = linspace(1, 7, 20)
        y_interp = poly(x_interp)
        y_interp

Out [1]: array([2.         , 3.25397288, 4.15089663, 4.73538417, 5.0520484, ...])
```

Note that there, `poly` is a `Polynomial` object that contains already the coefficients determined during the “fitting” procedure. It also has a number of other useful properties and methods (take a look at the online `numpy` documentation [1] for further details).

12.2.2 Extrapolation

Obtaining values *beyond* the range of known observations is called *extrapolation* and is another kind of function approximation, cf. Fig. 12.4b. For given support points, one can, e.g., construct a polynomial function and use it to predict data outside the range of the given support points. This is, however, affected by uncertainties, called *Extrapolation error*, that are becoming larger the further away from the last support point the extrapolation is. In time series prediction, extrapolation is an important method where already data up to the present is available. Based on that, a future value is to be predicted. For extrapolation, the Python code would be the same as above; only the values of `x_interp` would need to be changed.

12.2.3 True Functions and Noise

Often, in statistical ML we assume at least implicitly that for a dataset under consideration, there exists an underlying *true function* h from which the data has been generated, possibly in conjunction with some statistical sampling procedure. Hence, one of the goals of an investigation would be to understand some characteristics of the true function through *statistical inference*; see Sect. 12.2.7. Such a true function f would be the mathematical representation of, e.g., a physical law which, however, in general is unknown. Almost always, f is additionally additively superimposed with *random noise* ϵ which, e.g., could stem from measurement errors or other uncertainties and can be written as

$$Y = f(X) + \epsilon \quad (12.3)$$

for the simplest case of a scalar function f , as also shown in Fig. 12.4c. If f returns a vector valued result, then each of the N_o components j is superimposed with an individual noise term, ϵ_j :

$$Y_j = f_j(X_1, \dots, X_n) + \epsilon_j \quad \text{with} \quad 1 \leq j \leq N_o. \quad (12.4)$$

Evaluating the function f at X together with the noise would then ultimately result in the data that we use, e.g., for training a ML model. The trained model can then help to understand aspects of the true function (see the explanations of *inference* in Sect. 12.2.7). In Fig. 12.4c the true function is indicated as a dashed line and also shown in Fig. 12.4d.

12.2.4 The Training Process and the Machine Learning Model

In the context of ML regression, *training of a ML model* means to identify a function $h(x)$ that fits best to the data under consideration (we chose the letter h because below this will be introduced as a *hypothesis*). This requires that already a functional form for h has been chosen which defines the general mathematical structure of the model function. For example, one could choose as function class (which later will be introduced as *hypothesis class*) all polynomials of the shape

$$h(x) = w_0 + w_1x + w_2x^2 + \dots + w_nx^n. \quad (12.5)$$

Training then determines the set of parameters $\{w_0, w_1, \dots, w_n\}$ such that $h(x)$ fits best to the training data. To differentiate between the general functional form with unknown parameters and a function with already determined parameters, one may also write for the latter $h(x; w_0, w_1, \dots, w_n)$ which is pronounced as *h evaluated at x for the given parameters $w_0, \dots$* .

For a visualization see Fig. 12.4c where the solid line shows the trained model which approximates the given training data. However, the goal is that the trained model not only approximates the given training data but that it captures the relevant aspects of the underlying true function such that the *trained model h* is a reasonable estimate of the true function f . The training error results from the vertical “deviations” of h and the noisy data; typically, this error is computed as the *mean squared error (MSE)* (cf. Sect. 11.7.3).

For training such a model, only the line of Python code `poly = Polynomial.fit(...)` from the code used for interpolation in Sect. 12.2.1 would be required

```
In [1]: from numpy.polynomial import Polynomial
        poly = Polynomial.fit(x=[1, 3, 4, 7], y=[2, 5, 4, 8], deg=3)
        poly.coef
```

```
Out [1]: array([ 4. , -3.375,  1. ,  6.375])
```

where the four output numbers are the coefficients of the polynomial. For example, the “4.” corresponds to the constant term of the polynomial, and the corresponding polynomial is $4 - 3.375x + x^2 + 6.375x^3$.

12.2.5 Hypothesis/Hypothesis Space and the Parameter Space

We now introduce notions that originally stem from the rather theoretical field of inductive logic programming and which are regularly encountered in the literature, sometimes without being well-defined, though: the *hypothesis*, the *hypothesis space*, and the *parameter space* (see, e.g., [3])

A *hypothesis* in the context of ML denotes something that is nearly identical to a trained model. To see the difference, let us first take a look at the definition of the *hypothesis space*:

Definition 12.1 (Hypothesis Space) The *hypothesis space* $\mathcal{H}$ consists of all functions h that map an element x_i from the dataset to a target value or label, y_i . The functional form or mathematical structure of these functions is fixed, and two different hypotheses, taken from the hypothesis space, differ only in the particular choice of parameters.

Usually, for the hypothesis space, the letter $\mathcal{H}$ is used. An example for $\mathcal{H}$ is a cubic polynomial with parameters $w_0, \dots, w_3$. There the hypothesis space consists of cubic polynomials that differ only in the particular numerical values of the four parameters w_i . Closely related is the *parameter space* which is defined based on elements of the hypothesis space:

Definition 12.2 (Parameter Space or Weight Space) The set of all parameter combinations used for the functions contained in the hypothesis space are spanning the *parameter space* which is also sometimes called *weight space*.

In the next section, we will see how the parameter space can be used as a way to visualize aspects of the hypothesis space.

How does this relate to machine learning? The ML algorithm chooses elements of the hypothesis space and determines the most suitable hypothesis—the trained model. However, the notions of a model and a hypothesis have slightly different connotations: “model” often implies the concrete implementation, while “hypothesis” implies that we have an idea about an underlying (usually) mathematical form of the input-output relation that we express through the hypothesis h . In particular, when we are talking about “using a hypothesis h to approximate the true function

f ,” then the word *hypothesis* suggests that it can be tested and that there even might be a connection to an underlying theory.

12.2.6 Predictions

A sometimes misunderstood or confusing notion is that of *prediction*. In the context of mathematical function approximation, prediction is often used synonymously with extrapolation. In Fig. 12.4b the dashed lines show such “predictions” of how the function would continue outside the region defined by the support points. However, in the context of ML the notion of prediction is used differently: there, it means to use the trained model (the estimated function h) for computing the dependent variable based on given input data *that has not been used during the training process*. Typically, this implies that the values of x used for prediction is roughly in the same range as the training data and the emphasis is not on extrapolation. As this is a crucial point, we highlight it here again.

Words of advice



In the context of ML you will often encounter phrases such as “the performance of a model is estimated using new data” or “we make predictions for previously unseen data.” This does *not* mean that with a ML regression model, we can ask for a prediction for *any* input data.

If the regression model has been trained with features in a certain range of input values, e.g., from within $x_A, \dots, x_B$, then typically we still don’t know much about the behavior of the data outside that range, and we cannot (or only in a very limited way) make predictions for completely new data that are outside $x_A, \dots, x_B$.

12.2.7 Inference

The notion of *inference* is not always used consistently across different communities. For example, in the deep learning community, inference is sometimes used as a synonym for making predictions because “inferring a value” is mostly interpreted as “predicting a value.” However, in this text, we take a perspective that is influenced by the statisticians view:

Definition 12.3 (Inference) Inference is understood as a concept quite distinct from prediction. While the goal of prediction is to use a trained model to estimate the value of an output variable for a given input, inference has its origins in the field of statistics and refers to trying to understand (aspects of) the relation between the feature (i.e., the input of a model) and the target (i.e., the output) of a model.

Typical questions that are answered during *making inference* are questions that elucidate structural or mathematical aspects of the model, e.g., how does the output change when the whole input or a specific feature changes in a specific way? Is there a linear relationship or is it nonlinear? Which are the relevant features? As the model represents relevant aspects of the physical reality, inference often also helps to get insights into aspects of the underlying physics. It is thus quite different from prediction, which boils down to evaluating a function for a given input value.

Most of the regression models used in statistical ML are suitable for making inference because there, at least roughly speaking, the *model* essentially is tantamount to a *mathematical function* which can be easily interpreted in terms of mathematical aspects but also in terms of physical implications.

12.2.8 Under- and Overfitting

Underfitting and overfitting are two notions that you will encounter frequently in ML because they are directly related to the quality of a ML model. *Underfitting* takes place if the fitting function that was used as approximation is too stiff and is not able to adapt sufficiently to the data. If the fit function mostly captures the noise of the data and not so much the general trend of the data, as a very flexible curve would do, then *overfitting* occurs. The concept of under- and overfitting is visualized in the left and right panel of Fig. 12.5. A good approximation in the sense of a good ML model is a model that is able to ignore the irrelevant fluctuations or noise and in which the relevant behavior of the underlying physics becomes visible, as shown in the middle panel of Fig. 12.5. What is relevant and what is not can be difficult to judge, but below we will at least learn some techniques that help us to determine an optimal degree of line stiffness.

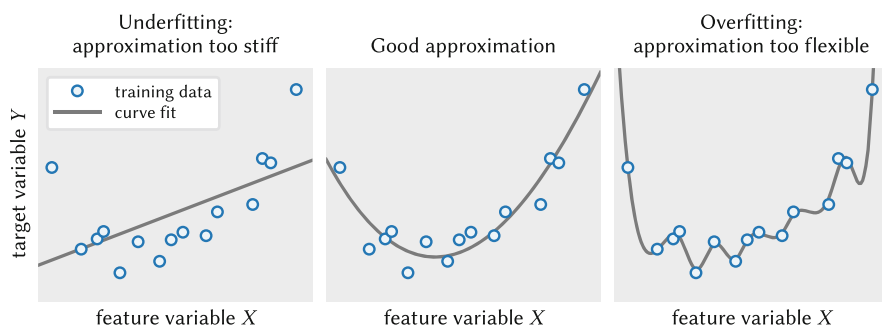


Fig. 12.5 The concepts of underfitting and overfitting: The line in the left panel is too stiff and does not sufficiently capture the relevant details, it underfits the data. The right panel shows overfitting where the too flexible line meanders through every data point and also irrelevant details of the particular data distribution are reflected line. The middle panel shows an approximation that is neither too stiff nor too flexible

12.2.9 Notations for Input and Output Variables

In this chapter, we adhere to the notations and conventions already introduced in Chap. 3. Here we only briefly recall these notations and will put them into the context of ML regression.

- Input variables (e.g., the x in $h(x)$) are often also called *independent variables*, *features*, or just *variables*, all of which will be used in this text. In the regression literature, they are sometimes also referred to as *predictors* or *explanatory variables*.
- If there is only one input variable, we denote it by a capital X . If there are multiple features, we use X_i for the i -th feature.
- Typically, a dataset consists of several data records or samples with several features. In that case X_i is the name of the i -th feature and denotes a column vector where each entry corresponds to a sample of the input variable i . Even though that this is in fact a column taken from a matrix of input data and vectors in general are indicated by boldface letter, in this case, the character of the variable corresponds to a scalar value, and we denote the i -th feature by X_i with

$$X_i = [X_{1i}, \dots, X_{mi}]^T. \quad (12.6)$$

where m is the number of rows.

- The output variable of a ML model is Y , i.e., the numerical value that we seek to predict. It is also called *label* or *target*. Similar to the input variable, it may consist of several components in which case we use indices, e.g., Y_i .
- A single data record i consisting of values for all features is written as lowercase $\mathbf{x}_i$, similarly y_i for the record i from the outputs.
- In this chapter we typically have a single or multiple (e.g., n) features and only a single output. Then the pair $(x_i; y_i)$ or $(x_{i1}, \dots, x_{in}; y_j)$ are the “coordinates” of the j -th data point for the feature number i . Each of such pairs is called an *instance* or an *example*.
- The training dataset $\mathcal{D}^{\text{train}}$ is then the set of all N training instances

$$\mathcal{D}^{\text{train}} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N, \quad (12.7)$$

the testing dataset $\mathcal{D}^{\text{test}}$ is defined analogously.

12.2.10 Training and Testing: Datasets and Errors Types

One of the important concepts in ML is to split the whole dataset into a *training dataset* and a *testing dataset* (and sometimes even a *validation dataset* as introduced in Sect. 16.2 which is required for cross validation). The train-test split is sometimes also called *holdout method*. It divides the whole dataset into two disjunctive subsets,

the training and testing dataset, where each subset was obtained from statistical sampling of the whole dataset (for other testing strategies, please refer to Sect. 16.2). The purpose of the training dataset is to train the ML model.

The trained model can be used to make predictions for new data, including the testing data. Such “new data” would consist of only the features, and the prediction would give the target values. The purpose of the testing dataset is to enable an estimate of how good the predictive capabilities (i.e., predicting target values) of the model are and how well the model “generalizes,” i.e., how well it works with new data. It is important that this estimation is done with data that has not been used during the training process because otherwise the model might have already “memorized” the data. Figure 12.6 shows the train-test split along with the training and testing errors for each point. For computing the overall *training error*³ of all N data points, one could use the MSE of the output of a particular $h(x)$ and the training data such that

$$\text{training error } \mathcal{E} = \frac{1}{N} \sum_{i=1}^N \left(h(\mathbf{x}_i^{\text{train}}) - y_i^{\text{train}} \right)^2 \quad (12.8)$$

where $h(\mathbf{x}_i^{\text{train}})$ uses the trained ML model h to predict the output for the given training input $\mathbf{x}_i^{\text{train}}$. The target values y_i^{train} are the output used for training. During the training process, the training error $\mathcal{E}$ (typically the MSE as also in the equation above) is minimized, e.g., by iteratively adjusting the model parameters. The *testing error* of all M testing data points is in analogy to Eq. (12.8) given by

$$\text{testing error } \mathcal{E} = \frac{1}{M} \sum_{i=1}^M \left(h(\mathbf{x}_i^{\text{test}}) - y_i^{\text{test}} \right)^2 \quad (12.9)$$

Here, h has already all parameters fixed from the training and is only evaluated at the given data $\mathbf{x}_i^{\text{test}}$, which we above introduced as *making predictions*. Below, we will discuss in details how different levels of training and testing error values can be interpreted.

To observe the influence of overfitting on the training and testing performance, we use the same dataset as in Fig. 12.6, but this time use a very flexible model, as shown in Fig. 12.7. Even though the flexible curve seems to fit perfectly well to the training data (middle panel of Fig. 12.7), the curve necessarily will exhibit larger deviations from the testing data (right panel of Fig. 12.7); it is said that the model does not generalize well because it is too specialized toward a particular dataset. This often also implies that the model is fairly complicated (in the figure, a polynomial of degree 13 was chosen as compared to a quadratic function in Fig. 12.6). We can also compute the resulting training and testing MSE values: for

³ In Sect. 12.3.3, we will see that there are different types of “errors,” but for now we use *error* just as a synonym for any sort of deviation.

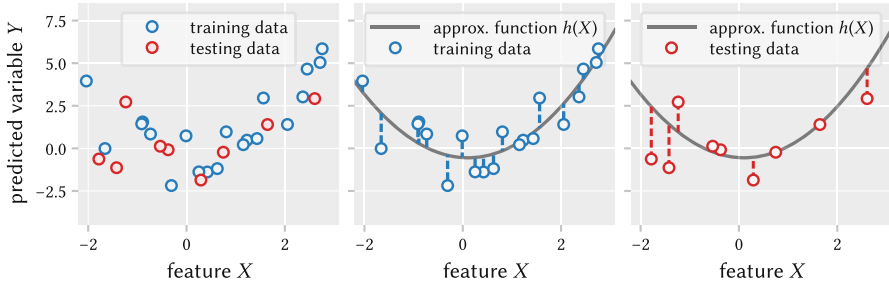


Fig. 12.6 The left panel shows the full dataset that has been split into a training and a testing set. Only the training dataset was used to fit a polynomial function, resulting in the trained model h . In the middle panel, the dashed lines indicate the deviations for each point that altogether results in the *training error*. To evaluate how this model performs with new, previously unseen data, the *testing error* was evaluated in the right panel, consisting of the distances of the testing data to the model response using h

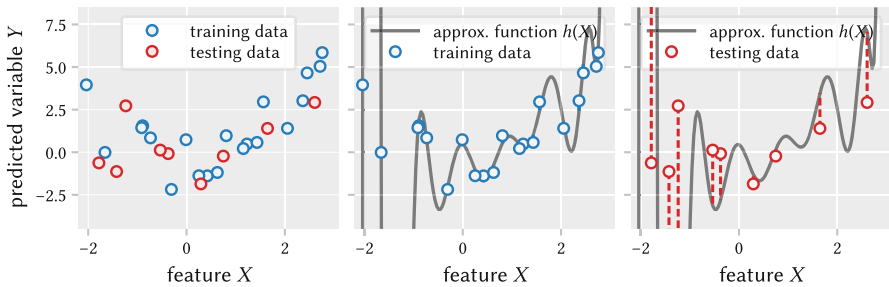


Fig. 12.7 The same dataset as in Fig. 12.6 is used together with a much more flexible polynomial that fits the training data quite nicely but introduces larger errors for the testing data

the balanced model, we have $\text{MSE}^{\text{train}} = 1.1$ and $\text{MSE}^{\text{test}} = 2.7$ both of which have the same order of magnitude, while for the flexible model we obtain $\text{MSE}^{\text{train}} = 0.1$ and $\text{MSE}^{\text{test}} = 7098$ which are off by nearly five orders of magnitude!

Therefore, usually the best model is the one that is as simple as possible and which at the same time also shows a good generalization behavior. If the model is too flexible it does not generalize well to the testing data. And what happens if the model is too simple? Then both training and testing errors will be large. We will investigate this behavior more systematically in the sections below.

Typically, the training dataset is larger than the testing dataset, e.g., 70% or 80% of the dataset are chosen for training; the remainder is used for testing. A simple Python implementation of splitting a dataset is given in Python Listing 12.1. This function takes the feature and target matrix as input together with the fraction of the data to be used as training data. A fourth function parameter allows to specify a seed for the random number generator (line 10): specifying an integer number ensures that the random shuffling in line 15 is reproducible. Without the seed, new (pseudo) random numbers are generated for every call of this function. In line 14,

```

1 import numpy as np
2
3 def train_test_split(X, Y, fraction, seed=None):
4     """Returns a training and a testing dataset
5
6     X and Y are numpy arrays with as many rows as samples, fraction (=0..1) defines
7     the relative amount of training data, and seed is the initialization of the
8     random number generator.
9     """
10    rng = np.random.default_rng(seed)          # initialize random number generator
11    n_total_data = X.shape[0]                  # number of data in the full DS
12    n_training = int(fraction * n_total_data)   # use fraction of data for training
13
14    indices = np.arange(n_total_data)          # range of integers for indexing X and Y
15    rng.shuffle(indices)                       # shuffle elements of "indices" in place
16    X = X[indices]                             # X and y are shuffled such that each X[i]
17    Y = Y[indices]                             # ... is still associated with the same Y[i]
18
19    X_training = X[:n_training]                # use the first "n_training" points for the
20    Y_training = Y[:n_training]                # training dataset and ...
21    X_testing = X[n_training:]                 # the rest of the data for testing.
22    Y_testing = Y[n_training:]
23
24    return X_training, Y_training, X_testing, Y_testing

```

Python Listing 12.1 Python function for performing the train-test split. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact

the implementation uses `numpy.arange` to create an array with numbers ranging from 0 up to (and not including) number of samples. Each of these numbers will be an index to an element in the arrays `x` and `y`. By randomly “shuffling” the indices in line 16 and 17, we can therefore obtain random samples from both `x` and `y` such that each element of `x` still corresponds to the same element of `y`. Last but not least in lines 19–22 we use the first `n_training` rows as training data and the remainder as testing data. It follows a short example that shows how to use the function. There, we assume that the function has already been imported and is available. We start by defining a dataset consisting of one-dimensional array for `X` and `Y`:

```

In [1]: import numpy as np
        fraction = 0.6          # fraction of training data from the whole dataset
        n_total_data = 10       # number of data in the full DS

        X = np.arange(n_total_data)      # input data
        Y = 10 + np.arange(n_total_data) # output/target data

```

Subsequently, the function for performing the train-test split is called. If the function is called with always the same integer number for the seed, then the random number generator always results in the same sequence of random numbers. As a consequence, the train-test split will always result in the same datasets which is useful, e.g., for debugging code:

```

In [2]: X_train, Y_train, X_test, Y_test = train_test_split(X, Y, fraction)

```

And finally, we print the different datasets:

```
In [3]: print('X =', X, ' Y =', Y)
        print('X_train =', X_train, ' Y_train =', Y_train)
        print('X_test =', X_test, ' Y_test =', Y_test)

Out [3]: X = [0 1 2 3 4 5 6 7 8 9] Y = [10 11 12 13 14 15 16 17 18 19]
        X_train = [0 3 9 7 6 4] Y_train = [10 13 19 17 16 14]
        X_test = [8 2 1 5] Y_test = [18 12 11 15]
```

This code can be used as a starting point for some further experiments such as plotting the training and testing data, varying the number of data records or the percentage of training and testing data, and observing how representative the testing dataset is for the training dataset.

Things to Remember about concepts and terminology



- Beware of the distinction between interpolation and extrapolation, prediction and inference, and functions, models, and hypotheses.
- Also beware that some notions might be used in other communities slightly differently or with slightly different connotations.
- Using training and testing data is essential for machine learning and allows to compute training and testing errors that help to differentiate between “good” and “bad” models.
- The best model is one that is as simple as possible and that still has a good generalization performance w.r.t. to new data, such as testing data.

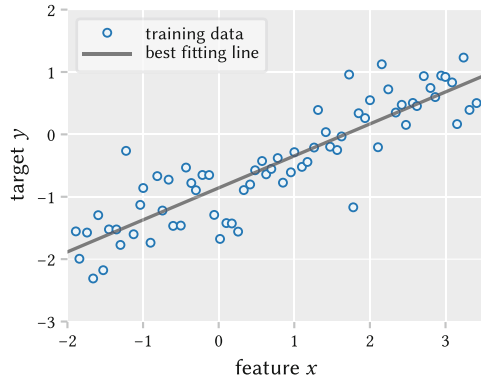
12.3 Simple Linear Regression

In this section we start by introducing the simplest type of regression models: the problem under consideration is called *simple linear regression* if it has only one scalar input variable and also returns a scalar variable as output. If the problem has more than one independent variables, it is called *multiple linear regression*. The attribute *linear* indicates that the model is a *linear* function of the form

$$h(x; w_0, w_1) = w_1x + w_0 \quad (12.10)$$

There, w_1 and w_0 are the parameters to be learned from the data, also called *coefficients*. As this equation represents the equation for a line, w_0 is also called *intercept* (i.e., the intercept of the line with the y-axis), and w_1 is the slope of the line.

Fig. 12.8 Linear regression with a single feature variable and a single target variable: for given data, we seek the line that fits best to the data



The Link to Neural Networks

In the context of neural networks (cf. Chap. 17), w_0 would typically be called *bias* b .



Linear regression essentially fits a line to a set of given data points. Thereby, the whole dataset is approximated by a simple line, cf. Fig. 12.8, but not necessarily a *straight* line, as discussed in the following section. Such a line is useful for showing trends in datasets and for emphasizing, e.g., the position of outliers. In the context of ML, however, we are rather interested in using the function of the line for making predictions.

12.3.1 The Linearity in Linear Regression

One might ask if a *linear model* isn't extremely limited in terms of the phenomena that can be represented. Luckily, *linearity* is not related to the shape of the curve. In a linear model, only the parameters have to occur in a linear form; however, variable(s) may occur as arbitrary terms. Unfortunately, due to historical reasons, the literature is full of figures and plots showing straight lines which suggest that this is where the attribute “linear” stems from. Hence, the differentiation between nonlinear and linear models and their meaning might be somewhat counterintuitive because in ML, unlike in common mathematical functions, the (non)-linearity is an attribute of the *parameters* and not the *variables*. To make this a bit more clear, here are some brief examples:

This has the consequence that a linear model is not only able to predict linear relationships between features and the target variable (e.g., a straight line or a plane in one-dimensional or two-dimensional feature space, respectively) but also nonlinear behavior—at least as long as the weights are just simple, linear

- $h(x) = w_1x + w_0$... is a linear model because each w_i occurs only as a factor.
- $h(x) = w_1x^2 + w_0$... is still a linear model as both parameters (i.e., the “coefficients”) occur as linear terms, regardless of the x^2 term.
- $h(x) = (w_1)^2x + w_2$... is a nonlinear model because w_1 occurs as quadratic term.
- $h(x) = w_0 \cos(w_1)x + w_0$... also is nonlinear as the coefficient of x is a product of w_0 and the already nonlinear function $\cos(w_1)$.

coefficients. Section 13.1 will go into more details concerning nonlinear model behavior.

12.3.2 Hypothesis Space and Parameter Space

After the model type has been determined, each combination of parameters results in a particular hypothesis about the true function. Here, we assume that our design choice led to the linear equation Eq. (12.10). Then the hypothesis space $\mathcal{H}$ (cf. Definition 12.1) contains all linear functions $h(x)$ with particular values of w_0 and w_1 :

$$\mathcal{H} = \{h(x) = w_1x + w_0 \mid w_0, w_1 \in \mathbb{R}\} . \quad (12.11)$$

The set of all parameter combinations (w_0, w_1) spans the *parameter space* which is sometimes also called *weight space*. As the functional form of all contained hypotheses does not change, the parameter space can be used as an alternative to the hypothesis space which can be more easily visualized than the hypothesis space itself. A demonstration is shown in Example 12.2.

Example 12.2 (Linear Regression: Data Space and Weight Space)

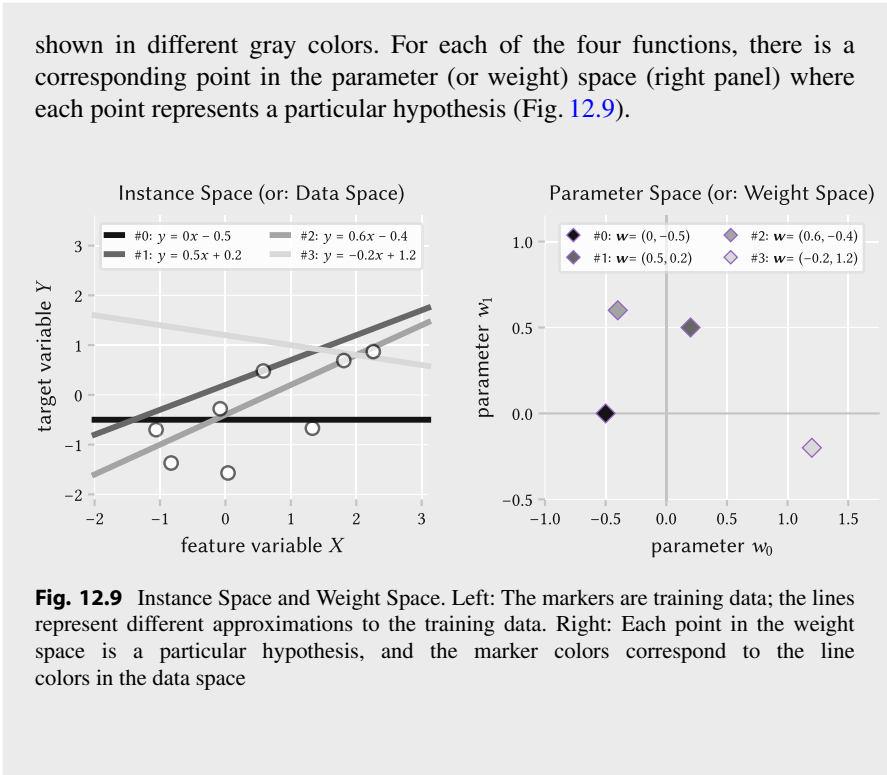
Given is a training dataset which is also shown by the markers in the instance space (left panel):

$$X = [-1.06, -0.83, -0.08, 0.04, 0.58, 1.33, 1.81, 2.26]$$

$$Y = [-0.70, -1.37, -0.28, -1.57, 0.48, -0.67, 0.69, 0.87].$$

Additionally, a number of lines with different parameters are used to approximate the underlying true function of the dataset. The four lines are

(continued)



The weight space is also a way to connect to physical knowledge that we already may possess about the investigated situations, e.g., there may be physical constraints restricting the range of the parameters which then also can be directly visualized in the parameter space. In Example 12.2 we additionally introduced the notion of the *instance space* which is sometimes also called *data space*: it is the space in which the whole dataset “lives.” With these definitions of different spaces, we are now able to precisely express what kind of quantity and relation we are referring to. Before we turn to estimating which hypothesis is most suitable to represent the data, we briefly summarize the new concepts and notions:

Things to Remember about simple linear regression and hypothesis space ★

- a hypothesis is a particular model function where the weights are determined and fixed;
- the hypothesis space often contains infinitely many functions and is therefore usually not suitable for practical purposes;
- the concept of parameter/weight space is useful for compact visualization of the hypotheses and can also visualize physical constraints of the parameters.
- despite the name, linear regression models can model nonlinear relations, the linearity concerns only the weights;

12.3.3 Estimating Deviations: Errors and Residuals

We now turn to the following questions: Given a specific hypothesis, how can we determine the “goodness” of the fit to the data? Or How can we determine the weights such that they minimize an error?

To answer that, we first of all need to clarify the concept of *errors*: What our common sense has called “error” can, in the context of statistics, be further specified, as there exist two closely related error concepts. The *statistical error* is a measure for the whole population and not just the sample (e.g., our training dataset). In practical situations, this quantity usually cannot be computed, because we don’t have exact knowledge about the whole population. The *Residual error*, on the other hand, is based on the sample data only and thereby serves as an estimate for the (non-observable) statistical error. This difference becomes particularly relevant when we use a trained model as approximation for the true behavior. Coming back to the concrete, simple regression problem, we can define the residual as follows:

Definition 12.4 (Residual or Residual Error) The residual r_i of a data record i is defined as the difference between the actual and the predicted target value

$$r_i = y_i - h(x_i; w_0, w_1) , \quad (12.12)$$

or for general numbers of weights and features

$$r_i = y_i - h(\mathbf{x}_i; \mathbf{w}) . \quad (12.13)$$

The prediction is done based on a hypothesis function h , and y_i is a given target value (which can be easily generalized to multiple target values).

We observe that the residual does not involve the true function f . While at least a statistician would only denote the difference between $f(x_i)$ and y_i as *error*, in the ML community this distinction is only rarely made as the true function is generally unknown. Note that this residual formulation takes only the distance of the *target* values into account, which is here the vertical distance from the training point to the line.

About the “distances” used in residuals

There are other approaches than using the vertical distance, e.g., in Sect. 15.2 we discuss the method of principal component analysis that on a first glimpse has some similarity to regression analysis but that uses an Euclidean shortest distance as measure as for the quality of a fit approximation.

12.3.4 The Loss Function and the Loss

The *loss function*, similar to the residual error, characterizes how good or bad the agreement is between a prediction and the given target value. We define it as follows:

Definition 12.5 (Loss Function and Loss) The loss function $\mathcal{L}$ is a function acting on each record of the dataset which characterizes the amount of agreement that was lost. It results in one number, the loss value $\mathcal{L} \geq 0$, where a value of zero indicates perfect agreement between predicted value and given value (the ground truth).

Here are some examples for commonly used loss functions:

$$\begin{aligned}\mathcal{L}^{\text{abs}}(\mathbf{w}|\mathcal{D}^{\text{train}}) &= |r_i| = |y_i - h(x_i; \mathbf{w})| && \dots \text{absolute loss} \\ \mathcal{L}^{\text{sqr}}(\mathbf{w}|\mathcal{D}^{\text{train}}) &= (r_i)^2 = (y_i - h(x_i; \mathbf{w}))^2 && \dots \text{squared/quadratic loss} \\ \mathcal{L}^{\text{hinge}}(\mathbf{w}|\mathcal{D}^{\text{train}}) &= \max(0, 1 - y_i h(x_i; \mathbf{w})) && \dots \text{hinge loss} .\end{aligned}$$

For regression problems, the squared loss is the most commonly used form, while the hinge loss occurs in classification problems using support vector machines. It is listed here for the purposes of showing that the loss function can but doesn't have to be based on the residual.

Choosing a suitable function is important as it has to be “compatible” with the goal of the training and the problem at hand. For the ML regression problems

considered in this chapter, however, using the quadratic loss function works well. Section 11.7 has an overview and a discussion of different types errors and deviations.

12.3.5 The Cost Function

The *cost function* (represented by a calligraphic “ $\mathcal{J}$ ”) quantifies how good (or rather how bad) the *whole* set of target values agree with all corresponding predicted values. Here is a definition.

Definition 12.6 (Cost Function) The cost function (or error function) $\mathcal{J}$ maps *all* given data points to a single, real number $\mathcal{J} \geq 0$. It quantifies the “badness” of the overall agreement of a particular model approximation or hypothesis, where larger numbers indicate worse agreement. Typically, it is the mean value of all N_s individual loss values of a dataset:

$$\mathcal{J}(\mathbf{w}|\mathcal{D}^{\text{train}}) = \frac{1}{N_s} \sum_{i=1}^{N_s} \mathcal{L}(\mathbf{w}|x_i, y_i) \quad (12.14)$$

Unless stated differently, throughout this chapter, we assume that $\mathcal{L}$ is the quadratic loss such that $\mathcal{J}$ is obtained as the [MSE](#).

The goal of the training is to minimize the cost function through the best choice of parameters w_i . Using a loss function $\mathcal{L}$ that is always ≥ 0 is advantageous, as then loss contributions for different records do not cancel each other out. Using a squared loss as compared to the absolute loss has the additional benefit that the function’s minimum is differentiable, resulting in a mathematical form that later on becomes much simpler than those from essentially any other formulations.⁴ Note that this formulation is nothing but the training [MSE](#). Using such a squared loss function for finding best fitting parameters lent itself to the name of the well-known *least square least square (LSQ) method*.

⁴ Hint: For finding the minimum, we have to take the derivative. Then, the the squared residuals simply becomes just the residual—a very simple function.

Words of advice

In the deep learning community, the two notions *loss* and *cost* are not always used as we do it here, and the “training loss” in fact denotes the *average* training loss of a batch, i.e., the cost. Sometimes the loss function also denotes what we introduced above as residual; the cost function however *always* denotes the function that maps all given data points to a single real number that represents the “badness” of the current approximation or assumption.

Last but not least, there is another related notion that of the *objective function* which stems from mathematical optimization theory and is the most general term for any function that can be optimized. It can be the same as the cost function, but it sometimes also contains an added regularization term for numerical reasons. In this text, we will mostly avoid to use the notion “objective function.”

We are now ready to return to the problem of simple linear regression. To find weights for which the regression line fits best to the data, the cost function should be minimized for the given training data and with respect to the weights

$$\mathcal{J}(w_0, w_1 \mid \mathcal{D}^{\text{train}}) \rightarrow \text{MIN} , \quad (12.15)$$

in which the notation already indicates that the weights are now considered as the “variables” and the training data are given and fixed. Using the average of the squared residuals is identical to computing the *mean squared error* (MSE), and the *training cost function* takes the following form:

$$\mathcal{J}(w_0, w_1 \mid \mathcal{D}^{\text{train}}) = \frac{1}{N} \sum_{i=1}^N (y_i - h(x_i; w_1, w_0))^2 = \frac{1}{N} \sum_{i=1}^N (y_i - (w_1 x_i + w_0))^2 , \quad (12.16)$$

where the training dataset consists of the N data points (x_i, y_i) with $1 \leq i \leq N$. Figure 12.10 shows how changing the inclination w_1 of a line changes the error value. In particular the right panel shows that the cost function has a distinct minimum.

In analogy to the mean training loss, we can also obtain *testing cost* function. However, now the weights are considered as fixed and are taken as the ones that resulted from the training. Then, the testing cost function is defined as

$$\mathcal{J}(w_0, w_1, \mathcal{D}^{\text{test}}) = \frac{1}{N} \sum_{i=1}^N (y_i^{\text{test}} - h(x_i^{\text{test}}, w_1, w_0))^2 . \quad (12.17)$$

To minimize the training error $\mathcal{J}(w_0, w_1 \mid \mathcal{D}^{\text{train}})$, a number of different approaches can be used. A “brute force” approach is the so-called *grid search* method where systematically all relevant parameters are varied, e.g., by subdividing each parameter range into n values, until the minimum is found. In all generality,

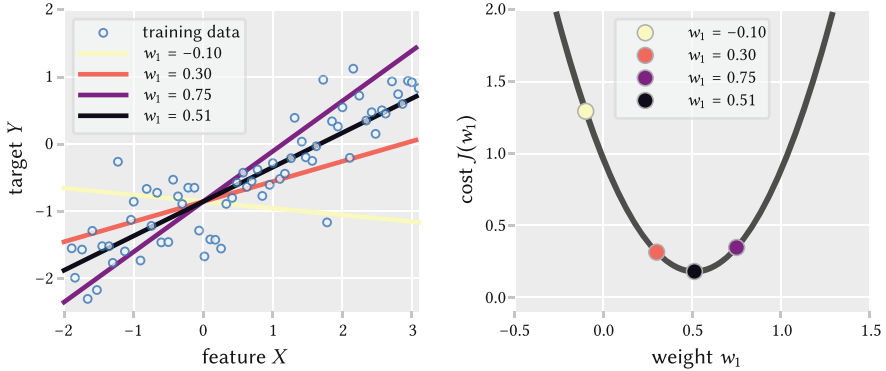


Fig. 12.10 Lines with different inclinations w_1 (top left) and the cost function for the ideal, fixed value w_0 and varied w_1 . The cost values of the straight lines are indicated by the four markers

for a method with k weights, this results in n^k combinations (see Sect. 5.1.6)—a number that very quickly cannot be handled anymore.

A general approach that is more efficient than a grid search and also applicable for many other ML problems is to perform a numerical minimization where the values of the parameters are determined iteratively. Mathematically speaking, a necessary condition for finding a function minimum is that the first derivative is zero. The first derivative of Eq. (12.17) is a derivative w.r.t. the weights

$$\frac{\partial \mathcal{J}(w_0, w_1 | \mathcal{D}^{\text{train}})}{\partial w_0} = 0 \quad \text{and} \quad \frac{\partial \mathcal{J}(w_0, w_1 | \mathcal{D}^{\text{train}})}{\partial w_1} = 0. \quad (12.18)$$

Insertion of Eq. (12.16) and replacing the hypothesis h with y^{pred} gives

$$\frac{\partial \mathcal{J}(w_0, w_1 | \mathcal{D}^{\text{train}})}{\partial w_0} = \frac{2}{N} \sum_{i=1}^N \left(y_i - \underbrace{(w_1 x_i + w_0)}_{=h(x_i; w_1, w_0)} \right) (-1) = -\frac{2}{N} \sum_{i=1}^N (y_i - y^{\text{pred}}_i) \quad (12.19)$$

$$\frac{\partial \mathcal{J}(w_0, w_1 | \mathcal{D}^{\text{train}})}{\partial w_1} = \frac{2}{N} \sum_{i=1}^N \left(y_i - \underbrace{(w_1 x_i + w_0)}_{=h(x_i; w_1, w_0)} \right) (-x_i) = -\frac{2}{N} \sum_{i=1}^N (y_i - y^{\text{pred}}_i) x_i \quad (12.20)$$

These expression can then be used with a numerical method to iteratively determine the values of the weights for which these two equations are zero. This will be introduced in Sect. 12.3.7.

For the particular linear problem considered here, it is also possible to obtain the result using a direct mathematical derivation which is computationally fast and also useful for validating the numerical approach. Therefore, in the following

subsections, we start with a closed form derivation for minimization of the cost function before the numerical approach will be explored.

12.3.6 Cost Function Minimization: Analytical Derivation

To find the parameters w_0 and w_1 that minimize the cost function $\mathcal{J}(w_0, w_1 | \mathcal{D}_{\text{train}})$, we take the first derivative with respect to the first and the second parameter and look for the parameter values that result in zero. For finding w_0 , this equation then reads:

$$\frac{\partial \mathcal{J}(w_0, w_1 | \mathcal{D}_{\text{train}})}{\partial w_0} = 0. \quad (12.21)$$

Insertion of the quadratic loss function from Eq. (12.16), we obtain

$$2 \frac{1}{N} \sum_{i=1}^N (y_i - (w_1 x_i + w_0)) (-1) = 0, \quad (12.22)$$

where the summation is over all items of the training dataset. Realizing that $\sum_{i=1}^N x_i / N$ is the mean value of the x_i and therefore replacing it with the abbreviation $\bar{x}$, we get

$$-\bar{y} + w_1 \bar{x} + w_0 = 0 \quad (12.23)$$

$$\Leftrightarrow w_0 = -w_1 \bar{x} + \bar{y}, \quad (12.24)$$

where only w_1 is unknown. Therefore, we again compute the first derivative, but this time with respect to w_1 , set the resulting term to zero, and also insert the previous result:

$$\frac{\partial \mathcal{L}(w_0, w_1 | \mathcal{D}_{\text{train}})}{\partial w_1} = 0 \quad (12.25)$$

$$\Rightarrow \frac{1}{N} \sum (y_i - (w_1 x_i + w_0)) x_i = 0 \quad (12.26)$$

$$\Leftrightarrow \sum y_i x_i - w_1 \sum (x_i)^2 - w_0 \sum x_i = 0 \quad (12.27)$$

$$\Leftrightarrow -w_1 \sum (x_i)^2 - (-w_1 \bar{x} + \bar{y}) N \bar{x} = - \sum x_i y_i \quad (12.28)$$

$$\Leftrightarrow -w_1 \left(\sum (x_i)^2 - N \bar{x}^2 \right) = - \sum x_i y_i + \bar{x} \bar{y} N \quad (12.29)$$

$$\Leftrightarrow w_1 = \frac{\sum x_i y_i - \bar{x} \bar{y} N}{\sum (x_i)^2 - N \bar{x}^2} \quad (12.30)$$

where we used that $\sum x_i = N\bar{x}$ with N the number training data. These results for w_0 and w_1 can now be used to specify the hypothesis Eq. (12.10) that minimizes the MSE error for the particular training dataset $\mathcal{D}_{\text{train}}$. $h(x)$ can now be easily used with the fixed weights for making predictions (note that the weights don't show up as argument of h anymore!), e.g., for a new and arbitrary value x^* the corresponding output is $h(x^*) = w_1 x^* + w_0$.

We conclude this section by mentioning that it is possible to derive a general closed form solution which is valid for *all* linear regression problems—regardless of the number of features or the type of hypothesis. This requires a more complicated derivations and linear algebra which will be presented in Sect. 13.2. The numerical approach in the following section is an approach that is applicable not only for linear models but is a prerequisite for essentially all, more complex machine learning problems.

Things to Remember (Closed form solution for linear regression)



The trained model is given by the function $h(x) = w_1 x + w_0$ where the two weights that minimize the cost can be obtained as closed form solution:

$$w_1 = \frac{\sum_{i=1}^N x_i y_i - \bar{x} \bar{y} N}{\sum_{i=1}^N (x_i)^2 - N \bar{x}^2} \quad \text{and} \quad w_0 = -w_1 \bar{x} + \bar{y}.$$

There $\bar{x}$ and $\bar{y}$ indicate the mean values of all x and y values of the training dataset, and N is the number of training examples.

12.3.7 Cost Function Minimization: The Method of Gradient Descent

Here, we focus only on simple linear regression and one of the most simplistic numerical minimization methods. Finding the minimum of a function is a ubiquitous task in math, science, and engineering. Most of the time, however, it is not possible to derive a closed form solution, which is where numerical minimization strategies come into play. One of these numerical methods is the gradient descent method. Even though it is a method that does not perform sufficiently well for any real problem, it can be well used to explain the general methodology and how numerical minimization strategies can be used in the ML context.

To keep the notation as accessible as possible, the gradient descent method is not introduced in a general n -dimensional space but rather already formulated for the specific regression problem, i.e., in terms of the weights and the cost function. In the subsequent subsection, we then apply the method to the problem of linear regression with one feature.

Problem Formulation

Let us start with the general problem formulation: the problem at hand is a *minimization* problem where we seek to find the minimum of a function.⁵ There is a whole mathematical branch dedicated to such minimization problems, where often the type of problem is further restricted to *convex function minimization*. As an important property, this implies that the function has no more than one minimum and no maximum, which is quite convenient for mathematical reasons. A quadratic function (as shown in Fig. 12.10) is convex; the function shown in Fig. 12.11 on the left is non-convex. A key element in mathematical optimization is the so-called *cost function* or *objective function*, shown in the left panel of Fig. 12.11, written as $\mathcal{J}(w_0, w_1)$. This is the function to be minimized which represents the deviation between a desired property and an actual property. For our problem this is the sum of the (squared) distances of the training data from the line for a given set of weights. The objective is to find the optimal set of weights such that this residual error is minimized. If the mean loss were zero, we would have reached a perfect fit. However, in reality the error usually cannot become zero, and we are satisfied if it reaches some minimum value. Last but not least, most problems are non-convex, and therefore the method of steepest descent is only able to find a *local minimum* and not the *global minimum* (i.e., the overall minimum of the whole considered domain). There exist other methods that are at least partially able to resolve this problem that will be introduced when they are required in the next chapters.

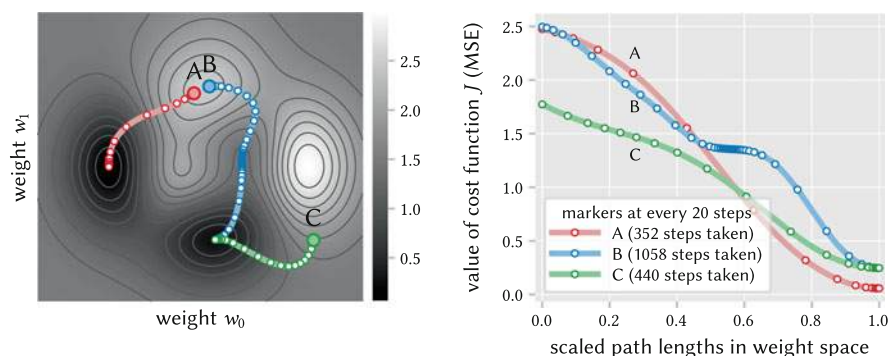


Fig. 12.11 Error as a function of the weight space (left) with trajectories of the steepest descent method. The different initial positions are the large markers labeled “A”, “B”, and “C”. The minimization procedures result in weights that locally minimize the cost function, but only one of them ends up in the global minimum. The right panel gives an idea about the “speed” with which the error landscape is traversed. It also compares the MSE magnitude reached at the end of the iterations. As step size a value of $\eta = 0.01$ was used. The iterations were stopped when the change between two subsequent MSE values was below 10^{-7} . The small round markers are plotted every 20 steps

⁵ In case that we are looking for a maximum, we still can use the whole “minimization machinery,” and only the sign of the cost function needs to be changed.

Motivation of the Gradient Descent Method

Taking a look at a hypothetical two-dimensional error landscape $\mathcal{J}(w_1, w_2)$ in the left panel of Fig. 12.11 we already can intuitively develop a straightforward idea for finding the minimum: starting from some point in the weight space, we move always perpendicular to the contour lines of equal error values, i.e., into the direction that most effectively reduces the error—in other words, *the negative gradient of the cost function*, $\nabla \mathcal{J}(w_0, w_1; \mathcal{D}^{\text{train}})$. A “refresher” and a brief introduction into calculus including the gradient operator ∇ can be found in Appendix A.1).

This approach is rather appealing as it can be comprehended intuitively. For example, an often used hypothetical situation used as explanation is the following: Assume that you are hiking the mountains and that you got lost. All you know is that the cabin is at the bottom of the valley. As it has also gotten dark you can’t see very far. But you can see (or rather feel) how steep the soil under your feet is. If you’re always going into the direction downhill of the steepest slope, then eventually you will find a point from which you can’t get further down and you hopefully found the cabin.

Algorithmic Formulation and Numerical Aspects

The method of *gradient descent* (or: *steepest descent*) can be stated more formally as shown in Algorithm 12.1.

During the initialization in 2. a start position in weight space needs to be chosen. Furthermore, because gradient descent is an iterative numerical method, a few numerical parameter need to be defined: (i) a parameter for checking for convergence, TOLERANCE; (ii) a parameter for deciding how many steps we would be willing to try at maximum before we entirely give up, MAX_STEPS; and (iii) the learning rate η , i.e., a factor that scales the size of the steps taken.

Finding a new position in weight space is done incrementally based on updating the old position by an increment (in 4.). For obtaining the increment (in 3.), the gradient of the cost function w.r.t. the weights is required (cf. Eq. (12.19) and Eq. (12.20)) which gives the direction of the steepest slope. This, however, is the direction of the steepest *ascend*. Thus, in 4. the “-” sign is required to get the direction of the steepest *descent*. Additionally, the scaling factor $\eta > 0$ which for ML problems is called *learning rate* determines for how far we proceed along this direction. The update in 4. will be done as long as there is a significant change in the weights during the update steps or until the maximum number of allowed steps are reached. How big a *significant change* is, is determined by the constant TOLERANCE. Here, this is an absolute value, e.g., a value of TOLERANCE= 10^{-7} could be reasonable. What exactly a “reasonable value” is depends on the problem, but as a rule of thumb at least the lower bound is given by the *machine epsilon*, which is roughly speaking the value difference that can just be represented by the particular number representation in the computer. For double precision numbers (i.e., a C++ double or a Python float), this is $\approx 10^{-16}$. In practical situations this value is often too small and may result in a huge number of iteration steps. For further details on numerical computations, the reader is referred to the vast body of literature, e.g.,

Algorithm 12.1: The Method of Steepest Descent for Simple Linear Regression

1. **Given are N training data records:** $\mathcal{D}^{\text{train}} = \left\{ \left(x_i^{\text{train}}, y_i^{\text{train}} \right) \right\}_{i=1 \dots N}$

2. **Initialization of position in weight space and numerical parameters:**

- choose start values for both weights $\mathbf{w}^{(0)} = [w_0^{(0)}, w_1^{(0)}]^T$
- choose the step size parameter ("learning rate") η
- choose tolerance for convergence test TOLERANCE
- choose maximum number of steps s MAX_STEPS

3. **Compute gradient at current position**

- The predicted value of y for any value of x is $y^{\text{pred}} = w_0 + w_1 x$
 - The gradient needs to be computed with respect to the weights w_0 and w_1
- where $\nabla_{\mathbf{w}} \mathcal{J} = \left[\frac{\partial \mathcal{J}}{\partial w_0}, \frac{\partial \mathcal{J}}{\partial w_1} \right]^T$

$$\frac{\partial \mathcal{J}}{\partial w_0} = -\frac{2}{N} \sum_i \left(y_i^{\text{train}} - y_i^{\text{pred}} \right)$$

$$\frac{\partial \mathcal{J}}{\partial w_1} = -\frac{2}{N} \sum_i \left(y_i^{\text{train}} - y_i^{\text{pred}} \right) x_i^{\text{train}}$$

4. **Update the weights** for the new step $(s + 1)$

$$\mathbf{w}^{(s+1)} \leftarrow \mathbf{w}^{(s)} + \eta \left(-\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}^{(s)}) \right)$$

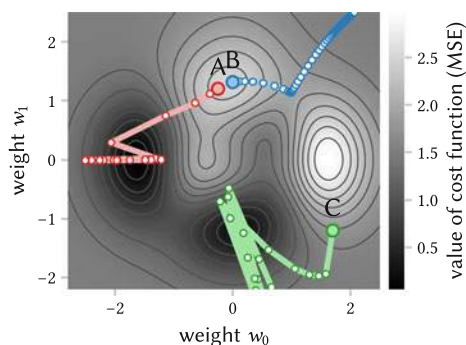
5. **Check convergence**

- **IF** $\|\mathbf{w}^{(s+1)} - \mathbf{w}^{(s)}\| < \text{TOLERANCE}$
THEN: convergence reached $\rightarrow$ EXIT
- **IF** number of steps $i > \text{MAX\_STEPS}$
THEN: no convergence reached $\rightarrow$ EXIT
ELSE: new iteration step, GOTO 4.

the book on numerical recipes and scientific computing by Press et al. [7] is highly recommended. As a last aspect concerning the convergence behavior, we note that monitoring the cost function can sometimes be a good alternative for monitoring all weights individually. In this case the convergence test is obviously done only for the value of the cost function.

Determining a suitable value for the learning rate can be tricky: on the one hand we seek an as-large-as-possible learning rate as this implies a smaller number of

Fig. 12.12 The steepest descent method with too large step sizes ($\eta = 0.5$ for the leftmost starting point “A”, $\eta = 0.25$ for “B”, and $\eta = 0.9$ for point “C”). None of the trajectories converges to one of the two minima. Each of the small round markers indicates one step



steps and thereby a faster learning. In particular when the calculation of the gradient is computationally complicated and therefore takes a lot of time, this becomes relevant. On the other hand, the larger the learning rate, the larger also the errors can become because the step taken might be too large to capture changes in the error landscape. This can be seen in Fig. 12.12 where we start with the same initial positions but this time with a much larger learning rate. Trajectory A seems to get close to the minimum but then is trapped in an infinite “back-and-forth” between the left of the minimum and right of it. This is also a typical situation where the maximum number of steps are required as termination criterion; without this, our code would never stop. Trajectory B suffers from accumulating errors and, as a consequence, takes a wrong turn left and moves forever into the wrong direction, again, a case for the maximum number of iterations. The (attempted) minimization C is similar to A but keeps on jumping around wildly.

Mathematically speaking, the reason for large, accumulating errors is the fact that the chosen update scheme for the weights is an *explicit* scheme: the gradient is calculated based on the information of step s and is then used to predict the new position in the weight space, i.e., at step $s+1$. However, the gradient at this new point can have a very different value than the one used to calculate the new position. The discrepancy between the gradients at the original and the updated position gives rise to an error; the steeper the gradient, the larger the error. Ultimately, the learning rate needs to be chosen according to the region with the steepest gradient. This results in unnecessarily small steps in all other regions (e.g., compare the middle section of line B in Fig. 12.11) which makes the method computationally inefficient and which is one of the reasons why the gradient descent method is not used in more complex application cases.

If the cost function to be minimized is a convex function, then the choice of any start point for the steepest descent iterations works and eventually results in finding the global minimum. However, if the error landscape is non-convex as the one shown in Fig. 12.12, then the choice of reasonable initial values for the weights is crucial: the algorithm might find two very different minima even if the two initial points (A and B) are very close by.

There are a number of numerical aspects that we did not touch upon here, e.g., adaptive determination of the learning rate or the problem that the computation of the gradient can be expensive if the number of training data is large. Such more advanced topics are considered in later chapters.

12.4 Computational Aspects of Vectorization

As a preparation for the complete numerical implementation in the next section, we still need to briefly explain some computational aspects of the *vectorized implementation*. There we have to differentiate between two different approaches.

12.4.1 The First Type of Vectorization: Partial Vectorization

In the following Python implementation, we use a technique called *vectorization* where instead of writing `for` loops over vector/array elements, a mathematical operation is performed simultaneously for *all* elements. For example, the vectorized sum of two arrays `a` and `b` can be simply written as

```
In [1]: import numpy as np
a = numpy.array([1, 2, 3])
b = numpy.array([4, 5, 6])
c = a + b
```

```
In [2]: c
```

```
Out [2]: array([5, 7, 9])
```

and we don't have to iterate over all elements of the matrices `a` and `b`:

```
In [3]: a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Create "uninitialized" array c that has the same shape as a
c = np.empty_like(a)

# Loop over all 3 elements and define the elements of c
for i in [0, 1, 2]: # alternative: `for i in range(len(a)):`
    c[i] = a[i] + b[i]
```

```
In [4]: c
```

```
Out [4]: array([5, 7, 9])
```

Obviously, the first and vectorized version is much shorter. Additionally, the vectorization by writing `c = a + b` creates code that looks very similar to the mathematical formulation which makes it intuitively accessible and also easier to

debug. Furthermore—and this is an at least equally important reason for using vectorization—this code brings a significant speed up for the execution of the program as compared to the use of `for` loops. For time-intensive computations, a factor in between 2 and even more than 100 can be reached without having to bother with any parallelization of the code. The computational time becomes relevant for many ML problems, and in particular, if there are many data records involved, if the functions to be evaluated are getting more complex, or if the algorithms need many iterations. Python (or more specifically `numpy`) as well as MATLAB® both offer vectorization as part of their language. This, however, often requires that some parts of the code needs to be adapted (e.g., `for` loops over individual elements are to be removed). The next subsection shows the derivation and Python implementation for partial vectorization of the simple linear regression example that is still quite fairly easy to understand.

As an example, consider the computation of the predicted label $y^{\text{pred}}_i = w_0 + w_1 x_i$ for all records i which we would implement as

```
In [1]: import numpy as np
x = np.array([1.1, 1.5, 2.4, 2.5, 3.0])
w0, w1 = -0.1, 2.1
y_pred = w0 + w1 * x
```

There, the equation is evaluated for *all* entries of `x` at the same time such that `y_pred` has as many entries as `x`.

12.4.2 A Second Type of Vectorization: Full Vectorization

There is, however, another approach that results in what we call a *full vectorization*: rewriting the above equations exclusively in terms of linear algebra operations is another way of using vectorization that additionally can be generalized to problems with arbitrary numbers of features and weights. Now, the weights would be represented by a weights vector, $\mathbf{w} = [w_0 \ w_1]^T$, and we use a scalar product for each element i ,

$$y^{\text{pred}}_i = \underbrace{[1 \ x_i]}_{=: \tilde{\mathbf{x}}} \cdot \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \tilde{\mathbf{x}} \cdot \mathbf{w} . \quad (12.31)$$

The implementation needs to be such that again the scalar product is applied simultaneously to all records of the dataset which requires some more preparations as well as formulations that are different from the original mathematical formulations. For instance, the vector $\tilde{\mathbf{x}}$ would become a matrix with a first column of ones. As this requires some more careful work when formulating the expressions for computing the derivatives of the loss (is the matrix transposed or not? are the weights multiplied from the left or the right? etc.) and might not be of interest to all readers, the derivation and implementation is shown in the separate section, Sect. 13.2.

12.5 A Worked Example of Simple Linear Regression

We will now return to the problem of univariate linear regression and step by step introduce how Algorithm 12.1 can be transformed into Python code. Different types of visualizations are shown and the results from this regression analysis are then discussed and interpreted such that this section can be used as a “template” for further studies.

12.5.1 Python Implementation

For computing the gradient we need the two individual derivatives of the cost function w.r.t. the two weights. Luckily, for the problem under consideration, this can easily be done, and in fact we already computed them above in Eqs. (12.19) and (12.20), so that we only summarize them here

$$\frac{\partial \mathcal{J}(w_0, w_1; \mathcal{D}_{\text{train}})}{\partial w_0} = -\frac{2}{N} \sum_{i=1}^N (y_i - y^{\text{pred}}_i) \quad (12.32)$$

$$\frac{\partial \mathcal{J}(w_0, w_1; \mathcal{D}_{\text{train}})}{\partial w_1} = -\frac{2}{N} \sum_{i=1}^N ((y_i - y^{\text{pred}}_i) x_i) . \quad (12.33)$$

where y_i is the target value of the i -th training data record and $y^{\text{pred}}_i = w_1 x_i + w_0$ is the predicted value for the two weights. We can now easily transfer the above algorithm into Python code. Quite unsurprisingly, we start by importing the numpy package. Then the training data is defined by randomly sampling the `x_train` values from within an interval given by `xlim`. The array `y_train` is obtained as a linear function, superimposed with some noise (through the variable `eps_noise`).

```
In [1]: import numpy as np
        rng = np.random.default_rng()
        xlim = (-2., 3.5) # min. and max. x-values
        n_total_data = 50 # number of data points

        X_train = xlim[0] + rng.random(size=n_total_data) * (xlim[1] - xlim[0])
        eps_noise = 0.5 * rng.standard_normal(n_total_data)
        y_train = -0.7 + 0.5 * X_train + eps_noise
```

This is one of the simplest way of creating a training dataset. As a next step, we can now define a function that returns the two components of the gradient of the cost function, Eqs. (12.32) and (12.33):

```
In [2]: def gradient_of_cost(weights, X, y):
        y_pred = weights[0] + weights[1] * X
        N = y_pred.size
        residual = y - y_pred
        dL_dw0 = -(2 / N) * np.sum(residual)
        dL_dw1 = -(2 / N) * np.sum(residual * X)
        return np.array([dL_dw0, dL_dw1])
```

There, the `numpy.sum` returns the sum all elements of the given one-dimensional array. We also could have simply implemented $(1/N) \sum(\dots)$ as `numpy.mean(...)` but wanted to stay as close to the mathematical formulation as possible. The statement `residual * X` is an element-wise multiplication of all entries of the two variables. The above function then returns the two components of $\nabla \mathcal{J}$ as a numpy array. Similarly, we also implement a function for computing the MSE:

```
In [3]: def cost(weights, X, y):
        y_pred = weights[0] + weights[1] * X
        N = y_pred.size
        mse = (1 / N) * np.sum((y - y_pred) ** 2)
        return mse
```

This, however, is only used for showing the final results. As a last preparation we now have to define a few numerical parameter and the initial guess for the values of the two weight:

```
In [4]: maximum_number_of_steps = 1000
        step_size = 0.05
        tolerance = 1e-4
        weights = np.array([2.2, 1.5])
```

Finding a good `step_size` often requires a bit of trial-and-error and also depends on the magnitude of the involved values; if all values are roughly in the range of unity then a reasonable value for the “stopping criterion” (the `tolerance`) is 10^{-7} but larger values might work as well (as is here the case). Taking a look at the convergence behavior of the weights or the value of the cost function helps to determine this parameter and is always recommended. We can now perform the update scheme where a maximum of `maximum_number_of_steps` update steps are allowed in the `for` loop:

```
In [5]: for step in range(maximum_number_of_steps):
        delta_w = step_size * (-gradient_of_cost(weights, X_train, y_train))
        weights += delta_w # same as: weights = weights + delta_w

        if np.linalg.norm(delta_w) < tolerance:
            break
    else:
        print("ERROR: maximum number of iterations exceeded")
```

Inside the `for` loop, we first compute the weight increment (recall that `delta_w` is an array with two values) and update the weights to their new values. Then we check if the length of the weight increment in weight space is smaller than the tolerance (`np.linalg.norm` is just the Euclidean norm). In case the weight increment is negligible, convergence has been reached, and the `break` statement exits the `for` loop, skipping the `else` statement. If the `for` loop is finished without encountering the `break` statement (in other words, convergence has not yet been reached), it will continue with the `else` clause and print the error message. Last but not least, we summarize some of the most important results:

```
In [6]: print(f'w0 = {weights[0]:.3f}, w1 = {weights[1]:.3f}')
        print(f'{step + 1} steps taken')
        print(f'final cost J: {cost(weights, X_train, y_train):.4f}')

w0 = -0.857, w1 = 0.512
99 steps taken
final cost J: 0.1794
```

We have now implemented a full training procedure for the regression problem that even could be relatively easily generalized to more than one features and multiple outputs. Next, we will now analyze the results and the performance of the training process itself.

12.5.2 Visualization and Discussion of Training Results

Often, one of the most accessible approaches toward analyzing the results is visualization of some aspects of the data. Obviously, a whole zoo of different visualization types exists. All of them can be useful, depending on the context and goals—this is our choice and requires a bit of experience.

One of the interesting investigations is to observe the evolution of chosen quantities during the training procedure. This is helpful, e.g., for understanding the convergence behavior of the training process. Two of such plots are shown in Fig. 12.13. The first plot shows the training data together with regression lines at different “stages” of the training process. The respective weight values are shown in the legend, and the colors of the lines corresponds to different stages (light = initial weights, darkest = final weights). Visually, we can already see that the darkest line fits well to the training data. This is an important verification and helps to see if the Python implementation yields reasonable results at all. For producing this plot, for each training step, we record the current values of the two weights, e.g., by

```
In [1]: all_weights = []
        for step in range(maximum_number_of_steps):
            # ...
            all_weights.append(weights)

        all_weights = np.array(all_weights)
```

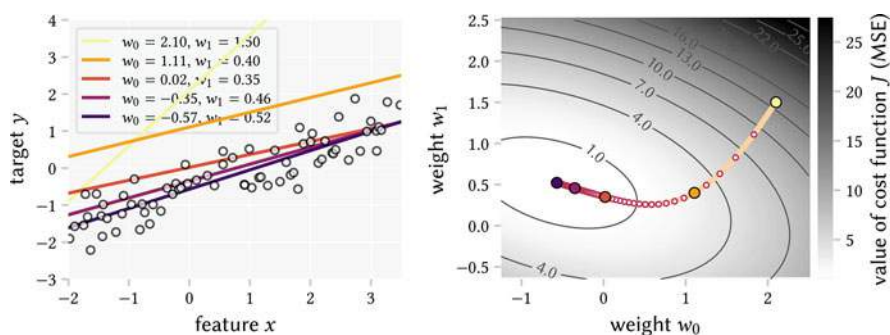



Fig. 12.13 Training data and lines for a few chosen weights from the training process (left panel). The error distribution (“loss landscape”) is shown in the right panel, and the light-red line is a “trajectory” for the chosen initial weights where each small red marker represents one training step. The colors of the markers correspond to the line colors in the left panel

where we used a Python `list` for appending new weights, resulting in a list of arrays. In the end, this is then converted to a `numpy` array. Using a list is faster and more convenient to type than appending data to a `numpy` array (but this is also a matter of personal preference). Then, the values of the first weight for all steps could then be accessed by `all_weights[:, 0]`. Also the cost value of all iteration steps could be stored in this way. To produce a plot of regression lines, we can plot a line using $y = w_0 + w_1x$ for each weight pair and some given x values.

The second type of plot is a visualization of the weights space on top of a contour plot of the cost values as a function of the two weights, $\mathcal{L}(w_0, w_1)$.

Words of advice



The type of plot showing the cost $\mathcal{J}$ as a function of two weights is often referred to as the *loss landscape*, in particular in the deep learning community. Strictly speaking, this not consistent with the terminology used here in statistical ML. The *loss* $\mathcal{L}$ is related to a value for each *individual* data record, while the shown *cost* is a resulting value obtained from *all* data records.

The cost values are shown in different gray values. To create this plot, one has to evaluate the cost function for each “pixel” of the image, i.e., by creating a two-dimensional array of cost values. This is superimposed with the evolution of the current weights as obtained for the previous plot. It is shown as the light-colored line. Each of the tiny marker indicates one of the individual update step; the five larger, colored markers correspond to the colored lines in the previous plot. We observe that during the minimization steps, changes in the weight space are initially larger and are decreasing as the minimum is approached. The reason for this is that the update increment is scaled with the gradient value. Additionally, we can nicely see that the trajectory seems to be perpendicular to the iso-lines of the weight space. This is not surprising since the direction of the update is determined by *gradient*

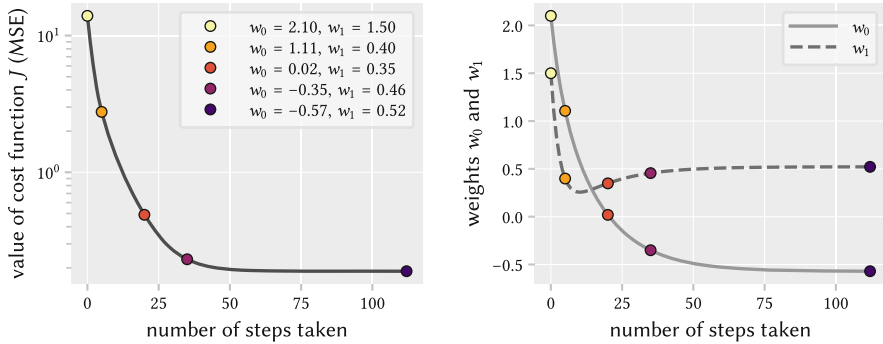


Fig. 12.14 Decrease of the cost (training MSE) during the training plotted with a logarithmic y-axis (left panel). Variation of the two weights during the training (right panel)

of the cost function. However, if for the learning rate we would have chosen an excessively large value, then this behavior would change and errors become larger. Thus, this type of plot also helps to judge the appropriateness of the learning rate.

While this already gives a good idea about the convergence of the training process, we can do more to see how some of the important variables and parameters change during the training iteration: plotting the cost $\mathcal{J}$ versus the step number in the left panel of Fig. 12.14 shows that the cost decreases very quickly from about 15 to ≈ 1 within the first 40 steps. In the figure we used a logarithmic scaling of the cost axis which reveals more details within the value range of small cost values.

The other plot in Fig. 12.14 reveals how the values of the two weights change as a function of the number of training steps. After 40 steps, w_1 nearly reached its converged value, while w_0 does not converge as quickly. However, this behavior strongly depends on the chosen initial values of the weights and on the two-dimensional structure of the cost function. Again, the colors of the five markers are chosen to be consistent with the colors in all panels of Figs. 12.13 and 12.14.

This should give already quite a broad picture of the behavior of the system. Nonetheless, there even is more that one could do, e.g., we could create a line plot showing how the required number of training steps to reach a certain accuracy depends on the chosen step size, the true function could be shown (cf. Fig. 12.15) and compared to the regression line, the “loss landscape” could be plotted as a surface in three dimensions for (possibly) better visualizing the shape, we can compare the results to the analytical solution from the previous section, or we could investigate how the convergence behavior depends on the initial guess. We encourage the reader to try out such analysis and visualization as an exercise.

12.5.3 Validation of the Trained Model

There is still one important part missing which is also crucial within the concept of machine learning: the validation of the trained model based on new data that has not been used for the training. For this, the initial dataset was split into a training and a testing dataset, e.g., using the strategy explained above in Sect. 12.2.10 and there in particular in Python Listing 12.1. The training and testing data along with the corresponding cost as a function of iteration steps is shown in Fig. 12.15. Note that the weights are adjusted only during the training. Evaluating the cost function with the testing data $\mathcal{D}^{\text{test}}$ and with the values of the weights $w_0^{(i)}$ and $w_1^{(i)}$ at iteration step i gives the testing MSE

$$\text{testing MSE}(i) \equiv \mathcal{J}(w_0^{(i)}, w_1^{(i)} | \mathcal{D}^{\text{test}}) \quad (12.34)$$

which is plotted in the right panel of the figure. In this case, the obtained testing MSE is always larger than the training MSE and after ≈ 40 steps doesn't change much even though the weights are still changing slightly. It is even possible that the value of the testing MSE is below the training MSE. This depends on how statistically representative (or comparable) the training and the testing datasets are. This is clearly an important aspect in ML as it has a direct impact on how reliable the training is. Luckily, for investigating this aspect of training and testing data, there are methods available, called *cross validation* which will be introduced in Sect. 16.2.

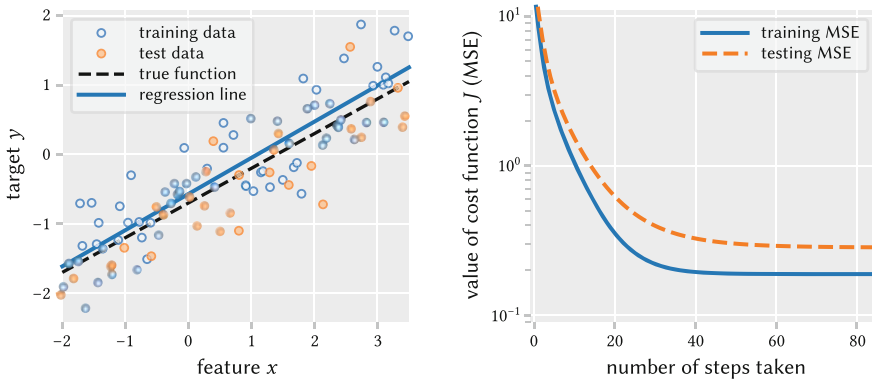


Fig. 12.15 Training and testing data (left) and the training and testing MSE during training. The regression line uses the weights identified during the gradient descent and is based on the training data only

12.6 Multiple Linear Regression Models

We have already made a big step forward toward understanding [ML](#) regression: many important aspects that build the foundation for generalizing simple linear regression to more complex [ML](#) models have been introduced in the previous sections. This section considers *multiple linear regression models* where the “multiple” refers to the fact that such models are able to consider multiple features as inputs. The following section starts by formulating the hypothesis for such a case.

12.6.1 Hypothesis Formulation

A generalization of the simple regression model is a model that can handle more than one input variable, e.g., the features $X_1, \dots, X_n$. Such a model is called *multiple linear regression model*, for which the hypothesis space consists of hypotheses of the following form:

$$h(X_1, \dots, X_n; w_0, \dots, w_n) = w_0 + w_1 X_1 + \dots + w_n X_n . \quad (12.35)$$

The algorithm for the gradient descent method for multiple linear regression is a straightforward generalization of the one shown in Algorithm 12.1. It is mostly identical to that of simple linear regression except for the number of weights and the resulting number of derivatives $\nabla_{\mathbf{w}} \mathcal{J}$ as will be shown in the subsequent two sections.

12.6.2 Special Case: Multiple Linear Regression with Two Features

The case of two features X_1 and X_2 , implies that three weights are required: w_0 , w_1 , and w_2 . The derivation for the gradient of the loss function can be done in analogy to Eq. (12.19) for the derivative w.r.t. w_0 and in analogy to Eq. (12.20) for the derivatives w.r.t. w_1 and w_2 , so that we only summarize the most important steps in the following. For two features, the derivatives of the cost function read:

$$\frac{\partial}{\partial w_0} \mathcal{J}(w_0, w_1, w_2; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^N (y_i - y^{\text{pred}}_i) \quad (12.36)$$

$$\frac{\partial}{\partial w_1} \mathcal{J}(w_0, w_1, w_2; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^N ((y_i - y^{\text{pred}}_i) x_{i1}) \quad (12.37)$$

$$\frac{\partial}{\partial w_2} \mathcal{J}(w_0, w_1, w_2; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^N ((y_i - y^{\text{pred}}_i) x_{i2}) . \quad (12.38)$$

There, x_{i1} is the i -th record of the first feature, X_1 , and x_{i2} is the i -th record of the second feature, X_2 (see the introduction to the notation for datasets in Sect. 3.2.4). We assume that each of the input features and output data are one-dimensional numpy arrays, which could look like this (note that only the first few elements are shown):

```
In [1]: import numpy as np

X1 = np.array([-1.889, -1.842, -1.741, -1.66, -1.593, -1.527, -1.451, ...])
X2 = np.array([ ...])
y = np.array([-1.555, -1.994, -1.574, -2.310, -1.295, -2.177, -1.523, ...])
```

Then, the gradient of the lost function can be computed using the following Python function:

```
In [2]: def gradient_of_cost(weights, X1, X2, y):
        y_pred = weights[0] + weights[1] * X1 + weights[2] * X2
        residual = y.ravel() - y_pred
        dL_dw0 = -2 * np.mean(residual)
        dL_dw1 = -2 * np.mean(residual * X1)
        dL_dw2 = -2 * np.mean(residual * X2)
        return np.array([dL_dw0, dL_dw1, dL_dw2])
```

With similar definitions of the numerical parameters as before, the gradient descent iteration also stays unchanged except for the changed function arguments of `gradient_of_cost`:

```
In [3]: maximum_number_of_steps = 100000
        step_size = 0.02
        tolerance = 1e-4
        weights = np.array([0.05, 0.2, 0.])

        for step in range(maximum_number_of_steps):
            dL_dw = gradient_of_cost(weights, X1, X2, y)
            delta_w = step_size * (-dL_dw)
            weights += delta_w

            if np.linalg.norm(delta_w) < tolerance:
                break
        else:
            print("ERROR: maximum number of iterations exceeded")
```

Figure 12.16 shows an example with two inputs and one target variable which results in the shown *regression plane*. For more features the plane becomes a hyper plane and cannot be plotted anymore.

Similar to simple linear regression, the important next step also for multiple linear regression would be to investigate how the model with respect to testing data performance. This is done in full analogy to what already was shown in Sect. 12.5.3 and will therefore be left as an exercise.

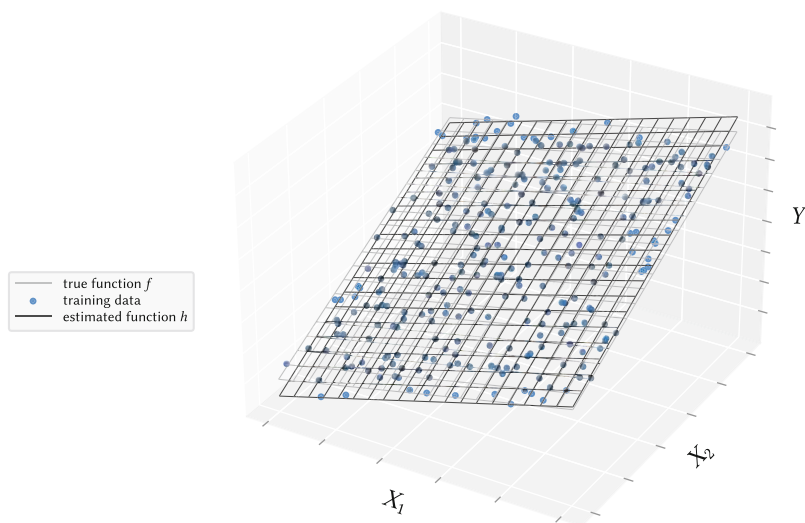


Fig. 12.16 Multiple linear regression for two features identifies the plane that fits best to the data. Both the true function that was used to create the dataset and the fitted plane are shown

12.6.3 Generalization to an Arbitrary Number of Features

A generalization to an arbitrary number of features is not too complicated: for k features, the derivatives of the loss function can be calculated in analogy to the derivative w.r.t. w_1 or w_2 from above, i.e., for $\partial \mathcal{J} / \partial w_k$ the only difference to Eq. (12.38) is a variable x_{ik} instead of x_{i2} at the end of the summation. In that case it is useful to rather operate with the feature matrix $\mathbf{X}$ which contains the features in columns. This approach is additionally also computationally favorable but requires some more linear algebra thinking. We will derive and demonstrate this in Sect. 13.2.

Algorithm 12.2: Steepest Descent for General Linear Regression



1. **Given are N training data records:**

$$\mathcal{D}^{\text{train}} = \left\{ \left(x_{i1}^{\text{train}}, \dots, x_{in}^{\text{train}}, y_i^{\text{train}} \right) \right\}_{i=1 \dots N}$$

which consists of the n features $X_1, \dots, X_n$.

2. Initialization of position and parameters:

- choose start values for all weights $\mathbf{w}^{(0)} = [w_0^{(0)}, \dots, w_n^{(0)}]^T$
- choose the step size parameter ("learning rate") η
- choose tolerance for convergence test TOLERANCE
- choose maximum number of steps MAX_STEPS

3. Compute gradient at current position

... with respect to the weights w_i ,

either analytically or numerically

$$\nabla_{\mathbf{w}} \mathcal{J} = \left[\frac{\partial \mathcal{J}}{\partial w_0}, \dots, \frac{\partial \mathcal{J}}{\partial w_n} \right]^T$$

4. Update the weights

$$\mathbf{w}^{(i+1)} \leftarrow \mathbf{w}^{(i)} + \eta \left(-\nabla_{\mathbf{w}} \mathcal{J}(\mathbf{w}^{(i)}) \right)$$

5. Check convergence

- **IF** $\|\mathbf{w}^{(i+1)} - \mathbf{w}^{(i)}\| < \text{TOLERANCE}$
THEN: convergence reached $\rightarrow$ EXIT
- **IF** number of steps $i > \text{MAX\_STEPS}$
THEN: no convergence reached $\rightarrow$ EXIT
ELSE: new iteration step, GOTO 4.

The general steepest descent algorithm is shown in Algorithm 12.2.

12.7 Exercises

12.1

- How is the "training error" of a trained, linear regression model computed?
- How is the "testing error" of a trained, linear regression model computed?
- What is a machine learning model?

12.2

- Consider the train-test split as shown in Python Listing 12.1. Why is the line with `rng.shuffle(...)` needed? Show an example for the variable `indices` and after shuffling.
- Create an example dataset and plot the training and testing data in two different colors. Compare plot of the data *with* and *without* shuffling.
- Try a few different values as seed for the random number generator until you find an "extreme" case where, e.g., many points of the training or the testing data are close to each other. What does that imply for training and for prediction?

Hint: the dataset should not be too big, e.g., 20-30 points are a good value.

12.3 Given are the support points of a polynomial

$$\{(1, 3.5), (3, 5.1), (4, 3.5), (7, 8)\}.$$

- (a) Take a look at the documentation of numpy's `Polynomial` class. How can you access the fitted coefficients of the polynomial?

Even though `Polynomial` is not a plain function but in fact a class the usage is still quite intuitive and does not require deeper knowledge about classes and object oriented programming.

- (b) Use numpy's `Polynomial` to fit a polynomial with degree of 3 to the data. Which are the interpolated values y_{interp} for the x values of $x_{\text{interp}} = (2, 2.5, 3, 3.5, 4)$? Write down the analytical function of the polynomial including the determined parameters.
- (c) Show the support points and the resulting polynomials of degrees $0 \dots 3$ in a plot. Why does only the polynomial of degree 3 pass through all support points? And how are the polynomials with the other degrees computed (hint: take another look at the `numpy` documentation).

Optional information As `Polynomial` is a class, it therefore may contain, e.g., variables or even functions “under the hood.” Take a look at how the function `fit` is called (it is `Polynomial.fit(...)` and returns a new `Polynomial` object where the parameters are already defined). After this, we can use this object to do some further things, such as to use it for interpolation, to compute derivatives, or to find the *roots* (i.e., the points x_r for which it is $f(x_r) = 0$). `numpy` has a number of other interpolation methods all of which have the same structure. So, it's definitely worth learning a bit about how to use the `Polynomial` class!

12.4 As a continuation of Example 12.2, the given training data should now be used to determine the best fit of a line and thereby to determine the parameters w_0 and w_1 .

- (a) Manually compute the mean values $\bar{x}$ and $\bar{y}$ and the two sum-terms required for computing w_1 . Which is the resulting value of w_1 ?
- (b) Which is the value for w_0 ?
- (c) Implement the solution as Python program.
- (d) Plot the training data as points and predictions as line.
- (e) Compute the value of the training [MSE](#).
- (f) How does the MSE change if the slope is doubled or if the offset of the line is increased by 0.5? Plot the resulting lines.

12.5 The dataset and the numerical parameters used in the example Sect. 12.5 is such that the training works very well. (a) Investigate and compare the training behavior for two datasets where the first is taken from the example and a second dataset that results in a different training behavior (e.g., in terms of the MSE or

how w_0 and w_1 vary). (b) Vary the learning rate and discuss how this influence the training behavior.

12.6 Section 13.1.1 discusses polynomial regression and hyperparameter optimization. Here, the task is to use `numpy`'s function fitting functionality for investigating and reproducing the shown example and to compare it to the numerical solution using the steepest descent method.

12.7 Create a multilinear dataset that has two features and a single target variable. For this, randomly sample points that are located on a two-dimensional plane and superimpose them with random noise for the feature. Then, try to “re-identify” the plane using a multiple linear regression model. Compute the training and the testing MSE.

References

1. Numpy. URL <https://numpy.org/>.
2. Website dedicated to Galton's life and work, 2022. URL <https://galton.org/>.
3. H. Blockeel. *Hypothesis Space*, pages 511–513. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. DOI https://doi.org/10.1007/978-0-387-30164-8_373.
4. L. N. G. Filon, G. U. Yule, H. Westergaard, M. Greenwood, and K. Pearson. Speeches delivered at a dinner held in university college, london in honour of professor karl pearson 23 april 1934. URL <https://archive.org/details/filon-et-al-1934-speeches-delivered-at-a-dinner/>.
5. F.R.S. Karl Pearson, Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11): 559–572, 1901. DOI <https://doi.org/10.1080/14786440109462720>.
6. F. Galton. *Natural Inheritance*. New York: Macmillan and Company, 5th edition, 1894.
7. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007. ISBN 0521880688.

Advanced Methods and Topics of Regression 13

The purpose of computing is insight, not numbers.

Richard Hamming (1915–1998)

American mathematician

13.1 Nonlinear Model Behavior with Linear Regression

While fitting straight lines to data is certainly useful, the world's most interesting phenomena are nonlinear—which holds in particular for physics, materials science, and engineering. Sometimes, *linearization* (e.g., by using a Taylor expansion) is a way of reformulating and simplifying a physical problem such that we are only left with linear behavior, but this is clearly limited and can come with a substantial error.

Before we start to introduce *nonlinear* models, let us start by defining what a *linear function* is.

Definition 13.1 (Linear function) In a linear function all variables only occur as powers of 1; no “mixed” terms involving more than one variable are allowed. Mathematically, two conditions need to be satisfied for a function f to be a *linear function*:

- Additivity: $f(x + y) = f(x) + f(y)$
- Homogeneity: $f(\alpha x) = \alpha f(x)$,

where x and y are the variables and α is a constant. If any of these conditions is not fulfilled, then the function is a *nonlinear function*.

Already in Sect. 12.3.1 we briefly talked about the “linearity in linear regression,” but the previous section might have created the impression that linear regression nonetheless is just about straight lines or planes, at most. We will now introduce how linear regression can be used to learn and predict nonlinear behavior between input features and the output.

13.1.1 Polynomial Regression

When the dataset is the result of an underlying, nonlinear behavior, *polynomial regression* is one of the methods that is able to produce a nonlinear model. The hypothesis class in case of only one input variable is given by

$$h(x; w_0, \dots, w_k) = w_0 + w_1x + w_2x^2 + \dots + w_kx^k = \sum_{i=0}^k w_ix^i. \quad (13.1)$$

To see why this is considered a linear function in the sense of Definition 13.1, we realize that the attribute “linear” in the context of a *machine learning* (ML) model refers to the weights: the w_i are considered the variables, and the (powers of) x are considered the constants. And they are constant indeed, as x represents, the input data of a given dataset. Note that Eq. (13.1) contains the class of simple linear regression models as a special case for setting $n = 1$. The nonlinear behavior enters through the various powers of x , but it is a linear equations with respect to the coefficients w_i .

Other than that, solving a polynomial regression problem is not fundamentally different from what we have already done for the case of multiple linear regression. In fact, by introducing the new features in form of powers of the original feature,

$$X_1 \leftarrow X, \quad X_2 \leftarrow X^2, \quad \dots, \quad X_k \leftarrow X^k, \quad (13.2)$$

we obtain a multiple regression problem of the same structure as already introduced in Sect. 12.6:

$$h(X_1, \dots, X_k; w_0, \dots, w_k) = w_0 + w_1X_1 + w_2X_2 + \dots + w_kX_k. \quad (13.3)$$

Then, the training of this model follows the same steps as in the multiple regression problem, and no changes to the numerical implementation are required. Predictions are to be made usually using the original feature X and Eq. (13.1) or (less commonly) the transformed features X_i together with Eq. (13.3).

The degree of the polynomial, k , is a *hyperparameter* (cf. Sect. 11.3) that needs to be fixed—either based on knowledge about the underlying physical mechanisms (e.g., if we are investigating the motion of a body due to the gravitational force, then we know that the distance is a quadratic function of time and the polynomial regression model should at least contain the constant, linear, and quadratic terms),

or they can be found by performing a hyperparameter study, e.g., by investigating how the training and testing loss changes as demonstrated in the following example.

Hyperparameter Optimization for Polynomial Regression

Figure 13.1 shows polynomials of different degrees that were fitted to a given training dataset. To evaluate the predictive quality of the trained models, we also evaluate the fitted functions based on new data, the testing dataset. Training and testing data were both obtained by statistical sampling from a larger dataset. Plotting the polynomials that were fitted to the training dataset on top of the training and testing dataset shows a good fit of at least the polynomials of degree one and four to both datasets.

As a quantitative analysis, we also recorded the training and testing cost values which are obtained as the *mean squared error (MSE)*. We observe a typical behavior: initially, both losses are high and the model *underfits* (also see Sect. 12.2.8).

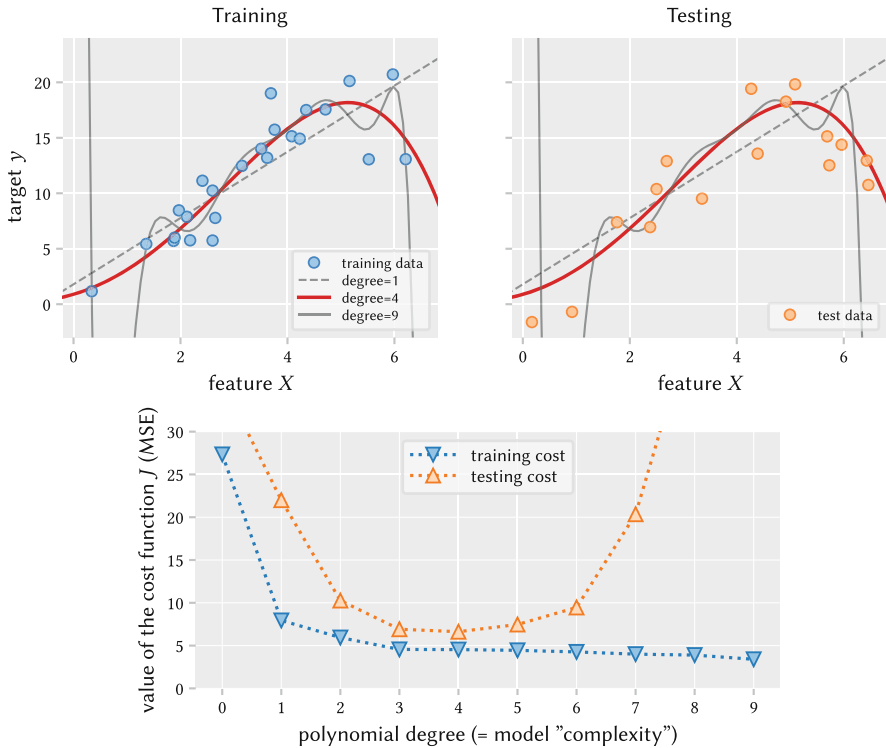


Fig. 13.1 Polynomial regression: the left panel in the top row shows the model functions of polynomials with different degrees trained on the shown training data. The right panel in the top row shows the same regression lines but now together with test data. The bottom panel shows the training and testing loss as a function of the polynomial “flexibility,” i.e., for polynomials of increasing degrees. The lines are only shown to guide the eye; data exists only at integer polynomial degrees

Then, both losses rapidly decrease, and the testing error reaches a minimum at a polynomial degree of 3 or 4 (the true function that was used to create the dataset had in fact a polynomial degree of 3). While the training loss decreases further with an increasingly complex model (i.e., polynomials with more terms), the testing loss already starts to increase. In other words, even though a more flexible model represents more and more features of the training dataset, it is strongly specialized toward the training dataset, and it does not generalize well to new data—the model *overfits*.

The Bias-Variance Tradeoff



The left region in Fig. 13.1 has high training *and* test errors; it is the region of *high bias* due to wrong model assumptions, resulting in underfitting. The region right of the minimum is dominated by a high sensitivity w.r.t. small fluctuations in the data, showing in a high testing cost and a low training cost. This error type is called *variance*. The *bias-variance tradeoff* denotes the property of a model to reduce the variance by increasing the bias and vice versa. The goal is to minimize both errors.

Hyperparameter optimization is a strategy for determining aspects of the hypothesis space $\mathcal{H}$, i.e., for finding the *most suitable* model parameters. Here, the objective would be to find the degree k of the polynomials that result in a trained model which also generalizes well in view of new data. While this can be done manually for models that contain one or two parameters, things will quickly become very complicated once more parameters need to be taken into consideration. Recall that the number of permutations with repetitions of k items with n choices is n^k (c.f. Sect. 5.1.6). For example, five parameters each of which can take, say, 40 different values, this results already in $40^5 = 102,400,000$ possible combinations which makes such a grid search nearly impossible, and smarter approaches should be chosen (e.g., based on a random sampling combined with an optimization strategy).

Limitations

Polynomial regression is usually restricted to polynomials with degree of less than approximately ten. The reason for this is that constructing higher-order polynomials, i.e., finding the respective weights, is often a numerically ill-conditioned problem. This means that in cases of high polynomial degrees, we cannot find the exact solution anymore—not even approximately. This difficulty is growing with a larger number of data records. Sometimes, feature scaling might help, e.g., by scaling all values to the range of $(-1, 1)$ or by subtracting the mean value of a feature. However, this will only slightly reduce problems in general. Using *orthogonal polynomials* helps to circumvent some of the problems (cf. Chapter 7 of [5] and references therein); however, often when fitting of polygons fails, it is advisable to rather use alternative methods, as, for example, those introduced below.

Polynomial regressions looks quite promising in the sense that we can simply increase the model complexity (i.e., the polynomial degree) to be able to represent very complex data. However, any additional polynomial degree effectively implies that an additional feature together with an additional parameter is added. This may result in insufficient data, as already discussed in the context of the *curse of dimensionality* in Sect. 2.4 which ultimately results in overfitting. Overfitting is a general problem with polynomial regression, even for one-dimensional datasets.

Polynomial regression is a *non-local method* which is a side effect of the construction method: changing the position of one data point can affect the whole curve. Trying to avoid this behavior leads to the method of *regression splines*, presented in Sect. 13.5.3.

Using the method of steepest descent for finding the best weights is generally not an advisable strategy as in particular for multiple features, and in the case of polynomial regression, the convergence behavior is very bad. As mentioned before in case of linear regression, there exists a closed form solution (derived in Chap. 13) that is much faster; however, this cannot be easily generalized to other classes of regression methods. Numerical minimization methods with a higher performance exist; however, we will not explore them in detail. Regardless which numerical method is used, the overall solution strategy for the ML problem always stays the same.

Things to Remember Linear Models for Polynomial Regression



- Linear models can even capture nonlinear behavior between input and output variables.
- A model is called *linear* because it is linear in the *weights* (and it doesn't matter how the features occur).
- Determining the degree of a polynomial can be difficult and is done either based on knowledge about the data generation process or based on a hyperparameter optimization.
- Usually polynomials with degrees of up to at maximum 6 or 7 are used; as an upper limit they should be below half the number of records.
- Too stiff polynomials can cause extreme over- or undershooting— visual inspection of the regression curves is always recommended (cf. the three polynomials in Fig. 13.1)!
- Don't use a model to “extrapolate” far beyond the range of the dataset.

13.1.2 Other Types of Hypothesis Functions

Everything that has been said about polynomial regression can also be easily transferred to other mathematical structures of the hypothesis class. Typically, the terms in which the feature occurs are linearly independent. Coming back to the example of polynomials: polynomials of up to a degree of n form the basis

for a vector space and are *linearly independent*, i.e., we cannot express either of the x^k as a linear combination of all other terms $\{x^i \mid i = 1 \dots n, i \neq k\}$. Linear independence might seem like a good criterion for formulating increasingly complex hypothesis functions. However, linear independence is not a requirement, and additionally it does *not* imply that these terms are not correlated. In fact, for polynomial regression, the terms may be highly correlated, e.g., if x is large then x^3 will be even larger, etc. Finding features that are not or only weakly correlated can be a criterion for the choice of a suitable hypothesis function, as we also will briefly discuss in the context of *multicollinearity* in Sect. 13.3.5.

Further Parametric Formulations

When mathematical functions are used as hypothesis functions, they are fully determined by a set of parameters, giving rise to *parametric models*. As a side effect, the choice of such functions can also be motivated by knowledge about underlying laws in the field of physics, materials science, and engineering, such that these model functions have some relation to physical aspects. However, in general, there is no restriction as to which function can be used. Here are a few mathematical functions that are often used in ML models:

- Trigonometric functions: For example, if the dataset is known to result from phenomena with periodicity in the time domain, e.g., $h(t) = \sin(w_0 + w_1 t) + \sin(w_2 + w_3 t)$;
- Exponential or logarithmic functions: e.g., $h(x) = w_0 + w_1 \log(w_2 x + w_3)$;
- Combinations of all the above functions, possibly also with polynomials, etc.

These hypothesis functions have the convenient feature that the trained model can be directly used for inference: the identified parameters of the hypothesis functions often have a specific physical meaning or we can at least directly explain aspects of the shapes of the functions. In general this is one of the biggest advantages of parametric formulations, as compared to non-parametric approaches.

13.1.3 Linearization of Nonlinear Hypotheses

We conclude this section by a short discussion of a well-known trick that is also useful for tackling situations where weights occur in nonlinear terms: transformation of the features and target variables. As an example, consider the prediction $y = h(x; w_0, w_1)$

$$y = w_0 \exp(w_1 x) \quad \text{with } w_0, w_1 \in \mathbb{R}. \quad (13.4)$$

This is an equation which is nonlinear in the parameter w_1 . It can be linearized by taking the logarithm on both sides

$$\underbrace{\ln(y)}_{y^*} = \underbrace{\ln(w_0)}_{w_0^*} + \underbrace{w_1}_{w_1^*} x \quad (13.5)$$

$$\Leftrightarrow y^* = w_0^* + w_1^* x . \quad (13.6)$$

This is now a linear equation in both weights. Subsequently, we can now use a simple linear regression model for obtaining the weights. One can also visually confirm that a prediction would be in fact linear. This might be trivial for this case; however, if a larger number of features are involved and the linearized equation is more complex, this still can be useful.

For making predictions or inference, we need to transform the weights back into the original representation. For this, the inverse function needs to be applied to the weights, which in this example is the exponential function. We obtain

$$w_0 \leftarrow \exp(w_0^*) \quad \text{and} \quad w_1 \leftarrow w_1^* . \quad (13.7)$$

Comparing this results that one would have obtained from nonlinear “curve fitting,” e.g., using a *least square* (LSQ) approach, we find that the obtained parameter are slightly different. The reason for this is that any coordinate transformation also changes the *metric* of the space, which also has an influence on how distances, because they are also affected by the transformation.

Another example is the nonlinear relation $y = w_0 x^{w_1}$ where for linearization a logarithm function needs to be applied which results in $\log(y) = w_0^* + w_1^* \log(x)$. After finding the two weights, they are back-transformed by $w_0 = 10^{w_0^*}$ and $w_1 = 10^{w_1^*}$.

We summarize the classical linearization approach in an algorithmic box.

Algorithm 13.1: Linearization of nonlinear hypothesis functions



1. Identify the function $g(\cdot)$ that is responsible for the nonlinear weight term(s).
2. Apply the inverse function $g^{-1}(\cdot)$ to both sides of the equation such that the nonlinearity is “neutralized.”
3. Perform a linear regression training with the linearized equation
4. Transform the obtained weights back by evaluating $g(\cdot)$ on them.

13.2 Generalized Formulations and Vectorization for Multiple Linear Regression

For linear regression problems, there is an alternative solution approach to the one introduced in the previous chapter. It is exclusively based on vector and matrix operations from linear algebra such as scalar products or matrix-vector multiplications. As an additional benefit, this method can easily be written in a

generalized form for including multiple features. It also lends itself to a very efficient, vectorized implementations in Python—both for a closed form solution and for numerical solution strategies.

The following derivation require vector and matrix operations which might be slightly more difficult to follow than the original formulation used in the previous chapter. To make the derivations more accessible, we sometimes use both index notation and compact vector-matrix notation.

13.2.1 Hypothesis and Extended Feature Matrix

We start by rewriting the mathematical expression of the hypothesis in a form so that it applies simultaneously to all m data records

$$h(X_1, \dots, X_n; w_0, \dots, w_n) = w_0 \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} + w_1 X_1 + \dots w_n X_n \quad (13.8)$$

where the X_i are a column vectors representing the i -the column of the feature matrix $\mathbf{X}$. The feature matrix has as many rows as there are data records. Recall that h will return a *scalar* value for each of the m records which is why h is not printed in a bold face (as opposed to a function that returns a vector valued quantity, e.g., a force vector).

The formulation in Eq.(13.8) appears, with small variations, in a number of different ML models. To simplify the derivations of the gradient of the loss, we prepend a column of “1”s to the feature matrix $\mathbf{X}$. For this *extended feature matrix*, we will use the blackboard character $\mathbb{X}$. Again, each row contains a data record. Explicitly writing the extended feature matrix gives

$$\mathbb{X} = \begin{bmatrix} | & | & & | \\ 1 & X_1 & \dots & X_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & \dots & x_{m,n} \end{bmatrix}. \quad (13.9)$$

Notation of the Extended Feature Matrix



By now the reader is probably well aware that there are many different quantities and variables in machine learning, in particular at the intersection between mathematical formulation and numerical implementation. To more easily memorize the meaning of $\mathbb{X}$ one can remember the extra column of ones and associate this with the extra diagonal line in the symbol of extended feature matrix, $\mathbb{X}$, as compared to the regular feature matrix $\mathbf{X}$.

(continued)

Sometimes one also encounters the notion *design matrix* which, however, sometimes refers to $\mathbf{X}$ and sometimes to $\mathbb{X}$. We avoid the use of that notion entirely in this book.

We can now rewrite the hypothesis in compact matrix-vector notation. For this, recall that we assume vectors to be column vectors such that in a matrix-vector multiplication the matrix is left of the vector, a *row matrix* (see Appendix A.1.2 for further details).

$$h(\mathbb{X}; \mathbf{w}) = \mathbb{X}\mathbf{w} = Y^{\text{pred}}, \quad (13.10)$$

where the product of the matrix with the vector again yields a vector which contains all records of the predicted variable Y . Insertion into the residual equation gives

$$R(\mathbf{w}; \mathcal{D}^{\text{train}}) = Y - Y^{\text{pred}} = Y - \mathbb{X}\mathbf{w}, \quad (13.11)$$

where $R(\mathbf{w}; \mathcal{D}^{\text{train}})$ is the column vector containing the residuals of *all* training records. Following the same argument as why the target variable Y is not written boldface, we also use a capital, non-boldface letter R for the residual. The residual for the record i is then written as r_i .

13.2.2 Derivation of Cost Function and Cost Gradient

The next and slightly more complex task is to formulate the computation of the training cost in terms of operations from linear algebra. To be as explicit as possible, we write the following equations both in index notation and in compact vector-matrix notation.

Taking the cost $\mathcal{J}$ as the mean of the squared residuals, we obtain in index and in compact vector notation

$$\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = \frac{1}{N} \sum_{i=1}^N r_i^2 = \frac{1}{N} R^{\top} \cdot R = \frac{\|R\|^2}{N}, \quad (13.12)$$

where the sum of all squared elements of R is simply the scalar product of R with itself¹ and $\|\bullet\|$ denotes the Euclidean norm of $\bullet$. Upon insertion of the residual term, we get in index and compact notation

¹ Recall that, $\mathcal{J}(w_0, \dots, w_n; \mathcal{D}^{\text{train}})$ would be pronounced as “*The resulting cost $\mathcal{J}$ for given training data is a function of the weights ...*”

$$\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = \frac{1}{N} \sum_i \left(y_i - \sum_n \mathbb{X}_{in} w_n \right) \left(y_i - \sum_m \mathbb{X}_{im} w_m \right) \quad (13.13)$$

$$= \frac{1}{N} (\mathbf{y} - \mathbb{X}\mathbf{w})^\top \cdot (\mathbf{y} - \mathbb{X}\mathbf{w}) . \quad (13.14)$$

For minimization of the loss with, e.g., the gradient descent method, the gradient of $\mathcal{J}$ with respect to the weights $\mathbf{w}$ is needed. This requires to calculate the derivatives of a scalar function (cf. Appendix A.1.9). We first realize that $\partial\mathcal{J}/\partial\mathbf{w}$ should again result in a column vector, containing the derivatives w.r.t. the different weights w_i . Starting from the index notation of $\mathcal{J}$ in Eq. (13.13) and applying the product rule, the important aspect is to consider about which indices the “contraction,” i.e., the summation about double indices, should be performed:

$$\frac{\partial\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}})}{\partial w_j} = -\frac{2}{N} \sum_i \left(y_i - \sum_k \mathbb{X}_{ik} w_k \right) \mathbb{X}_{ij} . \quad (13.15)$$

From there we can now switch to the compact vector-matrix notation

$$\frac{\partial\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}})}{\partial\mathbf{w}} = -\frac{2}{N} \mathbb{X}^\top (\mathbf{Y} - \mathbb{X}\mathbf{w}) = -\frac{2}{N} \mathbb{X}^\top \mathbf{R} \quad (13.16)$$

where the transposition of $\mathbb{X}$ is required because the index j should be the index of the items of the resulting column vector, $\partial\mathcal{J}/\partial w_j$.

Alternatively, we also could have started directly from Eq. (13.12). There, the steps are (i) to use the product rule, (ii) realizing that neither $\mathbf{Y}$ nor $\mathbb{X}$ depend on $\mathbf{w}$, (iii) that $\partial_{\mathbf{w}}\mathbf{w}$ is the identity matrix $\mathbf{I}$, as well as that (iv) the identity $(\mathbb{X}\mathbf{w})^\top = \mathbf{w}^\top \mathbb{X}^\top$ holds

$$\begin{aligned} \frac{\partial\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}})}{\partial\mathbf{w}} &= \frac{1}{N} \left[(\mathbf{Y} - \mathbb{X}\mathbf{w})^\top \cdot \underbrace{\left(\frac{\partial\mathbf{Y}}{\partial\mathbf{w}} \right)}_{=\mathbf{0}} - \mathbb{X} \underbrace{\left(\frac{\partial\mathbf{w}}{\partial\mathbf{w}} \right)}_{=\mathbf{I}} \right] + \dots \\ &\quad \dots \left(\underbrace{\left(\frac{\partial\mathbf{Y}^\top}{\partial\mathbf{w}} \right)}_{=\mathbf{0}} - \underbrace{\left(\frac{\partial\mathbf{w}^\top}{\partial\mathbf{w}} \right)}_{=\mathbf{I}} \mathbb{X}^\top \right) \cdot (\mathbf{Y} - \mathbb{X}\mathbf{w}) \quad (13.17) \end{aligned}$$

$$\begin{aligned} &= -\frac{2}{N} \mathbb{X}^\top (\mathbf{Y} - \mathbb{X}\mathbf{w}) \\ &= -\frac{2}{N} \mathbb{X}^\top \mathbf{R} , \quad (13.18) \end{aligned}$$

which is the same result as in Eq. (13.16). These derivations are probably more detailed than what one would normally find in such a text; however, we wanted to ensure that it is comprehensible even for those who do not have a strong background

in linear algebra. This formulations is also computationally efficient as it only relies on a single matrix-vector product. In analogy to Sect. 12.3.7 and in particular to the steps shown in Sect. 4, we can now use the method of steepest descent or any other suitable numerical minimization technique to obtain the weights, for which the cost is minimized. Additionally, in Sect. 13.2.3 we also derive a closed form solution.

13.2.3 Closed Form Solution

Even a closed form solution for computing the weights $\mathbf{w}$ for which the cost $\mathcal{J}$ is minimized can be obtained. For this, point of departure is Eq. (13.18) together with requiring that $\partial \mathcal{J} / \partial \mathbf{w} = 0$

$$\frac{\partial \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}})}{\partial \mathbf{w}} = 0 \quad (13.19)$$

$$\Rightarrow \quad \mathbb{X}^T (Y - \mathbb{X}\mathbf{w}) = 0 \quad (13.20)$$

$$\Leftrightarrow \quad \mathbb{X}^T Y = \mathbb{X}^T \mathbb{X} \mathbf{w} \quad (13.21)$$

$$\Leftrightarrow \quad \mathbf{w} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T Y, \quad (13.22)$$

where we assumed for inverting $(\mathbb{X}^T \mathbb{X})^{-1}$ that the product $\mathbb{X}^T \mathbb{X}$ results in a non-singular matrix. Any matrix of the form $\mathbb{X}^T \mathbb{X}$ is called *normal matrix* or *Gram matrix*. For inversion of this term, one typically has to resort to iterative numerical solution methods which, however, for larger matrices can become numerically problematic. This closed form solution is in the statistics literature also called *ordinary least squares (OLS)*. An example for a Python implementation of these equations is given in Example 13.1.

Example 13.1 Vectorized Linear Regression in Python



Below is an example implementation for a dataset with four records for the case of simple linear regression. We start by creating the feature $\mathbf{x}$, the target $\mathbf{y}$, as well as vector of weights. Since each of them is a column vector, we use `reshape` to convert it from a 1D array to a 2D array which has one column (the second parameter) and as many rows as necessary (indicated by the “-1”).

(continued)

```
In [1]: import numpy as np

# feature matrix with 1 feature
X = np.array([1.1, 2.2, 3.0, 3.9]).reshape(-1, 1)

# target vector (1 column)
Y = np.array([1.3, 2.4, 2.9, 3.6]).reshape(-1, 1)

# (initial) weights in 1 column
weights = np.array([-0.5, 0.2]).reshape(-1, 1)

# number of data records
n_records = X.shape[0]
```

The extended feature matrix $\mathbb{X}$ is created by prepending a column of “1”s from the left to $\mathbf{X}$ which can be achieved by “horizontally stacking” the column vector and the matrix:

```
In [2]: # stack arrays horizontally
IX = np.hstack((np.ones_like(X), X))
```

Next, the closed form solution for the weights, $\mathbf{w}^*$, for which the loss function reaches a minimum is implemented. There, the numpy function `inv` from the `linalg` module was used for inverting $\mathbb{X}^T \mathbb{X}$.

```
In [3]: exact_weights = np.linalg.inv(IX.T @ IX) @ IX.T @ Y
Y_pred = IX @ exact_weights
Y_pred.reshape(-1) # reshape only for getting compact output
```

```
Out [3]: array([1.37635294, 2.26670588, 2.91423529, 3.64270588])
```

As a small test we also made a prediction for the training data, $Y^{\text{pred}} = \mathbb{X}\mathbf{w}$. The symbol `@` denotes the numpy matrix product, cf. Appendix A.2.4. The shown output is quite close to Y which shows that the implementation seems to work well.

13.2.4 Summary

In this section we have derived an efficient formulation for the class of all multiple linear regression problems. Specifically, this required the introduction of the “extended feature matrix” $\mathbb{X}$ and the consistent use of vector and matrix operations. With this, even a closed form solution for finding the weights that minimize the cost could be derived. An even more general but also more abstract implementation in terms of basis functions will be introduced below.

We now briefly summarize this section and the most important equations.

Things to Remember Multiple Linear Regression



- Matrix-vector notation results in concise and in computationally efficient expressions
- Formulations are generic enough to still apply, e.g., in polynomial regression.
- The most important equations are:

- Residual Error : $R(\mathbf{w}; \mathcal{D}^{\text{train}}) = Y - \mathbb{X}\mathbf{w}$
- Cost and Gradient of Cost : $\mathcal{J} = \frac{1}{N} R^T \cdot R; \quad \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = -\frac{2}{N} \mathbb{X}^T R$

$\mathbb{X}$ is the “extended feature matrix” with the extra column of 1s, and N are the number of records. With these equations, a numerical minimization algorithm can be used to obtain the best weights.

- A closed form solution that minimizes the loss is given by $\mathbf{w} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T Y$

Note that R and Y are both column vectors with as many entries as there are data records. R and Y follow both the notation of the data matrix, Sect. 3.3.3.

13.3 Generalized Formulation of Linear Regression with Basis Functions

In Sect. 13.1.1 we saw that the problem of polynomial regression can be reformulated, by a suitable transformation of the terms $X^0, X^1, \dots, X^k$, in terms of the multi-variable regression problem. In the following we will generalize this approach further using the concept of *basis functions*.

13.3.1 Basis Functions

Linear regression requires that weights occur only as linear coefficients, and, e.g., no squares of a weight or products of different weights are allowed. However, the terms to which the weights are coefficients can take arbitrary shapes. One example that we already encountered are polynomials. Subsequently, these are again used to motivate the introduction of basis functions.

The Example of Polynomials

Mathematically, a polynomial can be constructed from a number of *monomials* which is defined as follows.

Definition 13.2 (Monomial) A *monomial* is a polynomial expression that consists only of a single term in which products of powers of one or more variables occur. Examples are x^2 , xy^2 , x^n , or 1.

The set of all monomials forms a *basis for polynomials*, e.g., it is directly clear that any polynomial of the form

$$p(x) = w_0 + w_1x^1 + \cdots + w_nx^n \quad (13.23)$$

can be created by summing monomials, weighted by coefficients w_i ,

$$p(x) = \sum_{i=0}^n w_i x^i = [x^0 \ x^1 \ x^2 \ \dots \ x^n] \cdot \mathbf{w}, \quad (13.24)$$

where $\mathbf{w}$ is a column vector which contains the weights. The monomial terms are called *basis functions* of the polynomial and are often represented by the Greek ϕ . For a single variable x , the basis functions are given by

$$\phi_0(x) = x^0 = 1, \quad \phi_1(x) = x^1 = x, \quad \dots, \quad \phi_n(x) = x^n. \quad (13.25)$$

A visualization of the first monomial basis functions and a polynomial is shown in Fig. 13.2. The polynomial $p(x)$ can then be written as the scalar product of a vector of basis functions $\boldsymbol{\phi}$ and the vector of weights:

$$p(x) = \boldsymbol{\phi}^\top(x) \cdot \mathbf{w} \quad \text{with} \quad \boldsymbol{\phi} = [\phi_0(x), \dots, \phi_n(x)]^\top. \quad (13.26)$$

An example in the right panel of Fig. 13.2 is given where the first five monomials ϕ_i are superimposed for given coefficients.

13.3.2 Hypothesis with Arbitrary Basis Functions

We now switch back from mathematical notation to “ML notation” with feature X and target Y each of which contain n records. Generalizing the example of the polynomial hypothesis to arbitrary basis functions is straightforward, and we can assume that in all generality, a hypothesis $h(X)$ has the following form:

$$h(X) = \boldsymbol{\phi}^\top(X) \cdot \mathbf{w} = \sum_{i=1}^n w_i \phi_i(X) \quad (13.27)$$

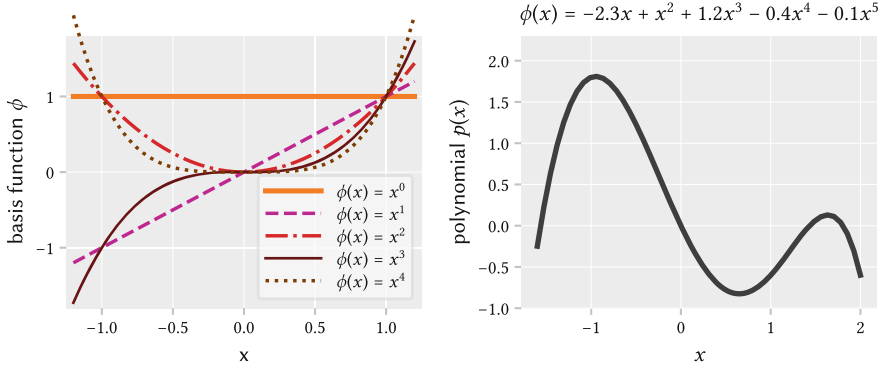


Fig. 13.2 Left panel: The first five monomial basis functions in the range of $x \in [0, 1]$. Monomials are linearly independent, i.e., we cannot construct a ϕ_k by linear superposition of any other monomials of degrees $p, q, r, \dots$ with $p, q, r, \dots \neq k$. This is directly visible for at least the first four monomials. The right panel shows a particular polynomial constructed from the first five monomial basis functions

where ϕ is a column vector consisting of the basis functions $\phi_0, \dots, \phi_n$, each of which maps an element of the feature X to a scalar value. For a single record x_i of the feature X , this is written as

$$h(x_i) = [\phi_1(x_i) \ \phi_2(x_i) \ \cdots \ \phi_n(x_i)] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \quad (13.28)$$

Which are then the features in this model? We again use the same trick as in Eq. (13.2) and map each basis function $\phi_i(X)$ to an individual feature, ξ_i :

$$\phi_1(X) \mapsto \xi_1, \quad \phi_2(X) \mapsto \xi_2, \quad \dots, \quad \phi_n(X) \mapsto \xi_n. \quad (13.29)$$

For a single data record x_i of the feature X , the hypothesis reads

$$h(x_i; \mathbf{w}) = [\phi_1(x_i) \ \phi_2(x_i) \ \cdots \ \phi_n(x_i)] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \quad (13.30)$$

$$\Rightarrow \quad h(\xi; \mathbf{w}) = [\xi_1 \ \xi_2 \ \cdots \ \xi_n] \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} \quad (13.31)$$

This is now the point of departure for a linear regression problem where both weights and features occur only in linear terms, the hypothesis, and in fact each

of the basis functions is linear in the ξ_i , while any nonlinearity is hidden in the mapping of the basis functions. In other words, we started from an one-dimensional input which can occur in form of an arbitrary functional, increase the dimension of the problem to a higher-dimensional problem with n features, and thereby turn it into a canonical problem that is linear in both the weights and the feature. This allows to perform all subsequent steps in great generality.

13.3.3 Extended Feature Matrix with Arbitrary Basis Functions

The above “linearized” hypothesis can now be written for the whole dataset in matrix-vector notation for the prediction Y^{pred}

$$\underbrace{\begin{bmatrix} y_1^{\text{pred}} \\ y_2^{\text{pred}} \\ \vdots \\ y_n^{\text{pred}} \end{bmatrix}}_{Y^{\text{pred}}} = \underbrace{\begin{bmatrix} \xi_{11} & \dots & \xi_{1n} \\ \xi_{21} & \dots & \xi_{2n} \\ \vdots & \ddots & \vdots \\ \xi_{m1} & \dots & \xi_{mn} \end{bmatrix}}_{=: \mathbb{X}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}}_{\mathbf{w}} \Rightarrow Y^{\text{pred}} = \mathbb{X} \mathbf{w}, \quad (13.32)$$

where each entry of the target vector and each row of the matrix corresponds to one data record. This equation looks identical to what we would expect for a multiple linear regression problem. There, $\mathbb{X}$ is again the extended feature matrix. Below we will show that from this formulation we can very easily derive all so far considered models of multiple or polynomial regression as special cases.

13.3.4 Derivation of Cost Function, Cost Gradient, and the Closed Form Solution

As in the case of multiple linear regression, we define the residual error of the i -th data record (cf. Eq. (13.12)) as

$$r_i = y_i - y_i^{\text{pred}} \Rightarrow R = Y - \mathbb{X} \cdot \mathbf{w}. \quad (13.33)$$

If the training cost is again based on a *ordinary least squares* (OLS) formalism, then we obtain the same formulation as in Eq. (13.12) for the cost function $\mathcal{J}(w_1, \dots, w_n; \mathcal{D}^{\text{train}})$. The derivations of the gradient of the cost functions w.r.t. the weights shown in Eqs. (13.16) and (13.18) were done for the problem of multiple linear regression. However, since only a generic, extended feature matrix $\mathbb{X}$ was assumed, the derivations did not require any further assumptions. Because we managed to transform the problem of linear regression with arbitrary basis function into the same mathematical structure as used for those derivations, the mathematical form of the results are the same as for the multiple linear regression. Last but not least, taking a look at the derivation of the closed form solution in Sect. 13.2.3 we see

that also this derivation can exactly be transferred to our current problem. Therefore, we only summarize our results.

Things to Remember Important Formulations Using Basis Functions

With basis functions one can use the same formalism as for multiple linear regression:

- Mapping of Basis Functions: $\phi_i(\mathbf{x}) \mapsto \xi_i$
- Extended Feature Matrix, e.g.: $\mathbb{X}_k = [\xi_{k1}, \dots, \xi_{kn}]$ (for row k)
- Hypothesis: $h(\mathbb{X}; \mathbf{w}) = \mathbb{X} \cdot \mathbf{w}$
- Residual Error: $R = Y - h(\mathbb{X}; \mathbf{w})$
- Cost Function: $\mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = \frac{1}{N} R^T \cdot R$
- Gradient of Cost Function: $\frac{\partial}{\partial \mathbf{w}} \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \mathbb{X}^T \mathbf{r}$
- Closed Form Solution:
(OLS estimator) $\mathbf{w} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T Y$

13.3.5 Numerical Aspects and Discussion

Closed Form Solution vs. Gradient Descent

Clearly, this is a very convenient result as it allows to use only one regression framework that is able to cover all different types of linear regression. Also, the availability of a closed form solution for *all* linear regression problems is very helpful. Why then bother with computing the gradient of the loss function and numerical minimization of the loss at all? There are two reasons: First of all, the closed form solution disguises the fact that in many real life situations, e.g., when handling large datasets with many features, the inversion of $\mathbb{X}^T \mathbb{X}$ also requires iterative numerical solvers. Even if the matrix $\mathbb{X}$ is not a dense matrix and therefore can be stored as a sparse matrix (see, e.g., the `scipy.sparse` module), the inverse of the product might turn out to be a dense matrix and therefore might require a huge amount of memory. Gradient descent (or other more efficient methods) requires more or less only to store the original, sparse matrix, the target vector and the weights. The second reason why one might not want to use the closed form solution is that linear regression is about the only method for which such a solution exists. All other methods have to resort to iterative optimization methods. Therefore, it might be desirable to have a common “platform” (the numerical optimizer) with which all methods can nicely interface. However, as long as you only want to quickly and

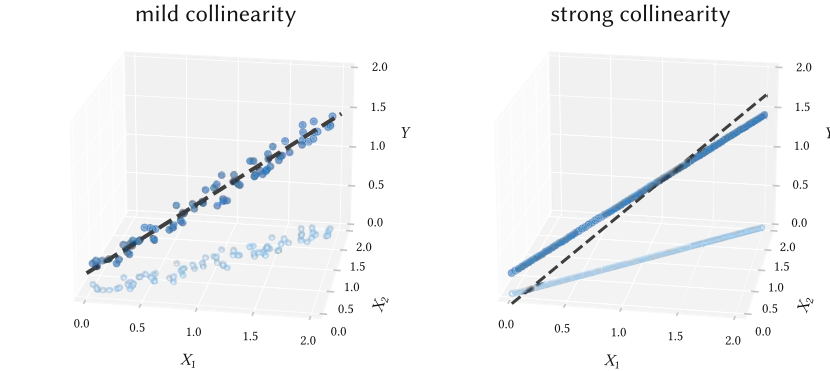


Fig. 13.3 The problem of collinearity: Dark marker are obtained from a linear equation with two features that were superposition of with random noise. Dashed lines are predictions obtained from the closed form solution for the weights. The light-colored markers are the projection into feature space in which collinearity between features can be easily observed. The left dataset contains significant noise, but the predicted line fits well. The right dataset has barely visible noise, but due to the collinearity between X_1 and X_2 , the error during matrix inversion of the estimator (and thus the prediction) is large. Removing the noise entirely would result in a singular matrix

with a few lines of code solve a linear regression problem, the closed form solution could be a good starting point.

Multicollinearity

As a last mathematical aspect, we want to address the problem of *collinearity* (or equivalently *multicollinearity*): if two of the features exhibit a very strong, linear relationship, one variable can be linearly predicted by the other variable; compare Fig. 13.3. This introduces redundancy in the dataset which has the effect that the extended feature variable has not full rank (cf. Appendix A.2.15) and therefore cannot be inverted. As a consequence, neither the closed form solution (OLS estimator of the weights) nor the numerical minimization strategy are able to determine the weights. In reality, two variables are usually never perfectly collinear. Nonetheless, approaching this situations already deteriorates the *condition number* of $\mathbb{X}$ (which states how sensitive a (linear) function is w.r.t. small changes in the input), and therefore, the inversion can at least be highly inaccurate.

13.4 Formulation of Chosen Cases in Terms of Basis Functions

In this section the general formalism for linear regression using basis functions will be used to derive some of the most common linear regression models.

13.4.1 Simple Linear Regression

The following hypothesis governs a simple linear regression problem:

$$h(\mathbf{w}; X) = w_0 + w_1 X , \quad (13.34)$$

which has only one feature, X . The training dataset consists of one column of input data, x_i , and the corresponding target data y_i . The solution of this problem results in the line that fits best to the given data (in a [OLS](#) sense). The basis functions can be identified as the following linear functions:

$$\phi_0(X) = X^0 \mapsto \xi_0 \quad \text{and} \quad \phi_1(X) = X^1 \mapsto \xi_1 . \quad (13.35)$$

After the “linearization” the basis functions ϕ_0 and ϕ_1 are considered as the two new variables, ξ_0 and ξ_1 . With those we arrive at the canonical form of the hypothesis, written for one data record:

$$h(\boldsymbol{\xi}; \mathbf{w}) = [\xi_0 \ \xi_1] \cdot \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} . \quad (13.36)$$

This can now be written for n data records such that the extended feature matrix $\mathbb{X}$ has n rows and two columns. All relations concerning the cost function, its derivatives, and the closed form solution still hold.

13.4.2 Multiple Linear Regression

The following hypothesis governs a multiple linear regression problem:

$$h(\mathbf{w}; X_1, \dots, X_n) = w_0 + w_1 X_1 + \dots + w_n X_n \quad (13.37)$$

The solution of this problem results in the best fitting hyperplane for the given data. The basis functions can be identified as the linear functions:

$$\begin{aligned} \phi_0(x_1, \dots, x_n) &= 1 \mapsto \xi_0 , \\ \phi_1(x_1, \dots, x_n) &= x_1 \mapsto \xi_1 , \\ \phi_2(x_1, \dots, x_n) &= x_2 \mapsto \xi_2 , \\ &\vdots \\ \phi_n(x_1, \dots, x_n) &= x_n \mapsto \xi_n , \end{aligned} \quad (13.38)$$

Considering the basis function ϕ_i as the i -th feature X_i , we arrive at the canonical form, written for one data record

$$h(\boldsymbol{\xi}; \mathbf{w}) = [\xi_0 \ \xi_1 \ \cdots \ \xi_n] \cdot \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix}, \quad (13.39)$$

and all relations concerning the cost function, its derivatives, and the closed form solution apply.

13.4.3 Polynomial Regression

The most commonly used set of basis function for polynomial regression are monomials as introduced in Sect. 13.3.1 and in particular in Eq. (13.25) which we repeated here for completeness:

$$\phi_0(x) = 1 \mapsto \xi_0, \quad \phi_1(x) = x^1 = x \mapsto \xi_1, \quad \cdots, \quad \phi_n(x) = x^n \mapsto \xi_n. \quad (13.40)$$

After the “linearization” the basis functions ϕ_0 through ϕ_n are considered as the $(n + 1)$ new variables, $\xi_0, \dots, \xi_n$. With those we arrive at the canonical form of the hypothesis, written for one data record:

$$h(\boldsymbol{\xi}; \mathbf{w}) = [\xi_0 \ \cdots \ \xi_n] \cdot \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix}. \quad (13.41)$$

This can now be written for m data records such that the extended feature matrix $\mathbb{X}$ has m rows and $(n + 1)$ columns. All relations concerning the cost function, its derivatives, and the closed form solution still hold. It is also directly clear from the above equations that simple linear regression is a special case of polynomial regression. We will now write out some of the governing equations for polynomial regression.

Feature Matrix

According to Sect. 13.3.3 the extended feature matrix $\mathbb{X}$ (which is sometimes also called *design matrix* in the statistics literature) has the following form:

$$\mathbb{X} = \begin{bmatrix} \phi_0(x_0) & \phi_1(x_0) & \cdots & \phi_n(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \vdots & \vdots & & \vdots \\ \phi_0(x_m) & \phi_1(x_m) & \cdots & \phi_n(x_m) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \quad (13.42)$$

This type of matrix is known as *Vandermonde matrix* where each entry in row j and column j is given by x_i^{j-1} . The first column of ones is the motivation for choosing the blackboard typeface for $\mathbb{X}$, there the additional diagonal bar represents the column of ones. This matrix can now be used for evaluating the closed form solution or for computing cost gradients for a numerical minimization. An example is shown in Example 13.2. Typically, predictions are made using the original feature X and Eq. (13.1); in principle, also the transformed features X_i together with Eq. (13.3) could be used, but this is less common and also makes inference more difficult.

Words of advice



Writing the columns of the extended feature matrix as various powers of the features can introduce numerical instabilities, cf. Sect. 13.4.5. In such cases, using orthogonal polynomials [1] such as CHEBYSHEV polynomials might be beneficial.

Further Commonly Used Expressions

For the sake of completeness, we now derive and show the most commonly used expressions in index notation which could also be obtained without the concept of basis functions.

The gradient of $\mathcal{J}$ w.r.t. the weight w_j is obtained from Eq. (13.12) as

$$\frac{\partial}{\partial w_j} \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = \frac{1}{N} \sum_{i=1}^n \frac{\partial}{\partial w_j} r_i^2 = -\frac{2}{N} \sum_{i=1}^n r_i \frac{\partial h(x_i; \mathbf{w})}{\partial w_j} . \quad (13.43)$$

With the definition of the hypothesis for polynomial regression, Eq. (13.1), we can explicitly state the formulation of the individual derivatives of the hypothesis

$$\partial_{w_0} h(x_i; \mathbf{w}) = x_i^0 = 1 , \quad \partial_{w_1} h(x_i; \mathbf{w}) = x_i^1 , \quad \dots , \quad \partial_{w_j} h(x_i; \mathbf{w}) = x_i^j , \quad (13.44)$$

Upon insertion of the derivatives into Eq. (13.43), we obtain

$$\frac{\partial}{\partial w_j} \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^n r_i x_i^j . \quad (13.45)$$

For the special case of only two weights, we recover the formulation of simple linear regression:

$$\frac{\partial}{\partial w_0} \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^n r_i \quad \text{and} \quad \frac{\partial}{\partial w_1} \mathcal{J}(\mathbf{w}; \mathcal{D}^{\text{train}}) = -\frac{2}{N} \sum_{i=1}^n r_i x_i . \quad (13.46)$$

Example

An example implementation of polynomial regression together with a numerical minimization strategy using the steepest descent method can be found in Example 13.2. There, it also can be seen that the computations are very compact and the Python code is almost identical to the mathematical formulations. This is one of the benefits of the vectorized approach that exclusively uses vector and matrix products. This also can make code maintenance and debugging easier—at the price that one has to have some control about the matrix operations in numpy.

Example 13.2 Vectorized Polynomial Regression in Python



Creating the data set, cf. Example 13.1, e.g., for the use of `reshape`.

```
In [1]: import numpy as np
X = np.array([1.1, 1.4, 2.1, 3.1, 3.7, 4.2]).reshape(-1, 1)
y = np.array([1.2, 1.5, 1.7, 2.1, 2.3, 2.9]).reshape(-1, 1)
n_records = X.shape[0] # number of rows = number of data records
```

Next, initialize the “extended feature matrix” as a Vandermonde matrix. `numpy.vander` requires the first argument to be a 1D array, hence the `flatten()` function. Setting `increasing=True` defines that the “1s” form the first column and not the first row.

```
In [2]: polynomial_degree = 3
IX = np.vander(X.flatten(), N=polynomial_degree + 1, increasing=True)
```

Next, we define some numerical parameters as well as the initial values for the weights vector and do the numerical minimization in analogy to the previous chapter.

```
In [3]: learning_rate = 0.0005
max_steps = 10000000
tolerance = 5e-7
weights = np.array([0.2, 0.2, -0.2, 0.2]).reshape(-1,1)
```

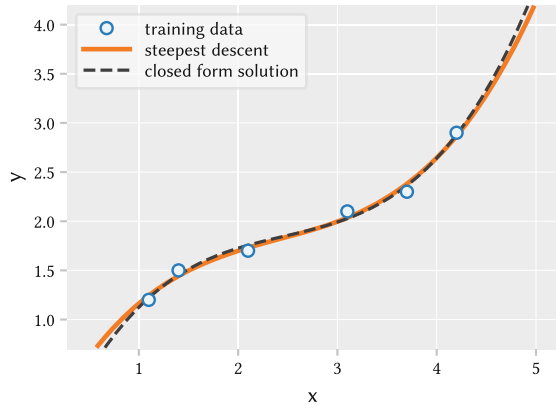
```
In [4]: for step in range(max_steps):
    residual = y - IX @ weights
    grad_of_loss = - 2 / n_records * IX.T @ residual
    weights_increment = -learning_rate * grad_of_loss
    weights += weights_increment

    if np.linalg.norm(weights_increment) < tolerance:
        break
else:
    print("max. number of steps reached w/o converging")
```

Finally, we give a bit of output:

(continued)

Fig. 13.4 Visualization of the results from Example 13.2. Shown are the predictions for weights obtained from the steepest descent method and from the closed form solution which serves as a reference



```
In [5]: print("|weight increment|      =", np.linalg.norm(weights_increment))
print("final loss value          =", float(residual.T @ residual / n_records))
print("number of steps taken      =", step)
print("computed weights           :", weights.flatten())
print("closed form weights        :", (np.linalg.inv(IX.T @ IX) @ IX.T @ y).flatten())

|weight increment|      = 4.999996546878193e-07
final loss value          = 0.002882401789420955
number of steps taken      = 1620612
computed weights           : [-0.19861264  1.96952451 -0.70613782  0.09787801]
closed form weights        : [-0.53150882  2.42437848 -0.88765607  0.11996224]
```

A visualization of the results from the example are shown in Fig. 13.4. There, the lines were plotted from the predictions with the two weight vectors $\mathbf{w}$. This was done in a two-step process: First, an array of x -values was created, e.g., by `x = np.linspace(0.5, 5)`. Then, an extended feature matrix $\mathbb{X}$ was obtained for these values by `numpy.vander(x, N=polynomial_degree + 1, increasing=True)`. Finally, the new prediction with the given weights is `ypred = IX @ weights`. There, it can be seen that even though the weights from the numerical minimization do not exactly match the analytical reference solution, the resulting prediction is fairly good. It also can be seen that the number of steps required for finding the weights is very large, which shows again that the gradient descent method is, for practical purposes, not a good choice.

13.4.4 Multiple Polynomial Regression

Polynomial regression with several features is a powerful method for representing multidimensional, nonlinear behavior. We assume that n features X_j are given and that each features occurs in a polynomial without mixed terms (i.e., there are no

monomials that contain both X_j and X_k with $j \neq k$). Furthermore, we assume that the polynomial is of degree $s - 1$, which implies that including the 0-th order monomial, there are in total s monomials. Again, using basis functions turns out to be a very convenient approach since we just have to follow the same “recipe” as before. We can sort and define the basis functions of degree ≥ 1 as follows (the zero degree functions will be included only once):

$$\begin{aligned}
 \phi_{11}(\mathbf{X}) &= X_1 \mapsto \xi_1, & \cdots, & & \phi_{1n}(\mathbf{X}) &= X_n \mapsto \xi_n \\
 \phi_{21}(\mathbf{X}) &= X_1^2 \mapsto \xi_{n+1}, & \cdots, & & \phi_{2n}(\mathbf{X}) &= X_n^2 \mapsto \xi_{2n} \\
 \phi_{31}(\mathbf{X}) &= X_1^3 \mapsto \xi_{2n+1}, & \cdots, & & \phi_{3n}(\mathbf{X}) &= X_n^3 \mapsto \xi_{3n} \\
 &\vdots & & & &\vdots \\
 \phi_{s1}(\mathbf{X}) &= X_1^{s-1} \mapsto \xi_{(s-1)n+1}, & \cdots, & & \phi_{sn}(\mathbf{X}) &= X_n^{s-1} \mapsto \xi_{s \cdot n}
 \end{aligned} \tag{13.47}$$

For brevity, we used $\mathbf{X}$ as function arguments of ϕ which implies all features as arguments, e.g., $\phi_1(X_1, \dots, X_n)$. Since after the “linearization” the ξ_i are now the names of the transformed variable, we numbered them consecutively from 1 to $n \cdot s$. With those we arrive at the canonical form of the hypothesis, written for one data record and including the “1” for the zero-th degree:

$$h(\xi; \mathbf{w}) = \begin{bmatrix} 1 & \xi_1 & \cdots & \xi_{n \cdot s} \end{bmatrix} \cdot \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_{s \cdot n} \end{bmatrix}. \tag{13.48}$$

This could now be written for all m data records and summarized in the extended feature matrix $\mathbb{X}$. Thus, $\mathbb{X}$ has m rows and $n \cdot s + 1$ columns where each column (except for the first column with ones) contains one of the terms from Eq. (13.47).

All relations concerning the cost function, its derivatives, and the closed form solution still hold, e.g., with this formulation the prediction is still obtained from $Y^{\text{pred}} = \mathbb{X}\mathbf{w}$ where $\mathbf{w}$ is the weight vector with $n \cdot s$ entries.

Two-Dimensional Example

As an example we show the extended feature matrix for two features and with polynomials up to a cubic term. For clarity, we directly express the elements of the extended feature matrix in terms of the original features X_1 and X_2

$$\mathbb{X} = \begin{bmatrix} | & | & | & | & | & | & | \\ 1 & X_1 & X_2 & X_1^2 & X_2^2 & X_1^3 & X_2^3 \\ | & | & | & | & | & | & | \end{bmatrix} \equiv \begin{bmatrix} 1 & x_{11} & x_{12} & x_{11}^2 & x_{12}^2 & x_{11}^3 & x_{12}^3 \\ 1 & x_{21} & x_{22} & x_{21}^2 & x_{22}^2 & x_{21}^3 & x_{22}^3 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{m1} & x_{m2} & x_{m1}^2 & x_{m2}^2 & x_{m1}^3 & x_{m2}^3 \end{bmatrix} \quad (13.49)$$

where X_1 and X_2 are column vectors and each entry corresponds to one of the m record. The vector of weights $\mathbf{w}$ has then $n \times s + 1 = 2 \times 3 + 1 = 7$ entries.

13.4.5 Discussion of Polynomial Regression

We are now able to perform linear regression with polynomial basis functions and for arbitrary numbers of features. This is clearly a powerful method; however, there are also certain aspects and limitations that should be taken into account. First of all, polynomial regression is a great method when inference is important. It is very easy to analyze the approximating function mathematically, and even derivatives can easily be obtained for arbitrary points (i.e., feature values). In many cases the regression curve has a clear relationship to an underlying physical phenomenon (e.g., the grain size distribution during normal or abnormal grain growth) which we can easily analyze in terms of details of the polynomial.

However, polynomials don't work well if the data represents a periodic phenomenon. In this case, regression, e.g., with *Fourier basis functions* consisting of sin and cos functions, might be a better choice.

Using polynomials of high orders is often problematic: above we derive a closed formulation for least-squares polynomial regression and a general numerical form. These (and essentially almost all other formulations) are always based on the extended feature matrix which contains the features to the power of up to the degree of the polynomial as columns. For a single feature, this results in a *Vandermonde matrix*. High powers of the input values may result in a system that is numerically very difficult to solve because if in the extended feature matrix the data is taken to the power of, e.g., 30, small changes of the feature can cause large differences. Numerically, this is very difficult as the system of equations might become (close to) singular. This problem also persists for the closed form solutions as it requires to invert the extended feature matrix. This is why in almost all practical situations, polynomial degrees of more than ≈ 15 should be avoided. Instead, piecewise polynomials, as introduced below, might be a good alternative.

Last but not least, there is another problem with polynomial regression that is related to inference aspects. Often, we are interested in derivatives, e.g., in case that we want to find out about a growth *rate* or the plastic strain *rate*. An aspect of polynomials is that their functional form gets simpler upon differentiating, e.g., the polynomial degree is reduced. This, however, is quite opposite to how things in nature typically are: integration smooths out fluctuations while differentiation

increases the complexity. In this respect, polynomials might not be the most suitable candidates in situations where maybe even several derivatives are required.

13.5 Semi- and Non-parametric Regression

The “common” regression is typically associated with *parametric regression* where the goal is to identify a function that fits best to given data. *Non-parametric methods* do not assume a particular mathematical form such as a polynomial, and all components of a model are directly derived from the training data without explicitly having to define a mathematical hypothesis class.

13.5.1 Problem Formulation

Parametric regression methods require a mathematical function h , given by the hypothesis space, which approximates the generally unknown true function f (see Sect. 12.2 for an introduction and definition of these notions). Then, the fundamental model for parametric regression is in all generality given by

$$Y_i = h_i(X_1, \dots, X_n; w_0, \dots, w_m) + \epsilon_i \quad \Leftrightarrow \quad \mathbf{Y} = \mathbf{h}(\mathbf{X}; \mathbf{w}) + \boldsymbol{\epsilon} . \quad (13.50)$$

where Y_i denotes the i -th output variable Y and ϵ_i is an error term.

Non-parametric Methods

Non-parametric methods do not assume a particular mathematical form such as a polynomial, and all components of a model are directly derived from the training data. Thus, there is no hypothesis about a functional relationship involved. The unknown model is treated like a “black box” and is denoted in the following by $\mathbf{m}$ (or m in the scalar case) which allows to formally write

$$Y_i = m_i(X_1, \dots, X_n) + \epsilon_i \quad \Leftrightarrow \quad \mathbf{Y} = \mathbf{m}(\mathbf{X}) + \boldsymbol{\epsilon} . \quad (13.51)$$

$\mathbf{m}$ is a placeholder for an arbitrary algorithm or a method that takes an input and turns it into an output. Note that in this notation $\mathbf{m}$ does not depend on further parameter or weights. Internally, $\mathbf{m}$ may require parameters, but they are not exposed and cannot be controlled or used for, e.g., writing down a mathematical function.

Both equations were written for the general case of multivariate input and vector-values output. In case of a scalar output Y , the hypothesis reduces to h and the model to m . Unlike the parametric model, m can usually not be easily interpreted, making inference and function analysis more difficult.

How does the non-parametric model component look like? There is a range of different classes of approaches. Many of them assume that the approximation will only be determined by the data in the direct vicinity of the point under consideration.

This leads to a local averaging of the neighborhood which effectively smooths the data.

In almost all practical situations, *some* assumptions need to be made, e.g., concerning the mathematical form of the smoothing. Why is this then still called a non-parametric method? The most important aspect is that there is no closed form hypothesis function. Furthermore, $m(\mathbf{X})$ may include some numerical or algorithmic components as well.

Semi-parametric Methods

There is also a type of ML regression in between the parametric and the non-parametric approach which is called *semi-parametric method*. Assume that we observe *some* simple general trend or that we know about an underlying physical relation but the overall dataset shows some additional behavior. This can then be represented by a parametric component that is responsible for the behavior, described through a more or less simple mathematical function. Additionally, this is superimposed with a non-parametric model contribution, cf. Example 13.3 for a materials science example.

Example 13.3 Nonlinear elastic deformation



An example is a dataset that exhibits *overall* a linear increasing behavior but which is superimposed with complicated, nonlinear contributions. A nonlinear elastic material could exhibit such a behavior. For example, the stress-strain response of a hyperelastic material consists of a linear elastic component and deviations thereof. In a phenomenological approach, the linear elastic contribution can be represented as having a parametric model, while the nonlinear contributions can be governed by a non-parametric model.

The resulting model can be written in compact matrix-vector notation as

$$\mathbf{Y} = \mathbf{h}(\mathbf{X}; \mathbf{w}) + m(\mathbf{X}) + \epsilon . \quad (13.52)$$

Examples for non-parametric models are spline regression models which fit a piecewise smooth curve consisting of segments of polynomials or kernel regression methods both of which will be discussed below.

13.5.2 Regression with Piecewise Polynomials

Polynomial regression (Sect. 13.1.1) suffers, among others, from the fact that it is a *non-local method* where changing the position of one data point more or less strongly affects regions of the curve that are further away. This can be avoided by using a number of polynomials (or, as shown in the next subsection, splines) each of

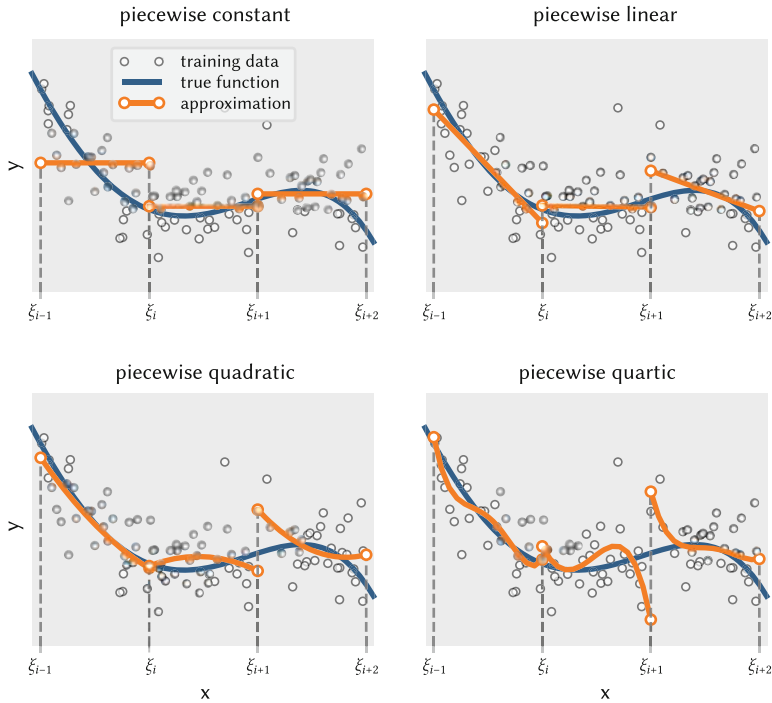


Fig. 13.5 Visualization of different piecewise polynomials. The true function from which the data points (dark marker) are obtained is shown as dark line; the intervals within the piecewise approximation is obtained are indicated by the thin dashed lines which also denote the support points ξ_i of the polynomials

which is fitted to a contiguous subset of the data. The points defining the boundaries of these regions are called *knots*.

Four examples showing a piecewise step function, piecewise linear, and piecewise quartic function are shown in Fig. 13.5. A *piecewise polynomial function* is obtained by only considering the training dataset $\mathcal{D}_i$ within the interval given by the left and right knots, ξ_{i-1} and ξ_i (see Fig. 13.5). The sub-datasets are “non-overlapping” such that for the corresponding datasets of any two intervals i and j , it is $\mathcal{D}_i \cap \mathcal{D}_j = \emptyset$. Furthermore, the datasets of all intervals together contain the total set of data, $\bigcup_i \mathcal{D}_i = \mathcal{D}$. Each interval i is associated with its own polynomial model, trained with $\mathcal{D}_i$.

Each individual polynomial has a finite number of monomials, and thus also the number of parameters (i.e., coefficients of the monomials) is finite; additionally, the whole function approximation consists of a finite number of polynomials, also implying that the total number of parameters increases in proportion to the number of knots. Nonetheless, piecewise regression polynomials or splines are considered non-parametric methods, because the number of parameters can be quite large. Even more importantly, the parameter relation is not directly suitable for inferences, at

least not in the same sense as the methods introduced in Chap. 12 where we could analyze the function and, e.g., its derivatives.

The piecewise constant polynomial is a polynomial of zero-th order and is also called *piecewise step function*. Effectively, it gives the mean of all data inside the respective interval.

The piecewise linear, quadratic, and quartic functions are polynomials of first, second, and fourth degree. The best fitting parameters of the polynomials of all intervals can be obtained as a least square estimate of the particular hypothesis function h , as introduced for polynomial regression in Sect. 13.1.1. As a consequence, the piecewise constant functions is the mean of all $y \in \mathcal{D}_i$, while the piecewise linear polynomial is the regression line for the data in the respective interval. A Python implementation is shown in Python Listing 13.1. An example usage is as follows.

```

1 from numpy import linspace
2 from numpy.polynomial import Polynomial
3
4 def piecewise_fit(x_data, y_data, knots_x, degree, n_interpol_points=10):
5     """Divides data into intervals and fits a polynomial to each ofthem
6
7     :param x_data, y_data: data points (numpy arrays)
8     :param knots_x: list ofthe horizontal positions ofthe support points
9                     (= (n_knots-1) intervals)
10    :param degree: polynomial degree
11    :param n_interpol_points: number of equally spaced points in each interval
12    :returns: lists with the approximated data for each interval
13    """
14    piecewise_data = []
15    for x0, x1 in zip(knots_x[:-1], knots_x[1:]):
16        idx = (x_data >= x0) * (x_data < x1)
17        poly = Polynomial.fit(x=x_data[idx], y=y_data[idx], deg=degree)
18        interp_x = linspace(x0, x1, n_interpol_points)
19        piecewise_data.append((interp_x, poly(interp_x)))
20
21    return piecewise_data

```

Python Listing 13.1 A Python function that returns piecewise polynomial approximations at equidistant points. The main difference to regression codes in the previous section is that we use `idx` to “filter out” all points that are not in the current interval. The resulting data for each interval are stored as a list of tuple of the x and y coordinates

As always, we assume that the training data is already available. It has the structure

```

In [1]: from numpy import array
        X = array([-0.92, 1.69, -0.96, 1.89, 0.14, ...], dtype=float)
        y = array([-0.49, -1.55, -0.37, -1.24, 0.13, ...], dtype=float)

```

The three dots `...` are not valid Python code—replace them by your own data. Then, we do the piecewise training. For this, we use four knots, and in each of these three intervals we evaluated the trained model at 10 equidistant points (note that we need at least 4×2 data records such that in each interval there are at least 2 points which is required by a linear function):

```
In [2]: piecewise_data = piecewise_fit(x_data=X, y_data=y,
                                     knots_x=[-1.8, -0.2, 1.3, 2.2],
                                     degree=1, n_interpol_points=10)
                                     piecewise_data

Out [3]: [(array([-1.8, -1.1, -0.4, ...]), array([ 1.5,  0.0, -1.4, ...])),
          (array([-0.5,  0.2,  0.8, ...]), array([-1.7, -1.5, -1.2, ...])),
          (array([ 0.8,  1.5,  2.2, ...]), array([-0.4, -1.3, -2.1, ...]))]
```

From the output it becomes clear that `piecewise_data` is a Python list containing tuples of the `x` and `y` data (the left and the right arrays). The list has a length of 3 as for each interval we have one tuple of interpolated `x` and `y` data. This makes it easy to plot the approximations for each interval:

```
In [3]: import matplotlib.pyplot as plt
        plt.plot(X, y, 'o', c='C0') # show the training data
        for interp_x, interp_y in piecewise_data:
            plt.plot(interp_x, interp_y, c='C3', lw=3) # show the learned function
```

While the implementation is not complicated, the question arises, how useful are predictions where the hypothesis function is discontinuous, and the function even shows large “jumps” at the support points. The answer is, “it depends”: if the underlying “true” function follows a bilinear form, then a piecewise linear function might be a suitable hypothesis, and discontinuities even might turn out to be only mild. However, another approach are *splines* where the segments are mathematically “glued” together.

13.5.3 Piecewise Regression Splines

In all cases where smooth or at least continuous curves are required, a remedy for the discontinuities at the knots is to “glue” the segments together such that a smooth curve is obtained. This results in the method of *regression splines*—one of the most important non-parametric methods for regression [2].

Usually, the support points ξ_i are then called *control points*, and it is mathematically enforced that the function and the first derivative are continuous. For details of the formulation, please refer to the literature on numerical function approximation, e.g., [3] or [4]. *Natural* splines additionally ensure that the second derivative at the boundary points are zero. Among others, this helps to avoid larger under- or overshooting at the ends, which is one of the problems of linear regression with polynomials of higher degrees. This is known as RUNGE’s phenomenon observed by CARL DAVID RUNGE in 1901 who found that higher-order polynomials do not necessarily increase the interpolation accuracy. A key aspect of piecewise regression splines is that for each interval a cubic, quartic or higher-order spline is created. These individual splines are in a *OLS* sense the best fit to the given points, while at the same time the continuity of the splines and their first (sometimes even higher) derivative across the intervals are enforced.

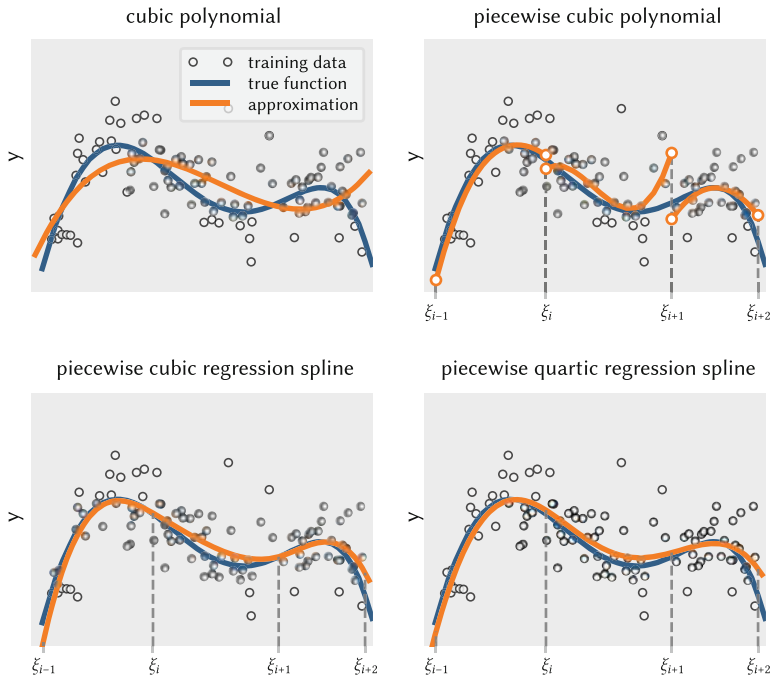


Fig. 13.6 Comparison of four different regression approximations: cubic polynomial (top left), piecewise cubic polynomial (top right). Piecewise splines of 3rd and 4th order (bottom left and right)

Figure 13.6 shows a comparison of four different OLS approximations to a dataset that was obtained from a polynomial of fifth order, superimposed with random noise. The cubic polynomial is not a bad approximation (depending on the dataset extreme oscillations can occur) but is not able to capture the downward trend toward the right of the dataset. The piecewise polynomial approximation exhibits strong discontinuities at the knots. Both approximations are rather sensitive w.r.t. changes in the dataset or in the case of the cubic polynomial, also w.r.t. changes of the knot positions. The two piecewise spline approximations both perform very well and are much more robust than the other two methods.

The Python package SciPy [6] provides an easy to use implementation in form of the SciPy function `LSQUnivariateSpline` which can be imported (besides the usual packages) as follows:

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import LSQUnivariateSpline
```


Next, we create a dataset by superimposing a polynomial of fifth order with random noise:

```
In [2]: x = np.linspace(-1.5, 2.2, 100)
y_true = -2.3 * x + x**2 + 1.2 * x**3 - 0.4 * x**4 - 0.1 * x**5
y = y_true + 1.5 * (np.random.rand(x.size) - 0.5)
```

The SciPy function requires the training data, the inner knots, and the degree of the polynomial. The outer knots are automatically taken as the minimum and maximum of the dataset.

```
In [3]: inner_support_points_x = [-0.5, 1.1]
spl = LSQUnivariateSpline(x, y, inner_support_points_x, k=3)
```

`spl` is an object that contains the information of the fitted spline and that can act as a function, e.g., we can evaluate it at the positions `x`:

```
In [4]: plt.plot(x, y_true, ls='--', label='true function')
plt.plot(x, y, lw=0, marker='o', label='noisy training data')
plt.plot(x, spl(x), label='prediction')
plt.legend()
```

The resulting plot is similar to the one for piecewise cubic regression splines in Fig. 13.6.

This is also a useful example for conducting some further experiments. Among others, it could be seen that the position of the knots place a very important role.

How to Choose the Interval Size

There are a number of options for determining the most suitable interval size and the number of intervals. The first approach just splits the domain into a number of equally sized intervals. As long as the data density is approximately constant, that generally works well. In case that data is distributed very heterogeneously, a better approach is to split the data into quantiles such that each interval contains the same number of data. Last but not least, looking at the data and using our intuition or knowledge about the underlying problem also can help to determine the number and position of the knots.

13.5.4 Kernel Regression with Gaussian Basis Functions

Kernel regression is an approach where all points in a certain neighborhood around a particular position $\tilde{x}$ are averaged. The kernel function acts as a weighting where typically points that are further away from $\tilde{x}$ get a smaller weight value assigned during averaging. Kernel functions with compact support (i.e., roughly speaking, from a certain point on their value is zero) act as a particular kind of efficient averaging window for computing *moving averages*. This operation results

in smoothing of the data which is also the reason why the process is called *kernel smoothing*.

Assume a one-dimensional kernel function $K(x^*, x_i)$ that is centered at one of the training features x_i . Then the prediction of the smoothed target value y^* at x^* is obtained from

$$y^* = \frac{\sum_{i=1}^n K(x^*, x_i) \cdot y_i}{\sum_{i=1}^n K(x^*, x_i)} . \quad (13.53)$$

From this equation, it is directly visible that the kernel function acts as a weighted average of the data around x_i : if the kernel function is just a constant, then y^* is the mean of all data; if it has a constant value in a region around zero and has zero value everywhere else, then this acts as a moving average.

A common choice for a kernel is a Gaussian-like function given by the following equation:

$$K(x, \mu) = \exp\left(-\frac{(x - \mu)^2}{\alpha}\right) , \quad (13.54)$$

where μ is the position of the center and α governs the spread of the distribution in analogy to the standard deviation. Any factor of the exponential function can be removed because (i) in Eq. (13.53) it would cancel out and (ii) in the context of regression with basis functions, the weights act as such pre-factors, as we will see below. As the Kernel contains the Euclidean distance between two points, this particular function is also called *radial basis function (RBF)*—the Gaussian in two dimensions is radial symmetric.

How can these kernel functions be used within the regression framework of basis functions? Starting from Sect. 13.3.2, we observe that the kernel functions *are* already basis functions. We position the *radial basis function (RBF)* at μ_j , and the target value y_i of each data point $\mathcal{P}^{(i)}$ is then weighted according to its position, i.e., the feature value x_i . This is done for all n *radial basis function (RBF)*

$$\phi_1(x_i) = K(x_i; \mu_1, \alpha_1), \dots, \phi_n(x_i) = K(x_i; \mu_n, \alpha_n) , \quad (13.55)$$

where we slightly changed the notation of K for clarity. The hypothesis for some weights $\mathbf{w}$ is then

$$h(x_i; \mathbf{w}) = [1 \ \phi_1(x_i) \ \dots \ \phi_n(x_i)] \cdot \begin{bmatrix} w_0 \\ w_1 \\ \dots \\ w_n \end{bmatrix} , \quad (13.56)$$

and thus we arrived at a formulation that can be used with the linear regression framework as introduced before.

Words of advice



Note that both parameters α and μ_i need to be chosen and fixed *before* the regression is performed; they are neither determined through the training, nor are they properties of the distribution of feature data used for training.

To understand how everything works together, we present a minimal Python implementation. For reuse, we define two functions, `RBF` and a function for assembling the feature matrix with the RBFs, `feature_matrix`. Both are shown in Python Listing 13.2. For using these function, we start by importing numpy and

```

1 import numpy as np
2
3 def RBF(x, mu, alpha=1):
4     """Evaluates the Gaussian RBF at positions x.
5
6     The location and shape is given by mu and alpha.
7     """
8     return np.exp(-(mu - x)**2 / alpha)
9
10
11 def feature_matrix(X, mu_rbf, alpha):
12     """Assemble the feature/design matrix from the RBFs.
13
14     mu_rbf is a list of locations of the RBFs,
15     alpha is 'spread' of the Gaussians.
16     """
17     IX = np.ones((X.size, len(mu_rbf) + 1))
18     for j, mu in enumerate(mu_rbf, start=1):
19         IX[:,j] = RBF(X, mu, alpha)
20     return IX

```

Python Listing 13.2 Kernel regression with radial basis functions: the Python function for the RBF and for creating the feature matrix (to be used with the code snippets in the text)

have to make sure that the two functions are available (e.g., by importing them from a file or by pasting them here):

```

In [1]: import numpy as np
        # either import the functions 'RBF' and 'feature_matrix'
        # or insert their code from the listing (see text) here

```

We generate a dataset from a polynomial and superimpose some random noise:

```

In [2]: X = np.linspace(-1.5, 2.2, 80)
        Y_true = -2.3 * X + X**2 + 1.2 * X**3 - 0.4 * X**4 - 0.1 * X**5
        Y = Y_true + 1.5 * (np.random.rand(X.size) - 0.5)

```

Next, we define the number of kernel functions to be used, their position, and their shape:

```
In [3]: n_rbf = 4                                # number of RBFs
        mu_rbf = np.linspace(-1.8, 2, n_rbf)      # position of RBFs
        alpha = 1                                # shape of all RBFs
```

With this, we can now call the function that populates the feature matrix with the data evaluated by the **RBFs** and compute the weights using the closed form solution for the ordinary least squares approach:

```
In [4]: IX = feature_matrix(X, mu_rbf, alpha)
        w = np.linalg.inv(IX.T @ IX) @ IX.T @ Y
```

For new data, `x`, we can create a new feature matrix keeping all parameters aside from `x` constant and use the weights to make a prediction.

```
In [5]: Y_pred = feature_matrix(X, mu_rbf, alpha) @ w
```

Finally, we can visualize the training data and the predictions

```
In [6]: plt.plot(X, Y_true, label='true function')
        plt.plot(X, Y, lw=0, marker='o', label='training data')
        plt.plot(X, Y_pred, label='prediction')
        plt.legend()
```

which gives a similar result to that shown in one of the panels of Fig. 13.7. In this figure we observe that the method is relatively robust w.r.t. the number of **RBFs** (in the figure we compare 4 **RBFs** to 10 **RBFs**), and we also see that the predictions match the data fairly well for a large range of values α . Furthermore, we observe that for a very small value of α (i.e., a small standard deviation), the individual GAUSSIANS become directly visible (upper left panel of Fig. 13.7). This would become more pronounced and result in overfitting if the number of kernels would be increased. Only if the kernel functions strongly overlap (bottom left panel) then the performance of the method breaks down.

This simple example also demonstrates how versatile and powerful the framework of linear regression with basis functions is. Adding the concept of “kernel smoothing” additionally augments the range of applications.

The shape factor of the distributions, α_i , is also called the *bandwidth* and influences the smoothing. In this regression approach, the α_i usually have the same values. This parameter could be obtained from an parameter optimization where the objective functions can be chosen as the **MSE** of a testing dataset. There are also approaches that determine different values for the α_i , e.g., based on the density of data which is useful in case that the resulting approximation has “holes,” i.e. regions that are not properly covered by the **RBFs**. The position of the kernel functions can be determined similar as what was sketched for the piecewise spline approximation.

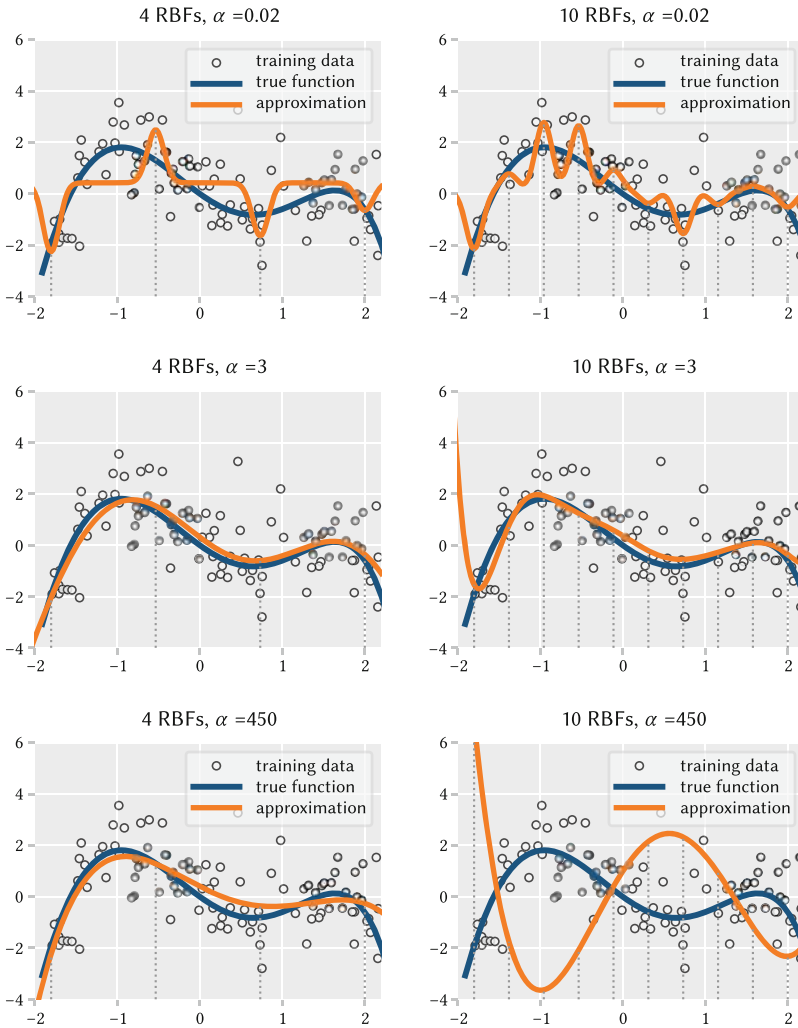


Fig. 13.7 Kernel regression with radial basis functions. The dashed lines indicated the positions of the kernel functions. See the text for more details

Kernel Density Estimation



Kernel density estimation (**KDE**) is based on kernel smoothing and is a way to estimate probability functions. It is contained as a standard tool in many statistics, machine learning, and data visualization toolboxes.

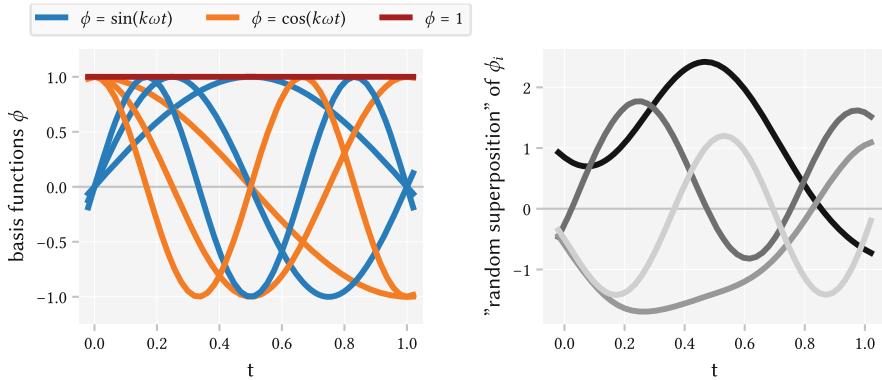


Fig. 13.8 Visualization of Fourier basis terms (left panel) and their superposition with randomly chosen parameters as a function of time (right panel)

13.5.5 Kernel Regression with FOURIER Basis Functions

An effective approach toward data that contains inherent periodicity is the use of *Fourier basis functions*. There, the angular functions $\sin$ and $\cos$ are used in analogy to a *Fourier series* because calculus tells us that almost any function can be approximated by a Fourier series. Obviously, the more Fourier terms are involved the better the accuracy. The Fourier basis consists of terms such as $\sin(\omega t)$, $\cos(\omega t)$, $\sin(2\omega t)$, $\cos(2\omega t)$, and so on. There, ω is the angular frequency, and in general also a *phase shift* φ might be added. The Fourier basis then takes the form

$$\phi_0(t) = 1 \quad \text{and} \quad \phi_k(t) = \cos(\omega_k t + \varphi_k) \quad \text{for } k \geq 1 \quad (13.57)$$

The first basis function is a constant offset. As this is an often encountered situation for periodic phenomena (e.g., the yearly temperature in Europe fluctuates around the mean value of roughly 10°C), it was included here as well. Otherwise, the dataset would have required appropriate preprocessing. A visualization of a few of the first basis functions is shown in Fig. 13.8 which also shows how the superposition of some terms can look like. All subsequent steps would then be done in full analogy to what was introduced in more detail for the Gaussian basis functions in Sect. 13.5.4.

13.5.6 Other Commonly Used Basis Functions

Fourier basis functions for periodic systems and monomial basis functions for non-periodic models have been the first choice for curve fitting and regression problems until the mid-twentieth century. Apart from their great numerical behavior, this is also due to the fact that both methods are accessible for mathematical analysis and therefore are based on a solid theoretical foundation.

While Fourier regression is particularly well suited for data that was obtained from periodic sampling in the time domain, however, for irregularly spaced data, *wavelet regression* is more suitable, which uses a related type of periodic basis functions.

An alternative for periodic problems are sinusoidal functions of the shape $\phi_1(\omega) = \sin(\omega)$, $\phi_2(\omega) = \sin(2\omega)$, $\phi_3(\omega) = \sin(3\omega)$, etc.

13.6 Further Nonlinear Regression Models

While we mainly investigated *linear* regression methods in this chapter, nonetheless, also the field of *nonlinear* regression is relevant and contains a number of important and commonly used methods. Some of them we have already encountered above: almost all semi- and non-parametric models are nonlinear models.

In what follows we very briefly introduce these methods. Even though it is not possible to understand all mathematical and implementation details from these descriptions, it should be sufficient, though, to be able to decide if this or that model might be suitable for a particular problem.

13.6.1 Ridge- and LASSO Regression

Linear regression operates with a hypothesis of the form

$$h(\mathbf{X}) = w_0 + w_1 X_1 + \cdots + w_n X_n + \varepsilon \quad (13.58)$$

where the weights were obtained during training, e.g., from minimizing the mean sum of the quadratic loss

$$\mathcal{J}(\mathbf{w}) = \frac{1}{N} \sum_i (y_i - h(\mathbf{x}_i))^2. \quad (13.59)$$

In Sect. 13.3.5 we already discussed the phenomenon of *multicollinearity* which may occur if two variables are strongly correlated and which can cause inaccurate estimations of the weights and bad generalization to new data.

LASSO regression remedies this problem by introducing a penalty term, thereby creating a new minimization problem of the form

$$\mathcal{J}(\mathbf{w}) = \frac{1}{N} \sum_i (y_i - h(\mathbf{x}_i))^2 + \lambda \sum_{j=1}^n \|w_j\|. \quad (13.60)$$

The term $\lambda \sum_{j=1}^n \|w_j\|$ is denoted as *shrinkage penalty* and acts as a *regularization*.

It has the effect that as this term is increased, the values of the weights are

automatically decreased. Additionally, the variance of the model is decreased, and therefore the model reaches a better test [MSE](#).

Ridge regression is not too different and uses as shrinkage penalty a slightly different term, $\lambda \sum_{j=1}^n w_j^2$. Effectively, this avoids that coefficients can become zero, while LASSO regression allows that some weights can become zero, which implies that these can be removed from the model. Cross validation (Sect. [16.2](#)) typically helps to determine the hyperparameter and to find out which of the two models performs better in a particular situation.

13.6.2 K-Nearest Neighbors Regression

The *k-nearest neighbors* ([kNN](#)) algorithm is one of the few algorithms that can be used both for supervised classifications and for regression. In a classification task, the model searches for the *k*-nearest neighboring data points and then performs a “vote” where the majority “wins” and thereby determines the class of the unknown datapoint. In a regression approach, however, the neighboring data points are typically averaged. [kNN](#) regression is easily able to predict highly nonlinear relations. The underlying [kNN](#) algorithm is explained in more detail in Sect. [14.4](#).

13.6.3 Support Vector Machine Regression

Support vector machines ([SVMs](#)) are another model type that can be used for both supervised regression and for classification. The idea is, roughly speaking, that the decision boundary in a classification problem can be envisioned just as well as a regression curve. The use for classifications problems is more common, and therefore we will discuss it in more detail in Sect. [14.6](#).

13.6.4 Artificial Neural Networks

Artificial neural networks ([ANNs](#)) also belong to the class of nonlinear methods that can be used for regression problems. They will be discussed in detail in later chapters.

13.7 Summary and Conclusion

This chapter had a focus on extending and generalizing methods and models for regression. There the focus was twofold: on the one hand, we formulated the problems from Chap. [12](#) in terms of compact matrix-vector notation and vectorized all operations to achieve a satisfying efficiency. On the other hand, we generalized

the regression framework by introducing the concept of basis functions. This is a “unifying framework” for parametric, semi-parametric, and non-parametric regression and additionally allows to easily incorporate nonlinear model response. Adding the concept of kernel functions increased the number of tools for nonlinear modeling strongly.

13.8 Exercises

13.1 Think of a problem where the fact that derivatives in polynomials result in simpler functions does not fit to the physical reality and discuss alternatives.

13.2 In Sect. 13.5.2 an implementation for piecewise polynomials of arbitrary degree was introduced. Implement the special case for piecewise constant polynomials that does not require `numpy`’s `Polynomial` class.

13.3 RUNGE’s phenomenon, the extreme over- or undershooting behavior of a higher-order polynomial at the boundaries was initially studied using RUNGE’s function $f(x) = 1/(1 + 25x^2)$. Investigate this function for the polynomials of degrees up to 5, and compare this with a least square spline approximation for different numbers of knots $x_i = [-1, \dots, +1]$.

13.4 Following Sect. 14.4, implement the k-NN regression method, and apply it to the MDS-1 (tensile test) dataset. Create a plot of the training, and test MSE and use it to justify your choice of the optimal number of nearest neighbors.

13.5 Explain what parametric and non-parametric methods are and why they are called like this.

13.6 Choose an appropriate ML method for learning the MDS-1 (tensile test) dataset. The method should be as simple as possible and contain the most relevant materials scientific phenomena. Which method do you choose? Implement the method of your choice, and compare your solution to a solution using linear regression with a polynomial of third degree.

13.7 In a dataset of ceramic materials, the recorded properties for each sample include the processing conditions such as temperature, pressure, and time and the corresponding hardness value. A researcher wants to find out the relationship between the processing conditions and the resulting hardness value using regression and regularization techniques. The researcher wants to know which of the processing conditions affect the hardness value the most, i.e., they want to find out the most relevant processing conditions that impact the hardness value and focus more on those and find and discard the irrelevant features. What kind of regularization should they use and why?

13.8 In the previous question, the researcher realizes that there is a strong correlation between the temperature and the pressure features. The presence of strong correlations between the predictors makes the model unstable, resulting in inaccurate estimations and leads to bad generalization to new data. To tackle this, what kind of regularization should they use and how does it help?

References

1. G. Arfken. *Mathematical Methods for Physicists*. Elsevier, 1985. DOI <https://doi.org/10.1016/c2013-0-10310-8>.
2. T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer New York, 2009. DOI <https://doi.org/10.1007/978-0-387-84858-7>.
3. Q. Kong, T. Siau, and A. M. Bayen. *Python Programming and Numerical Methods*. Elsevier, 2021. ISBN 978-0-12-819549-9. DOI <https://doi.org/10.1016/c2018-0-04165-1>.
4. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007. ISBN 0521880688.
5. G. A. F. Seber and A. J. Lee. *Linear Regression Analysis*. Wiley, Jan. 2003. DOI <https://doi.org/10.1002/9780471722199>.
6. P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17: 261–272, 2020. DOI <https://doi.org/10.1038/s41592-019-0686-2>.

“All models are wrong, but some are useful.”

George Box (191–2013)

British statistician

14.1 Introduction to Supervised Classification

Supervised classification of data is a concept where an *machine learning* (ML) model is trained to sort data into different groups, each of which corresponds to a different category or *class*. An important property of categorical data is, that—as opposed to typical numerical data—one cannot interpolate between categories. In supervised classification, the classes are predefined, i.e., the training data consist of a number of inputs, the features, and a single output, the class, which is typically an integer number. This integer, however, can be mapped to text labels (e.g., in a classification model for images, we could assign the value 0 to “car”, 1 to “tree,” 2 to “person,” etc.). The boundary between regions of data points of different classes is called *decision boundary*, e.g., indicated by the line in Fig. 14.1. For more than two features that line becomes a (hyper) surface in multidimensional space.

The most simple ML methods are *linear classifiers*, which aim to split a dataset into two parts using a straight line in two-dimensional feature space and or a (hyper) plane in a higher-dimensional spaces. Thus, linear methods only work if the data are *linearly separable*; however, by certain transformation methods (e.g., used in support vector machines, Sect. 14.6), linearly non-separable data can be made linearly separable. More complex methods are also able to find more complex decision boundaries, e.g., such as the one shown in Fig. 14.1.

In *binary classification*, data are sorted into exactly two disjunct groups each of which corresponds to one class. It can happen that the binary classification algorithm might misclassify some data: a data point of a class A that is wrongly identified as

Fig. 14.1 The decision boundary in supervised classification separates data points that belong to different classes (here represented by markers with different colors). Depending on the ML model a decision boundary might not sort all data points into correct classes, which results in “false positives” and “false negatives” together with a typically less ragged or twisted curve

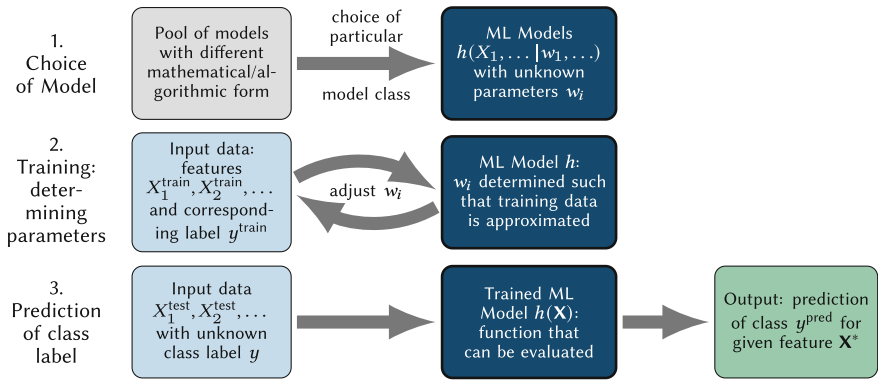
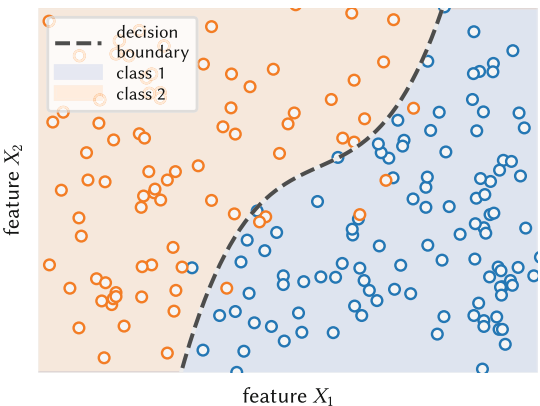


Fig. 14.2 Supervised classification consists of three parts: 1. Selecting a suitable model, e.g., a polynomial; 2. training the model, i.e., finding a suitable set of parameters (e.g., the coefficients of the polynomial) based on a given dataset consisting of input features and the class label for each data point; 3. making predictions with the model h where for each input data the class will be predicted

not belonging to class A is, from the perspective of class A, called a *false negative*. A *false positive* (from the perspective of class A) is a data point from class B that was wrongly labeled as belonging to class A. Section 11.8.1 introduces these measures in more detail. If more than two classes are present, the problem of sorting the data points into one of these classes is called a *multiclass classification*.

The three, general steps during supervised classification are shown in Fig. 14.2, which is similar to what we encountered for the case of regression in Fig. 12.3. Again, as a first step, the choice of a specific model class is included. While this is not always explicitly considered in the literature, it is an important step as the performance of the trained model strongly depends on this decision (also refer to the introduction in Chap. 12). For instance, the choice of the model determines whether the decision boundary can take the shape of a straight line or of a complex, curved

line. Which one is more appropriate depends on the data and the purpose of the classification, i.e., besides data science aspects, domain-specific aspects play a role. The mathematical function or the algorithmic form of the model has a number of parameters that need to be determined. As in regression, this is done in the subsequent training process where each data record is given by the input matrix $\mathbf{X}$ and the corresponding target variable Y .

Words of advice



In classification, the output matrix consists of only a single target variable, which gives the label of all features. Therefore, in classification, we almost never use the output matrix $\mathbf{Y}$ but instead only the target variable Y . Then the dataset is written as $\mathcal{D} = (\mathbf{X}; Y)$.

Additionally, the dataset needs to be split into a training and testing set. During the training, the model parameters θ_i are then determined such that the labels of the training data can be reproduced as good as possible. Finally, the model can be used to make predictions or inference, in analogy to Fig. 12.3. In classification, though, prediction is typically more important than inference because often the decision boundary is not governed by a simple mathematical functions (e.g., a polynomial) but rather by some numerical algorithm that does not allow to perform straightforward “function analysis.”

In the following sections, a number of ML classification models and algorithms are introduced in order of roughly increasing complexity. This is not a complete introduction of *all* possible methods. We rather focus on a chosen methods, and introduce them in details, such that the reader should be able to understand how the respective methods work and what the consequences are for their applications.

14.2 Rule-Based Classification Methods and Decision Trees

An intuitively comprehensible type of classification methods are rule-based methods, which, in some aspects, are very close to rules that we express through language. Another related method is decision trees. In the following, we explain both and then discuss the relationship between them.

14.2.1 The Idea Behind Classification Rules

Rule-based classification methods rely on a composition of one or more IF-THEN conditionals. Each such conditional consists of an *antecedent* (the precondition in the IF part) and a resulting *consequent* of the rule. Here is an example for such a rule:

IF *weather is sunny* AND *there is no school* THEN *Nice cream parlour ...*
...has many customers.
 (14.1)

The antecedent of the rule consists of two conditions (*weather is sunny* and *there is no school*), which are combined by a logical AND. The consequent (*ice cream parlour has many customers*—or not) is then the actual prediction of this simple classification rule: it predicts whether or not there are many customers in the ice parlour based on two inputs. The above rule can also be written more formally as

$$(weather = sunny) \wedge (school\_day = no) \Rightarrow \dots$$

$$\dots ice\_cream\_parlour\_has\_many\_customers = \text{True},$$
(14.2)

where the logical AND is represented by the wedge symbol ($\wedge$). This expression looks already quite formal and as if it is already in a shape, such that it could be easily implemented in a computer program. However, we still have to define and quantify some of the notions. For instance, what is *weather*? And when is it considered *sunny*? Is it sunny if the sun shines in a particular moment or do we require the past hour to have non-stop sunshine? And what if there are also a few clouds? We probably should refine and define what is important about the weather. As a first idea, we can use the average number of sunny hours per day, n_{sun} , and require them to be above a certain number of hours, N_{sun} .

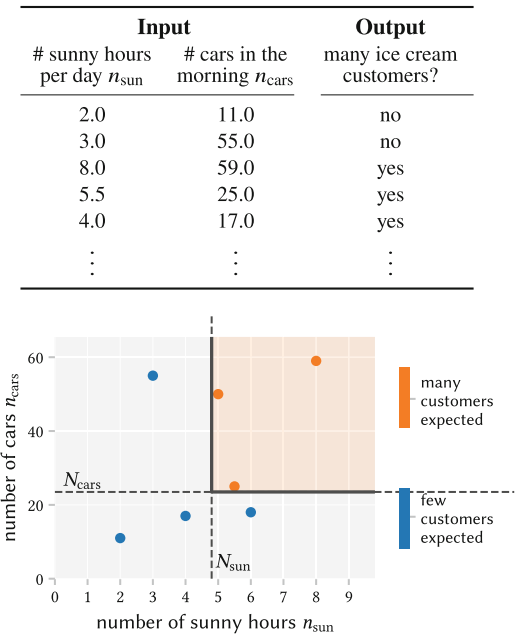
But how we can we detect if it's a school day and not a Sunday or a vacation day? Assume that the computer had access to, e.g., the traffic information of the road toward the school. This could yield the information of how many cars drove by in the morning. Let's call this number n_{cars} . On a school day there will be many cars in the morning while on any other day there will be just a few. Again, using a threshold value N_{cars} and comparing it to the actual number of cars help to make the decision. Now the problem can be restated as

$$ice\_cream\_parlour\_has\_many\_customers = (n_{\text{sun}} \geq N_{\text{sun}}) \wedge (n_{\text{cars}} \geq N_{\text{cars}}).$$
(14.3)

So far so good, but where is the “learning” aspect here? For determining the two threshold values, N_{sun} and N_{cars} , we can make an educated guess, or we could attempt to do a purely mathematical derivation, we could create a simulation model—or we could do this in a data-driven manner and use training data to determine them.

Figure 14.3 shows some records of a (artificial) dataset. Just by looking at the data, we find that the threshold value N_{sun} should be somewhere in between 2 and 5 hours, the threshold N_{cars} will be in between 19 and 25 cars. A computer program would use an automated strategy to find the threshold values—the program *learns* the data. In this case, all training data were reduced to only two numbers, the

Fig. 14.3 “Learning from data” whether an ice cream parlour should expect many customers (only they know, how many “many” are) based on the two features (i) number of sunshine hours and (ii) traffic density at the school as an indicator for whether it’s a school day or not. Based on the data in the table, the threshold values N_{sun} and N_{cars} were determined such that in the diagram both classes are completely separated. For making predictions, these two parameter are “fixed” and whenever a point falls into the orange-shaded region, the result will be that many ice cream customers are to be expected



two threshold values. Clearly, these two values cannot tell us where *exactly* all the training data points were located but it *can* tell us something about the more general aspects of the dataset.

For this example, the learning strategy that resulted in Fig. 14.3 was rather simple: *take the positive examples and enclose them in a rectangle with minimal area; the lower left corner gives N_{sun} and N_{cars} .* This is clearly not a great strategy as, e.g., it is very sensitive to outliers, but at least, it is very simple to implement. More robust solutions are based on an iterative optimization algorithms that minimize an error. However, for this, we need to define an error measure for such classification problems, which will be done below.

The result of the learning strategy for learning the (single) class of “many customers expected” can be seen in the plot in Fig. 14.3: there, the two solid lines separate the region of *positive examples* (the orange data points) for the class “many customers expected” from the *negative examples* (the blue data points). The line indicating the boundary of the class is called *decision boundary*.

Finally, we can now make predictions using the two threshold values: for a new input data record $\mathbf{x}^* = (n_{\text{sun}}^*, n_{\text{cars}}^*)$ we evaluate Eq. (14.3) with the now fixed parameters N_{sun} and N_{cars} and predict, if there are many customers to be expected today. Using more data records will (often/mostly/hopefully) increase the accuracy of the prediction.

Alternatively, we can also use the equation to analyze the structure of the data, e.g., by plotting those feature values that result in high customer numbers (that’s the

shaded area in Fig. 14.3). Additionally, we directly observe that such kind of rules typically result in straight decision boundaries (or combinations thereof).

Words of advice

This is a very simple example, and the “learning strategy” is even more simplistic. However, it helps to already understand a few core aspects. To make sure that you understand some limitations of the approach think about what happens to the decision boundary when more positive or negative data, or when a positive and negative one were swapped, by accident.

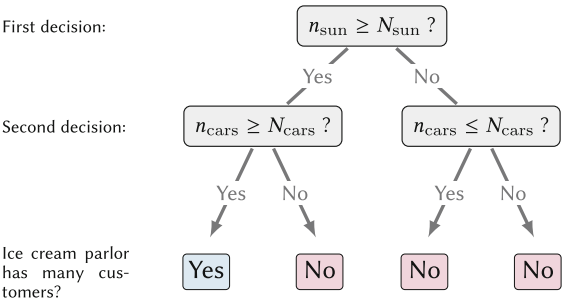
14.2.2 The Relation of Rule-Based Methods and Decision Trees

Any set of rules has a corresponding decision tree, and vice versa. However, even simple rules can lead to complex decision trees. If the antecedent of a rule consists only of conditions that are combined by logical ANDs, then a tree can be constructed by “chaining” the conditions as shown in Fig. 14.4. There, the consequent of the rule is given by the leaf on the very bottom, which assigns the class label “yes” or “no.”

Creating a rule based on a tree (“rule extraction”) typically results in more complex formulations than necessary. The reason is that the most intuitive approach is to create one rule for each path from the root to each leaf. Subsequently, all rules are combined through a logical AND, resulting in the antecedent.

IF ($n_{cars} \geq N_{cars}$) $\wedge$ ($n_{sun} \geq N_{sun}$) THEN ice_parlour_has_many_customers = True
IF ($n_{cars} < N_{cars}$) $\wedge$ ($n_{sun} \geq N_{sun}$) THEN ice_parlour_has_many_customers = False
IF ($n_{cars} < N_{cars}$) $\wedge$ ($n_{sun} \geq N_{sun}$) THEN ice_parlour_has_many_customers = False
IF ($n_{cars} < N_{cars}$) $\wedge$ ($n_{sun} < N_{sun}$) THEN ice_parlour_has_many_customers = False.

Fig. 14.4 Decision tree for the rule Eq. (14.3). There are two conditions, each of which has two results. Therefore, there are altogether four scenarios that only result in “yes” if both conditions are satisfied



By analyzing the tree beforehand and realizing that the two conditions on the second level concerning the number of cars are identical one could already optimize the rules. However, trees can also be unsymmetrical regarding the decisions, which makes “compressing” such rules sometimes difficult.

There are a number of further practical aspects that rather suggest the use of rules. Among those are simple extensions of rules that can be quite powerful, e.g., a default cases can be included using an ELSE statement

$$\text{IF } (x > N) \wedge (y > x + 2) \text{ THEN class A ELSE class B.} \quad (14.4)$$

Visualizing this in a decision tree is significantly more complicated than reading such a decision rule. Other extensions are exceptions, e.g.,

$$\begin{aligned} &\text{IF } (x > 0) \wedge (y > 0) \text{ THEN class A ELSE class B} \\ &\text{EXCEPT } 3 \leq x \leq 5 \text{ THEN class B} \end{aligned} \quad (14.5)$$

which represents already a relatively complicated rule that is nonetheless still very comprehensible. As an additional benefit, such rules can also easily be implemented in a Python code. However, both classification rules and decision trees are important tools, in particular because the way how decisions are made can be understood and related to domain knowledge. Here, we only summarize a few of the most important definitions, aspects, and notions for classification rules:

- *Mutually exclusive*: If two rules are covering two different data records, then they are *mutually exclusive*. If more than two rules need to be considered for a single data record, then they are said to be *not mutually exclusive*.
- *Ordering and Voting*: How can one decide which of the rules that are not mutually exclusive determine the class? Either, rules need to have different priorities and can be ordered such that the first rule that is “activated” is responsible for the class prediction. An alternative is to perform a *voting* among the rules for a each records’ class, possibly including some weighting. There, the rules do not have to be ordered, which may have additional implementation advantages.
- *Non-exhaustive rules*: If records of a dataset are not covered by the rules, then the set of rules is said to be *not exhaustive*.
- *Coverage*: The notion of *coverage* of a rule-based classifier quantifies how many records of a dataset are handled by a specific rule. Coverage is a number n_{coverage} that gives the percentage of the records in a dataset that fulfill the antecedent conditions. It is defined as

$$n_{\text{coverage}} = N_r / N_{\mathcal{D}} \quad (14.6)$$

where N_r denotes those records that are covered by the rule and $N_{\mathcal{D}}$ is the total number of records.

- *Accuracy of a rule:* The *accuracy of a rule* is defined as the number of correctly classified records divided by the total number of records that are covered by the rule under consideration.

Decision trees can be strongly generalized, and there exist many sophisticated methods for creating, compressing, and optimizing them. This, however, is beyond the scope of this text. For further information, the reader is referred to the large body of literature (see, e.g., the introduction in [4]).

14.3 Notions and Concepts for Classification Problems

Above, we introduced classification trees and rule-based methods in an intuitive manner as one way of defining an ML model. To systematically explore the structure of general classification problems and various methods, we will now introduce important notions, mathematical foundations, and concepts.

We assume that the reader is already familiar with the concept of splitting the data into training and testing data and with the hypothesis. Both were introduced in Sects. 12.2.10 and 12.2.5, respectively.

14.3.1 Datasets, Models, and the Hypothesis Class

We assume a dataset $\mathcal{D}$ with two given features X_1 and X_2 and known classification, i.e., the label Y . It consists of altogether $N_{\mathcal{D}}$ records (which are also called *examples*),

$$\mathcal{D} = \{(x_{i1}, x_{i2}; y_i)\}_{i=1 \dots N_{\mathcal{D}}}. \quad (14.7)$$

A supervised learning algorithm should then, based on the training data, yield a *hypothesis* h that is able to map the input data (X_1, X_2) to the output classification, Y . Clearly, there exist many different algorithms or mathematical functions that might be suitable for this task (just think of all the possible rectangles enclosing the positive examples in the plot of Fig. 14.3). The learning algorithm is responsible for finding the particular parameterization that is “most suitable” (we’ll discuss later what this exactly implies).

The above introduced classification rule, Eq.(14.3), defines a whole class of hypotheses each of which has the some structure but differs in the two parameters N_{sun} and N_{cars} . The collection of all hypotheses, i.e., feasible versions with different parameterizations is then the *hypothesis class*.

We will now explain this based on a more complex example, dataset DS-1 (Iris Flowers, Sect. 4.7), which consists of the measured width and length of the petal and the sepal of iris flowers and the corresponding name of the species. The goal of classification is here to find the name of the flower species (the class label) based on the measurements. An ML algorithm is typically operating with numbers. Therefore,

the three flower names (class labels) were encoded as integers 0, 1, and 2, the so-called *class IDs* (see the description of the dataset in Sect. 4.7 and in particular the table in Fig. 4.7a). Here, we only focus on two input features, the sepal length and width. In a first step, the goal is to train an ML model to identify the iris setosa based on the sepal length and width; the other two features contained in the dataset petal length and width, are not used.

A simple starting point for identifying the iris setosa species can be expressed as the plain language rule: *all data points contained in a particular region belong to class with ID=0*. Additionally, the region still needs to be mathematically described by a parametric, geometrical shape, e.g., a rectangle. As we saw already above, rectangles can be very conveniently defined by rules, e.g., all points inside a rectangle, which is given by the lower left corner (x_1, y_1) and the upper right corner (x_2, y_2) , are given by

$$\text{is\_inside\_rectangle} = (x_1 \leq x \leq x_2) \wedge (y_1 \leq y \leq y_2). \quad (14.8)$$

However, so far, the four parameters $x_1, \dots, y_2$ are not determined and thus, Eq. (14.8) constitutes the hypothesis class $\mathcal{H}$. The class $\mathcal{H}$ consists of infinitely many rules, each of which is defined by a different set of values (x_0, y_0, x_1, y_1) .

The dashed box in Fig. 14.5 visualizes a particular hypothesis (i.e., rule for a particular parameter combination): if a data point is contained inside the rectangle given by $x_1 = 0.0, y_1 = 0.6, x_2 = 0.8, y_2 = 2.4$, then it belongs to class 0 (iris setosa). Here, the four sides of the rectangle define the *decision boundary*, a notion that we will now define as:

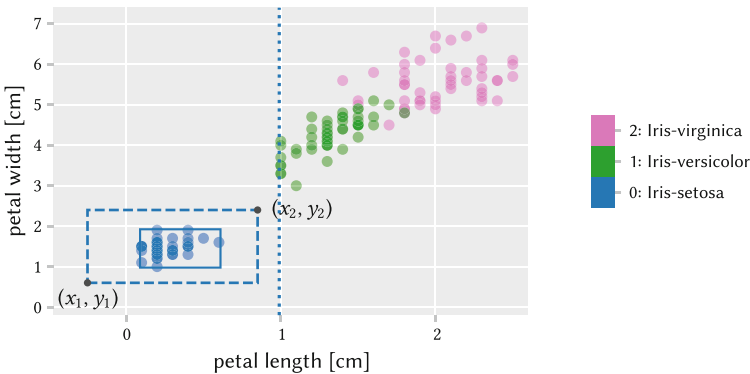


Fig. 14.5 The plot is a visualization of the extract from the iris dataset Fig. 4.7a where the only used features are the width and length of petals, and the color code indicates the class ID (the name of the species). The dashed box denotes a particular hypothesis for class 0, the solid rectangle corresponds to the most specific hypothesis for class 0, and the light-blue-colored region left of the dotted line highlights the most general hypothesis for class 0

Definition 14.1 (Decision Boundary). A decision boundary is a hyperplane that separates two classes. For a dataset with two features shown in two-dimensional space, this reduces to a line; in three-dimensional space, the line becomes a plane.

A further notion in the context of classification is the notion of the *most specific hypothesis*, which in our example is given by the smallest possible rectangle that encloses all positive examples, i.e., data belonging to class 0, and no data belonging to any other class. Taking a look at Fig. 14.5, it is easy to imagine what the most specific hypothesis for class 0 would look like (shown by the rectangle drawn with the thin solid line). It is also clear that this hypothesis is unique in the sense that there exists only one such rectangle fulfilling the requirements.

Another hypothesis type is that of the *most general hypothesis*, which for our example is the rectangle, that has the maximum possible size, contains all positive examples and no negative examples. This is a bit more difficult to visualize: for class 0 (iris setosa) it is the region that extends infinitely into positive and negative vertical direction, just misses to contain the data of class 1 on the right and stretches infinitely into negative horizontal direction, as indicated by the shaded region in the left half of the figure. The most general hypothesis needs to contain *all* data points of the respective class.

Separating class 2 from class 3 is more complicated, and the simple structure of the chosen hypothesis class, which contains all rectangular regions, is no longer sufficient. Even an inclined line would not be able to separate class 1 and 2 such that no negative examples occur. A line separating these would need to be curved or have many kinks. Obtaining such a decision boundary requires more complex algorithms, which we will derive in more details below.

We conclude this section by some thoughts on how to implement the search for a simple, rectangular decision boundary: a straightforward strategy for separating class 0 from the other two classes is to simply find the minimum and maximum of the data records of label 0 as shown below. We start by importing the obligatory modules.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
```

Here, the dataset is obtained from the scikit-learn package. Next, we extract the data to be used. For getting the two feature, we have to ensure that the columns with index 3 and 2 are the correct ones, e.g., by checking the documentation of the dataset:

```
In [2]: DS = load_iris()
X1 = DS.data[:,3]
X2 = DS.data[:,2]
y = DS.target
```

Subsequently, for finding the minimum and maximum coordinates, we only want to consider the data points with label value zero. This is achieved by creating the `mask`—an array with Boolean values:

```
In [3]: mask = (y == 0)
x0 = np.min(X1[mask])
x1 = np.max(X1[mask])
y0 = np.min(X2[mask])
y1 = np.max(X2[mask])
```

Now, we can visualize the data: we plot the rectangle and all data records where their color is chosen according to their class labels:

```
In [4]: fig, ax = plt.subplots()
ax.plot([x0, x1, x1, x0, x0], [y0, y0, y1, y1, y0])
ax.scatter(X1, X2, c=y)
```

This results then in a plot similar to Fig. 14.5 where the rectangle from the last Python cell corresponds to the one shown by the solid line in the figure.

14.4 Nearest Neighbor Classifier

The *k-nearest neighbors* (**kNN**) method is based on the experience that most datasets describe continuous phenomena. The consequence is that if two points are close to each other in the feature space, they will belong to the same class and therefore have the same label: the closer the points, the more similar their class labels should be. Hence, classifying a new data point comes down to finding the one (or several) neighboring points in feature space, so that, based on their labels, the label of the new data point can be determined.

In order to being able to quantify what the distance between points is, we start now by introducing some mathematical distance measures. Then, the 1-nearest neighbor classifier is introduced, which has an interesting algorithmic correspondence to a method of computational geometry and then generalized the concept toward using n nearest neighbors and multi-class classification.

14.4.1 Distance Measures

The most commonly used way of measuring distances is the *EUCLIDEAN distance*, which is the measure that we also use in our daily life. It is defined as

Definition 14.2 (EUCLIDEAN distance). Given two points that are represented by the vectors $\mathbf{x}_1$ and $\mathbf{x}_2 \in \mathbb{R}^n$, the Euclidean distance D_2 between the points is defined as

$$D_2(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_2 - \mathbf{x}_1\| \in \mathbb{R} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \quad (14.9)$$

The EUCLIDEAN distance can be easily generalized to arbitrary dimensions, e.g., in n -dimensional space the distance of two points $\mathbf{x} = (x_1, \dots, x_n)$ and $\mathbf{y} = (y_1, \dots, y_n)$ is

$$D_n(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}. \quad (14.10)$$

This measure is symmetric, which implies that the distance between point A and point B is the same as the distance between point B and A —as expected. The EUCLIDEAN distance is the most commonly used measure for the **kNN** method. It is not scale invariant: all features (i.e., dimensions) are treated equally, regardless the units. This might make scaling or normalization necessary to avoid that small relative changes in one feature lead to large differences in distance.

A generic distance measure is the **MINKOWSKI distance**, which is also called *p-norm*. It is defined as

Definition 14.3. **MINKOWSKI distance** Given are two points, represented by the vectors $\mathbf{x}_1$ and $\mathbf{x}_2 \in \mathbb{R}^n$. Then, the MINKOWSKI distance is defined as

$$D(\mathbf{x}, \mathbf{y}) = \sqrt[p]{\sum_{i=1}^n |x_i - y_i|^p} \quad (14.11)$$

There, p is a parameter that defines the type of measure:

- $p=2$ results in the EUCLIDEAN distance
- $p=1$: gives the *Manhattan* distance, which adds up the absolute values of the differences along each axis
- $p=0$: recovers the **HAMMING** distance, which effectively counts the number of features
- for $p \rightarrow \infty$ the distance approaches $\max_i |x_i - y_i|$, which acts as a logical OR

In the following, we will mainly use the EUCLIDEAN distance, but we suggest to keep in mind that this might not be optimal as soon as variables have different dimensions or orders of magnitudes.

14.4.2 The (First- or One-) Nearest Neighbor Method

The most intuitive approach is to search only for the nearest neighbor of a given point and then to assign that point the label of the nearest point. This is shown for two points in Fig. 14.6.

More formally, we can state this approach as follows: Assume a given training dataset consisting of features and labels. Then, for a new point P in feature space $\hat{x}$ with unknown label $\hat{y}$ we search for the most similar point of the training data, x_i , and assign its label to the new point, $\hat{y} := y_i$. “Most similar” is defined here as having the smallest EUCLIDEAN distance. This can be written as the following Python code:

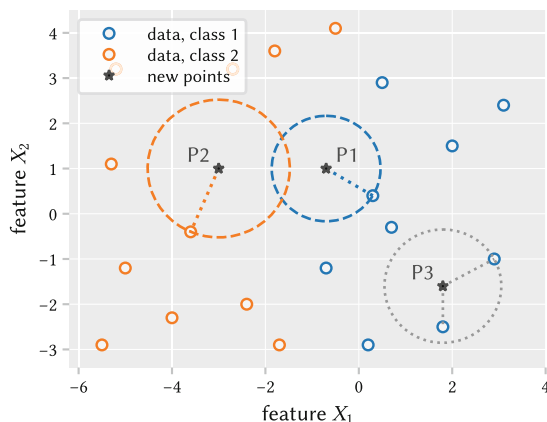
```
In [1]: import numpy as np

# X contains the training data: 2 features in columns and 4 records
# in rows, y contains the 4 labels (0 or 1) of the training data.
X = np.array([[ 0.19, -1.63], [1.62, 3.02],
              [-2.54, 0.61], [-1.29, -0.26]])
y = np.array([0, 0, 1, 1], dtype=int)

# Define the coordinates ("features") of a point for which the
# label is wanted:
x_new = np.array([2, 0])
```

where we defined the training data and the unknown point. Next, we compute the Euclidean distance of P to all points in X :

Fig. 14.6 Visualization of the nearest neighbor method: the training data is given by the round markers where the two colors denote two different labels. For the two unlabeled points P1 and P2, the closest point of the training data determines the label, i.e., P1 belongs to class 1 and P2 belongs to class 2. Additionally the two nearest neighbors are shown for a point P3 (cf. the explanations in Sect. 14.4.3)



```
In [2]: dist = []
        for x_train in X:
            # each row of X contains two coordinates ("features")
            dist.append(np.sqrt((x_train[0] - x_new[0]) ** 2 +
                               (x_train[1] - x_new[1]) ** 2))
```

Then, we can find the nearest point and print its label:

```
In [3]: # Find index of smallest element (=nearest) using
        idx = np.argmin(dist)

        # ... and show the predicted class label
        y[idx]
```

```
Out [3]: 0
```

There, a particularly useful function is `numpy`'s `argmin` function: it is applied to a `numpy` array and returns the index of the element in the array that has the lowest value.

Computation of Distance between Sets of Points

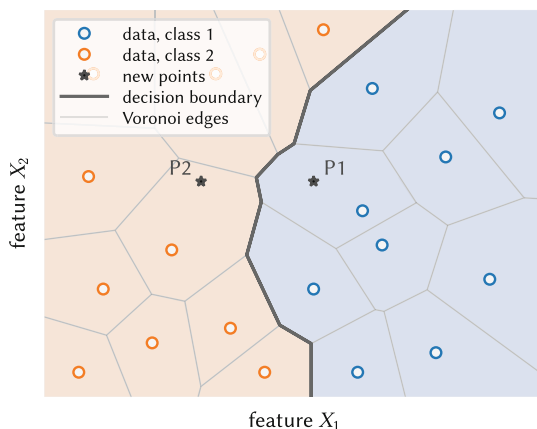
In general, computing the distance between two sets of points can quickly become a computationally expensive procedure. In such cases, using the efficient `scipy` function `distance_matrix` is recommended.

Relation to Voronoi tessellation There is a related mathematical problem that helps to understand how the nearest neighbor algorithm is working under the hood: it is concerned with the question how the areas around a set of given “seed points” can be partitioned such that all points within each area are closer to the respective seed point than to any other seed points. The solution can be found by creating a *Voronoi diagram* or *Voronoi tessellation*. Figure 14.7 shows a Voronoi tessellation of the data from Fig. 14.6, where additionally each cell was colored according to the label of the training data. More details on the Voronoi tessellation and related geometrical concepts and algorithms can be found in [1, 3].

Why do the boundaries between each cells consist of straight lines? This is due to using the EUCLIDEAN metric as distance measure; using, e.g., the Manhattan distance would result in nonlinear shapes. Voronoi diagrams can be used in multidimensional space and, in the context of materials science, have been used to generate three-dimensional grain structures as input for simulations.¹

¹ As most of the real grain microstructures result from a growth process, they obey often a lognormal distribution function. Therefore, many algorithms for creating artificial grain structures either postprocess the Voronoi structures or more recently use other algorithms.

Fig. 14.7 Using a “Voronoi tessellation” for partitioning the plane into areas, where each of them contains one of the given training data points (“seeds”, the shown markers). The areas are chosen such that the points belonging to them are closer to the respective seed than to any other seed. Additionally, the decision boundary is shown as the polygon that separates regions of different colors (classes)



With view on ML, it is quite remarkable that this relatively simple algorithm is able to generate a complex decision boundary, which takes the shape of a polygon. However, the flexibility of the method comes with the drawback that it is prone to overfitting.

14.4.3 The General k -Nearest Neighbor Method

The *k-nearest neighbors* (kNN) method is a straightforward extension of the above introduced method. kNN considers not only the direct neighbor point of an unknown point but rather a certain *neighborhood*. This is done by using the k neighbors where the label of the unknown point is inferred from the labels of the k neighbor data points (cf. P3 in Fig. 14.6). For determining the unknown label, different strategies exist. For example, *voting* assigns the label of the most frequent class among the k points; if there is a tie, then a class could be assigned by random choice. This, however, is the reason why often odd numbers for k are preferred.

Problems may occur in case of *class imbalance*, i.e., where one class has more data than the other. Then, voting is always biased toward the class with the most samples. One remedy is to weight the k points according to the inverse of the distance between the training data and the to-be-classified point. This method is known as *weighted nearest neighbor classifier*.

The hyperparameter k “regularizes” the classification as it reduces noise and fluctuations and avoids a strong dependency on only a single point. While values $k > 1$ do not smooth out the boundaries between classes, it still prevents that the decision boundary gets too strongly adapted to the training data. It thereby is useful for avoiding overfitting. Nonetheless, the decision boundary still will stay non-smooth and will have a polygonal shape with a number of kinks. The geometry is solely determined by the training data and by the used distance metric. Figure 14.8 shows the decision boundaries for four different values of k . There, it can be seen

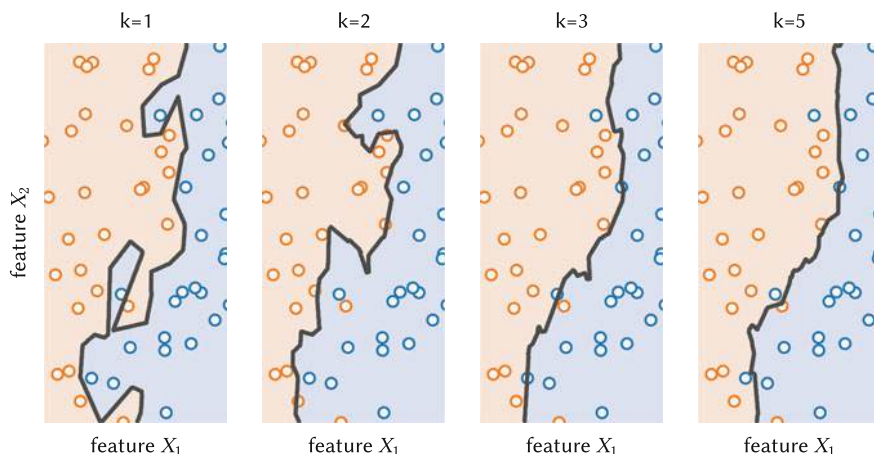


Fig. 14.8 Decision boundary of the k NN method for different values of k . The value of $k = 3$ already removes “kinks” and protuberances that are quite specific for this particular training dataset quite effectively

that for $k = 1$ and $k = 2$ the shape of the decision boundary is strongly influenced by “extreme points”, and it closely resembles details of the data structure. For $k = 3$, the curve is not bending around individual instances any more.

Determining which is a suitable value for the hyperparameter k can be done manually by trial and error (as a rule of thumb, using $k = 3 \dots 5$ often gives good results but is also dependent on the data) if one does not perform a full *hyperparameter optimization* study. If k is manually varied, it is often advisable to record the prediction accuracy and take the value of k for which the training error is minimal before it eventually increases again.

All k NN classifiers are so-called *lazy learners* because there is no model that is parameterized during the training process (e.g., recall that in a rule-based model, the free parameters are identified; in a neural network even millions of parameters, the weights, are identified during training). Thus, the key information of the training data is not encoded at training time. Instead, the whole training process of a k NN classifier consists of simply storing all training data and searching for the most similar instances at prediction time. This is also called *instance-based learning* (cf. Sect. 11.2.2) and has a few noteworthy side effects:

- the training time is negligible as no computation-intensive training takes place;
- as the number of training data records increase, predictions will become more and more costly as larger and larger datasets need to be searched
- deploying the ML model effectively means to deploy all training instances, which can result in huge datasets
- it is very easy to add new training data (simply store them), no retraining of the model is needed

kNN can also be used for regression problems. Then the voting is replaced by averaging.

14.4.4 Multi-class Classification

Multiclass classification can be done in analogy to what has been introduced above and does not pose any new methodological problems. Figure 14.9 shows the decision boundary for a classification of three classes. Also shown is the misclassification error (i.e., the total number of wrongly classified data) as a function of the nearest neighbor parameter k . The error was computed both for the training dataset and the validation dataset after splitting the whole dataset into 60% training and 40% validation data. There it can be observed that a low training error (there are even no misclassifications for $k = 0$!) does not imply a low testing error. Rather, the maximum of the testing error and the minimum of the training error are a clear indicator of overfitting. With increasing k , the training error increases while the testing error decreases to a minimum error of seven misclassifications.

Words of advice



Directly comparing the values of the *absolute* classification error for the training and validation dataset is not reasonable as the whole dataset was split into validation and training datasets of different sizes!

Without cross-validation, it is not easy to choose a suitable value of k ; any value between 9 and 15 seems reasonable. For example, for $k \geq 9$, we see a substantial decrease in the testing error. $k \leq 10$ means that each point is determined by a

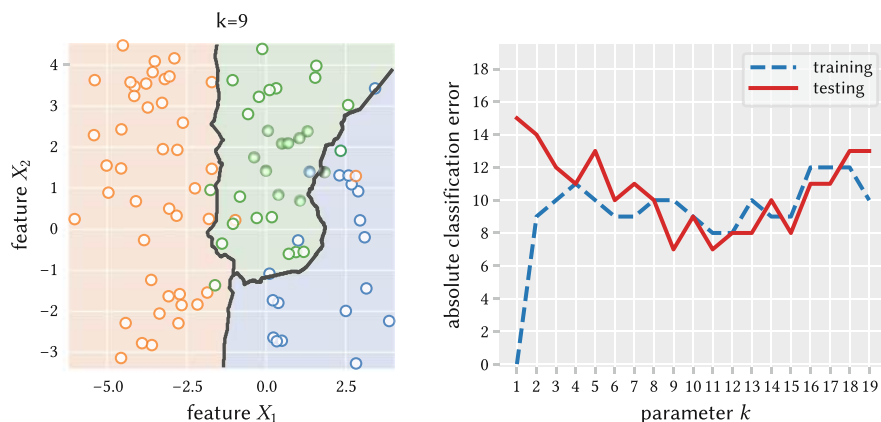


Fig. 14.9 Decision boundary of the kNN method for data from three different classes for a particular value of k along with the number of misclassified instances as a function of k for the training and validation dataset

neighborhood of 10 other points. This already corresponds to a relatively large area of the shown two-dimensional plane. Given that the spatial distribution of points is roughly homogeneous, that means that each predicted point is associated with a region that is, in this example, approximately a tenth of the shown area. If the physical phenomena investigated by this method exhibit a wave length that is much below that size, then one cannot expect to get a small error. Thus, choosing the hyperparameter $k = 7$ or $k = 9$ might be a reasonable choice. Example 14.1 presents an example, where the knowledge about the underlying problem and its data also turns out to be helpful for determining hyperparameters.

Example 14.1 kNN and a “Microstructural Length Scale”



Imagine you investigate a lamellar microstructure as shown in Fig. 14.10. You also might have acquired information for the $\alpha_2 + \gamma$ region about, e.g., the chemical composition, represented then in form of two classes stating if a point is part of the matrix or not. The chemical composition was measured at the points shown in Fig. 14.11 where the two colors denote the two classes (i.e., different chemical compositions).

Taking a look at the left plot we observe that there the whole dataset is associated with different length scales: The first one results from the random sampling and is the mean distance between sampling points. The second length scale is a microstructural length scale and is the spacing between two “bands.”

Choosing a suitable value for the number of nearest neighbors, k , is influenced by the microstructural length scale, e.g., a small value of $k = 5$ nicely identifies the lamellar structure, whereas a high value of k does not classify the structural details properly—it even predicts most of the blue instances as “orange class” and vice versa. In the left panel, the circle around the red astericks shows the area within which the five nearest neighbors are located, the middle panel shows the circle for the 47 nearest neighbors. The small circle has the same diameter as one of the bands, whereas the large circle stretches across three bands. This is the reason why $k = 5$ coincides with the minimal training error while $k = 47$ coincides with the maximum error—which is even higher than the baseline of 0.5!

In summary, this example shows that it can be quite important to relate the hyperparameter k to the physical phenomena responsible for the dataset – or that considering our physical background knowledge can help to avoid a costly hyperparameter optimization.

Fig. 14.10 Example of a lamellar microstructure found in a TiAl-based alloy that was reinforced with Carbide particles (taken from [2], Creative Commons CC BY 4.0, <https://creativecommons.org/licenses/by/4.0>)

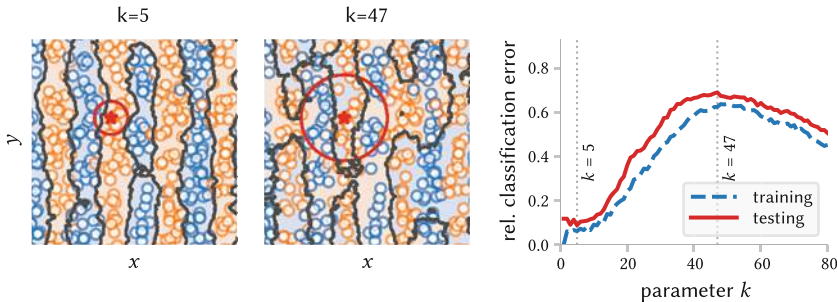
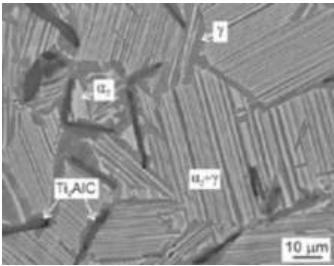


Fig. 14.11 Classification of measurement points of a lamellar microstructure. Shown are the training data and classification results of the **kNN** method using two different values of k ; the right panel shows the relative error

14.4.5 Implementation

A Python code that predicts class labels based on given training data is shown in Python Listing 14.1. It uses an efficient SciPy function for computing a matrix of the distances of all possible combinations. Additionally, numpy functions are used for sorting the instances according to distances and for the subsequent voting. Further explanations are given by the comments in the listing. Note, that there is no training function, as the **kNN** method is an instance-based learner.

```

1 from numpy import ndarray, empty, argsort, argmax, bincount
2 from scipy.spatial import distance_matrix
3
4 def predict(X_train: ndarray, y_train: ndarray, X: ndarray, k=1) -> ndarray:
5     """
6     Returns the k-nearest neighbors for all instances in `X` and given training data,
7     X and X_train are 2D numpy arrays, y_train is a 1D array.
8     """
9     # row i of dist contains the distances for vector X[i] to every
10    # other vector of the training dataset
11    dist = distance_matrix(X, X_train)
12
13    # Go through all rows of dist, i.e. the distance of vector X[i] to all training
14    # points: get the `k` closest points and check which class has the most entries
15    y = empty(X.shape[0])
16    for i, d in enumerate(dist):
17
18        # get the indices "that would sort the 1D array `d`" from nearest to farthest
19        sorting_indices = argsort(d)
20        kn_y = y_train[sorting_indices][:k].ravel()
21
22        # `bincount` returns the number of occurrences of each value of the array --
23        # therefore, the position of the maximum IS the class with the maximum count
24        count = bincount(kn_y)
25        y[i] = argmax(count)
26
27    return y

```

Python Listing 14.1 An implementation of the kNN algorithm. In case that there is a tie during voting (here implemented using the `argmax` function, which returns the index of the element with the maximum value), then only the first class with the highest count contained in `count` will be used

Things to Remember about the kNN classifier ...



- The special case of $k = 1$ is closely related to the Voronoi diagram.
- Complexity: It is a rather simple to understand methods that can be easily implemented
- Accuracy: It is a very accurate model that can predict complex decision boundaries and depends on only few hyperparameter (the number of neighbors k and the distance metric)
- Performance: quite slow predictions due to "lazy learning"; increasingly slow with growing dataset, only small dependency on value of k
- Multiple classes: kNN can easily be used for two-class and multi-class classification
- Memory: Requires large storage as all training data are stored

14.5 Gaussian Naive Bayes Model

The method called *Gaussian Naive Bayes* (GNB) is one of the well-established classification methods that has been around for several decades. It is a *probabilistic model* and belongs to the class of the so-called *naive Bayes methods*.

14.5.1 The Class of Naive Bayes Models

The naive Bayes methods all have in common that they are based on BAYES' theorem, which, for n features X_i and given a specific label Y reads:

$$P(Y | X_1, \dots, X_n) = \frac{P(Y)P(X_1, \dots, X_n | Y)}{P(X_1, \dots, X_n)} . \quad (14.12)$$

Words of advice



The naive Bayes models are probabilistic models, and therefore the “features” should be rather understood as random variables, and Eq. (14.12) should be written more explicitly as

$$\begin{aligned} P(Y = y | X_1 = x_1, \dots, X_n = x_n) \\ = \frac{P(Y = y)P(X_1 = x_1, \dots, X_n = x_n | Y = y)}{P(X_1 = x_1, \dots, X_n = x_n)} . \end{aligned}$$

There, it is more obvious that, e.g., X_1 is the random variable and x_1 is the particular value. For different data records, the random variable stays the same, and only the particular value changes. However, for brevity, we will use the above short notations.

The key idea of the naive Bayes methods is to make the “naive” assumption, that the X_i and X_j of every possible pair of input variables are conditionally independent for a given label Y , which translates to

$$P(X_i | Y, X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n) = P(X_i | Y) \quad (14.13)$$

an which generalizes to the following equation:

$$P(X_1, \dots, X_n | y) = \prod_{i=1}^n P(X_i | Y) . \quad (14.14)$$

Upon insertion of this identity into Eq. (14.12), it follows:

$$P(Y | X_1, \dots, X_n) = \frac{P(Y) \prod_{i=1}^n P(X_i | Y)}{P(X_1, \dots, X_n)} . \quad (14.15)$$

Finally, upon realizing that $P(X_1, \dots, X_n)$ is constant we find that the probability to obtain the label Y for the given features is proportional to the nominator of Eq. 14.12:

$$P(Y | X_1, \dots, X_n) \propto P(Y) \prod_{i=1}^n P(X_i | Y) . \quad (14.16)$$

Now, we have a probability for every possibly class label, given a set of feature values. The most likely class label is obtained as the one that gives the maximum probability,

$$Y^{\text{pred}} = \arg \max_Y \left(P(Y) \prod_{i=1}^n P(X_i | Y) \right) . \quad (14.17)$$

The question left to be answered is: How can $P(Y)$ and $P(X_i | Y)$ be obtained? The first probability is, for all naive Bayes models, straight forward: $P(Y)$ is simply the relative frequency of class Y w.r.t. the training dataset $\mathcal{D}^{\text{train}}$:

$$P(Y = y) = \frac{\sum_{i=1}^n [[y_i = y]]}{n} , \quad (14.18)$$

where $[[y_i = y]]$ is 1 only if $y_i = y$ and 0 otherwise. Effectively, this gives the number of times that label y occurs in the training dataset.

For a naive Bayes model, the factor $P(X_i | Y)$ can be estimated in a similar way, whereas for a *Gaussian* naive Bayes model the formulation is shown in the following.

14.5.2 Maximum-Likelihood Estimation for the Gaussian Naive Bayes Model

The particular formulation of $P(X_i | Y)$ is, where the various naive Bayes methods differ; for the *Gaussian* Naive Bayes, the likelihood of every feature is assumed to follow a (Gaussian) normal distribution (cf. Sect. 10.7). To calculate the conditional probability $P(X_i | Y)$ it is important to realize that in this expression, Y denotes a particular class label, $Y = y$. Therefore, the dataset needs to be split into sub-datasets, each of which contains only data records that have the same label. For example., for label j , we write this as

$$\mathcal{D}_{y=j} = \left\{ (x_1, \dots, x_n, y) \mid (x_1, \dots, x_n, y) \in \mathcal{D}^{\text{train}} \text{ and } y = j \right\} \quad (14.19)$$

For example, for feature number i and label j , it is

$$P(X_i | Y = y) = \mathcal{N} \left((x_i, y) \in \mathcal{D}_{y=j}; \mu_y, \sigma_y \right) . \quad (14.20)$$

Note that the normal distribution is fitted to all data records that have the same class j . This needs to be done for all features and for all class labels. Also note that the conditional independence of the features implies that for each variable a *univariate* normal distribution is fitted and not a multivariate distribution to all features at the same time.

14.5.3 Algorithm and Python Implementation

As a preparation for a Python implementation, we summarize the above formulations in the steps shown in Algorithm 14.1. In particular, we emphasize the difference of the training and the prediction by using explicit superscripts.

Algorithm 14.1: Gaussian naive Bayes Classification



Given is a training dataset $\mathcal{D}^{\text{train}}$ consisting of n features, m records and a single target variable that contains the class labels in form of integer numbers.

1. Split the dataset into disjunct datasets, $\mathcal{D}_{y=k}^{\text{train}}$, such that the class label within each of them are the same.
2. Find the mean values and the standard deviations by fitting a normal distributions to the training data—individually, for each class label and for each feature. For example, for the records taken from $\mathcal{D}_{y=1}^{\text{train}}$, this results in the mean values

$$\mu_{X_1, y=1}, \dots, \mu_{X_n, y=1}.$$

3. Compute $P(Y = y)$ for all class label. E.g., for label k this is

$$P(Y = k) = \frac{|\mathcal{D}_{y=k}^{\text{train}}|}{|\mathcal{D}^{\text{train}}|}$$

where $|A|$ denotes the number of elements in set A .

4. Choose a new input record for which the label is to be predicted is $(X_1^{\text{pred}}, \dots, X_n^{\text{pred}})$.
5. Compute the conditional probabilities $P(X_i | Y)$ for each feature, separately for each dataset $\mathcal{D}_{y=k}^{\text{pred}}$. E.g., for class number 1 this reads

$$\begin{aligned} P(X_1^{\text{pred}} | Y = 1) &= \mathcal{N}(X_1^{\text{pred}}; \mu_{X_1, y=1}, \sigma_{X_1, y=1}) \\ &\vdots \\ P(X_n^{\text{pred}} | Y = 1) &= \mathcal{N}(X_n^{\text{pred}}; \mu_{X_n, y=1}, \sigma_{X_n, y=1}); \end{aligned}$$

where the normal distributions $\mathcal{N}$ are evaluated at the records of the testing dataset X_i^{pred} ($i \in [1, n]$).

6. Compute the “score” for each possible class (the term inside the parentheses in Eq. (14.17)), e.g., for class $Y = 1$:

$$\text{SCORE}(Y = 1) = P(Y = 1) \cdot \left(P(X_1^{\text{pred}} | Y = 1) \cdot \dots \cdot P(X_n^{\text{pred}} | Y = 1) \right)$$

7. Compare the scores of the record, for which the label is to be predicted, for all classes and choose the one with the highest score as the most likely one.

In principle, [GNB](#) is a simple method. However, the devil is in the detail, which is here, mainly the split into sub-datasets and when to use training or the to-be-predicted data. For example, we also realize that the training process is finished after Step 2. Step 3 is then a preparation for the following prediction. The advantage of this detailed algorithm is that we can very easily transfer it now into Python code.

We write this for the special case of two classes and avoid `for` loops and advanced vectorization in favor for readability.

```
In [1]: import numpy as np
        from scipy.stats import norm          # Gaussian distribution
        from numpy.random import multivariate_normal
```

Here, `norm` is required for computing the *probability density function (PDF)*, `multivariate_normal` for sampling from a bivariate Gaussian distribution for creating training data:

```
In [2]: X_class0 = multivariate_normal(mean=(-1, 0), cov=[[1, 0], [0,2]], size=30)
        X_class1 = multivariate_normal(mean=(2, 3), cov=[[1, 0], [0,1]], size=20)
        Y_class0 = np.zeros_like(X_class0)
        Y_class1 = np.ones_like(X_class1)
```

Next, we compute the mean values for all classes and for all features. As additional preparation, we also already computed the marginal probabilities of both classes.

```
In [3]: means_class0 = [np.mean(X_class0[:,0]), np.mean(X_class0[:,1])]
        means_class1 = [np.mean(X_class1[:,0]), np.mean(X_class1[:,1])]

        stddev_class0 = [np.std(X_class0[:,0]), np.std(X_class0[:,1])]
        stddev_class1 = [np.std(X_class1[:,0]), np.std(X_class1[:,1])]

        P_Y_for_class_0 = np.size(Y_class0) / np.size(Y)
        P_Y_for_class_1 = np.size(Y_class1) / np.size(Y)
```

Now, the model is trained, and we choose a feature vector for which we seek the class label.

```
In [4]: X_pred = np.array([2, 2])
```

`X_pred` is just a one-dimensional array (if the code should be vectorized, this needs to be changed). Subsequently, we create numpy “Gaussian random variables” that can be used to evaluate the PDF at the “location” `X_pred`. (cf. Sect. 10.7 for more details).

```
In [5]: # class 0
rv0 = norm(loc=means_class0[0], scale=stddev_class0[0]) # first feature
rv1 = norm(loc=means_class0[1], scale=stddev_class0[1]) # second feature
P_X1_given_class0 = rv0.pdf(X_pred[0])
P_X2_given_class0 = rv1.pdf(X_pred[1])

# class 1
rv0 = norm(loc=means_class1[0], scale=stddev_class1[0]) # first feature
rv1 = norm(loc=means_class1[1], scale=stddev_class1[1]) # second feature
P_X1_given_class1 = rv0.pdf(X_pred[0])
P_X2_given_class1 = rv1.pdf(X_pred[1])
```

Finally, we are able to compute the scores for the point w.r.t. both classes

```
In [6]: P_X_given_class0 = P_X1_given_class0 * P_X2_given_class0
P_X_given_class1 = P_X1_given_class1 * P_X2_given_class1
score_class0_given_X1_X2 = P_X_given_class0 * P_Y_for_class_0
score_class1_given_X1_X2 = P_X_given_class1 * P_Y_for_class_1

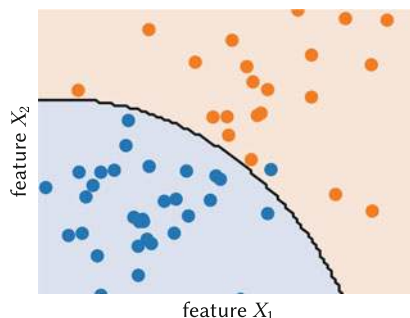
if score_class0_given_X1_X2 > score_class1_given_X1_X2:
    Y_pred = 0
else:
    Y_pred = 1
```

where `Y_pred` contains the predicted class label of `X_pred`.

As always, the decision boundary can be visualized by making predictions at equidistant points in the two-dimensional feature space, which are then used together with a contour plot, as shown in (Fig. 14.12).

We conclude the GNB model with two short comments: To avoid small numbers and numerical underflows for the values of the conditional probabilities (they tend to be rather small), many implementations use logarithmic probabilities. If there are

Fig. 14.12 The Gaussian naive Bayesian model: decision boundary and training dataset



not data at all for a specific class, then one might encounter a “division by zero” error; therefore, in a production code, one should guard against this case.

Things to Remember about the Gaussian naive Bayes model



- The GNB model is able to learn nonlinear decision boundaries.
 - Strictly speaking, the data need to be normally distributed; however, in practical situations, this is not strictly required.
 - The implementation is relatively simple, and the method is fast because no iterations for finding parameters are required.
 - As long as the conditional independence is fulfilled, it is quite a reliable method.
 - By comparison with other methods, larger amount of data are required for achieving good results.
-

14.6 Support Vector Machines

The main issue with **kNN** is the computational cost for comparing all n points to all other points. With an support vector machine, this approach changes entirely. The linear *support vector machine* (**SVM**) method is one of the older methods and was created already in the 1960s by VLADIMIR VAPNIK. The extension is the nonlinear **SVM**, which is a very powerful classification method. Both will be introduced subsequently.

Support Vector Classification



Because **SVMs** is used both for regression and classification, sometimes a **SVM** in the context of classification is in the literature also called *support vector classifier*, abbreviated as SVC while the variant used for regression is called *support vector regressor*, abbreviated as SVR.

14.6.1 Linear Support Vector Machine

Assume there are two classes of linearly separable data. There exists no unique solution for choosing, e.g., the inclination of a straight line for separating the two groups of points, compare the left panel of Fig. 14.13. This is exactly, where **SVMs** come into play: there, the idea is to maximize the distance from the line to the nearest data points, making the problem of dividing linearly separable data by a straight line uniquely solvable. This is shown in the right panel of Fig. 14.13. Note, that the solid line (the decision boundary) is located exactly centered between the

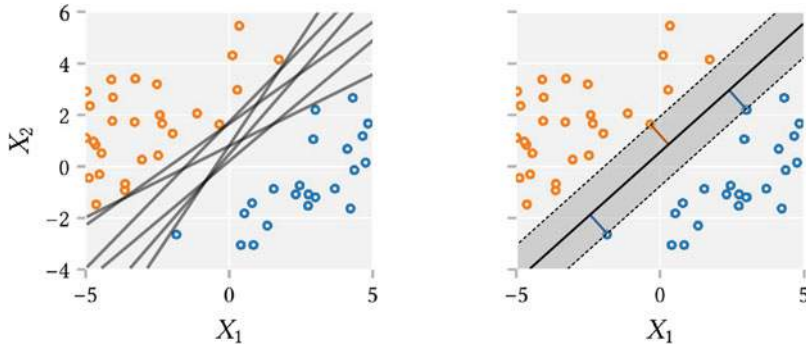


Fig. 14.13 Linearly separable dataset: the left panel shows possible linear decision boundaries; the right panel shows a decision boundary (solid black line) that maximizes the distance to the nearest data (dashed lines). The “support vectors” are indicated as the three short (two blue and one orange) lines, perpendicular to the decision boundary

two dashed lines. This concept can be generalized to higher-dimensional data. Then, the linear decision boundary becomes a (hyper-)surface; other than that, nothing changes. The short lines from the decision boundary to the three data points are the *support vectors* and, by construction, have the same lengths. This distance is called *margin*, and the hyperplane that maximizes the margin is called *optimal hyperplane*. There is a whole group of methods that maximize the distance from data to the decision boundary: these are called *maximum margin estimators*.

Hints on Numerical Implementation

The numerical algorithm is based on an optimization strategy: there, the goal is that the distance to the nearest points of the two classes should both be d , and it should be maximized. Typically, for classification problems, the *hinge loss* is used, which is defined as

$$\mathcal{L}(\mathbf{x}^{\text{train}}, y^{\text{train}}) = \begin{cases} 0 & \text{if } y^{\text{train}} \cdot y^{\text{pred}} \geq 1. \\ 1 - y^{\text{train}} \cdot y^{\text{pred}} & \text{otherwise.} \end{cases} \quad (14.21)$$

where the two class labels take the values -1 and $+1$. As a consequence, $y^{\text{train}} = y^{\text{pred}}$ implies a zero loss value while $y^{\text{train}} \neq y^{\text{pred}}$ results in a loss value of 2. The cost function will consist of two terms: (i) the mean hinge loss $\mathcal{L}$ of all examples and (ii) the constraint that maximizes the width of the data-free channel, which leads to a constraint optimization problem. y^{pred} depends on the inclination and the intercept of the line in two dimensions, which are given by the parameters to be optimized. In all generality, the free parameters define the hyperplane (or in the two-dimensional case: the line). If the parameters are chosen correctly, then the boundaries of the whole margin (the dashed lines of the shaded area in Fig. 14.13)

touch the data only in a few number of points. Generally, if the dataset has n features, then a total of $n + 1$ points necessarily touch the boundaries of the “channel”.

Taking the optimal hyperplane as a decision boundary is a robust approach: even if new training data are added, it is expected, that this hyperplane is, on average, the least affected as compared to any other hyperplane. Additionally, points that are located within the convex hull of the data of any class have no effect on the decision boundary and the support vectors. This also has the side effect that SVM is not very sensitive w.r.t. outliers.

Unfortunately, it is also obvious that this method (similar to logistic regression) only works for linearly separable datasets. There are approaches that allow a small number of points in the data-free, separating region. Those act as a regularization, as the original requirement is a bit relaxed, and leads to multi-objective optimization strategies. However, the main characteristics of the dataset still have to be such that the data are mainly linearly separable. Otherwise, one has to resort to nonlinear methods, such as the *nonlinear SVMs*.

14.6.2 Nonlinear Support Vector Machine

Data that are not linearly separable can be used for supervised classification with the method of nonlinear support vector machines. The main idea is, that the dataset is mapped to a higher-dimensional space, in which there are more possibilities to find a suitable decision boundary. The left panel of Fig. 14.14 demonstrates the idea based on a one-dimensional dataset which is turned into a two-dimensional dataset by assigning all blue markers the X_2 -value of 0 and all orange markers the X_2 value of 1. In the right panel, we can see that then it becomes nearly trivial to separate the two classes. In all generality, the functions that perform such a mapping are kernel

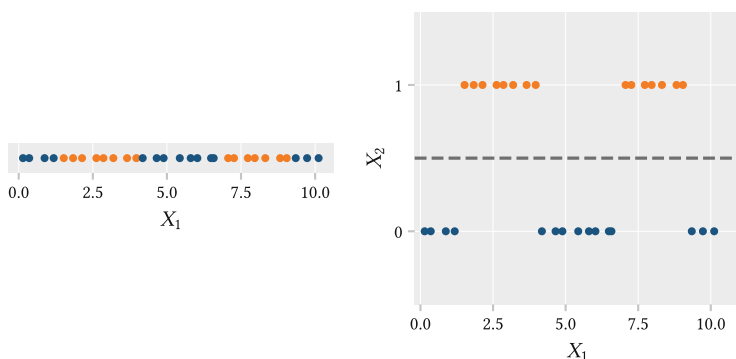
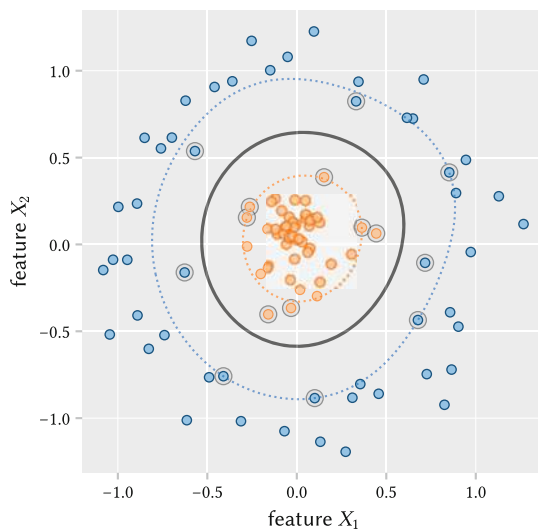


Fig. 14.14 The left panel shows a dataset with only one feature and with two different class labels, indicated by the two colors, which are clearly not linearly separable. By using the class labels to “extrude” the data and thereby to create a new feature, the two groups of data become linearly separable, indicated by the dashed line

Fig. 14.15 Support vector classification for a dataset with two features. The larger circles around the data points show the support vectors



functions, cf. Sect. 13.5.4. There, each point would be replaced by, e.g., a GAUSSIAN distribution, and weighted with the integer representing the class label. Then, each of these GAUSSIANS is superimposed, which, depending on the shape parameter of the distributions, results in a smooth function—the *kernel projection*—for which different “levels” can be easily separated using a threshold.

However, in practical situations, where datasets can have many features, this is a procedure that can become computationally very expensive. The solution is, what is known as the *kernel trick*. There, the training on the projected data can be done implicitly, without having to construct the higher-dimensional, superposition of kernel functions. This makes nonlinear SVMs not only very effective for data that are not linearly separable; the kernel trick is the reason, why this method is computationally very efficient.

There are a number of kernel functions that can be used, such as polynomials or radial basis functions. As the derivation and the implementation are more lengthy, we will not detail it here and rather suggest one of the ML libraries, such as scikit-learn, which has a very efficient implementation.

A demonstration of how the decision boundary looks using SVM for a dataset with two feature is shown in Fig. 14.15. For this, we used a radial basis function (RBF) kernel, which requires only the standard deviation σ as a parameter and is given by

$$f(X_1, X_2) = \exp\left(-\frac{\|X_1 - X_2\|^2}{2\sigma^2}\right). \quad (14.22)$$

The RBF kernel is often used as a “default” for nonlinear problems. Typical sizes of datasets for SVM are usually “mid-sized” records, e.g., up to several thousands of

data records. If the datasets are becoming much larger, say, several millions of data records, then it is usually advisable to first try to reduce the dataset by some other means or to use ML methods that are more suitable for large datasets.

14.7 Exercises

14.1. Figure 14.3 is a very simple example, and the “learning strategy” is even more simplistic. What happens to the decision boundary when more positive or negative data are added? Think about a simple approach that also would consider the negative examples and implement it in a Python code. Plot the decision boundary. How sensitive w.r.t. to outliers (=data points with extreme values) is your strategy?

14.2. Use your kNN Python implementation to numerically show that increasing the value of k does not significantly influence the computational cost for predictions.

14.3. Rewrite and optimize the code for the Gaussian naive Bayes method from Sect. 14.5.3. Proceed in the following two steps:

- (a) In a first step, use `for` loops such that you are able to apply the code to an arbitrary number of features. Also, predictions for an arbitrary number of records should be possible
- (b) Now, optimize the code for performance. For this, use vectorization to replace some of the `for` loops. Measure the performance gain as compared to task (a)

References

1. F. Aurenhammer, R. Klein, and D.-T. Lee. *Voronoi Diagrams and Delaunay Triangulations*. World Scientific, Aug. 2013. DOI <https://doi.org/10.1142/8685>.
2. J. Lapin, K. Kamysnykova, and A. Klimova. Comparative study of microstructure and mechanical properties of two tial-based alloys reinforced with carbide particles. *Molecules*, 25(15), 2020. ISSN 1420-3049. DOI <https://doi.org/10.3390/molecules25153423>. URL <https://www.mdpi.com/1420-3049/25/15/3423>.
3. C. Tóth, J. Goodman, and J. O’Rourke. *Handbook of Discrete and Computational Geometry*. Discrete mathematics and its applications. Chapman and Hall/CRC Press, 3rd edition, 2017. ISBN 9781498711395. DOI <https://doi.org/10.1201/9781315119601>.
4. I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 3 edition, 2011. ISBN 978-0-12-374856-0. URL <http://www.sciencedirect.com/science/book/9780123748560>.

“Study hard what interests you the most in the most undisciplined, irreverent and original manner possible.”

Richard Feynman (1918–1988)

American theoretical physicist

15.1 Introduction to Dimensionality Reduction

Twenty years ago, the need for dimensionality reduction was directly clear, as computer memory and storage capacities were very limited and expensive resources. Why should we still worry today about “a few dimensions more or less”?

15.1.1 Example

Let us start with a simple example to elucidate some aspects of the *dimensionality of data*:

Example 15.1 Dimensionality of Image Data



Consider the dataset DS-2 (MNIST), which consists of small images with a size of 28×28 pixel. Each of their pixel is either black or white and therefore has the value 0 or 1. How many different images can be created? There are $2^{28 \times 28}$ possible realizations. However, most of these images look mainly random and only extremely few of them contain notable structure and patterns that look like digits. Therefore, sampling from the high-dimensional space

(continued)

of *all* images is not reasonable, and we seek to construct lower-dimensional spaces that ideally contain *only* those data points that we are interested in.

Example 15.1 can also be interpreted such that an *machine learning* (ML) method that takes samples from the high-dimensional space would have a very hard time to separate the noise from the signal. The *curse of dimensionality*, introduced in Sect. 2.4, suggests the same but in a slightly different embodiment: reducing the dimensions of the feature space results in a denser data coverage, which is beneficial for the ML training. These are reasons why often, methods for dimensionality reduction are used to preprocess highdimensional data for a different ML method.

15.1.2 The extrinsic and intrinsic dimensionality

A similar problem is also known in the field of signal processing and information science. There, commonly used notions are that of the *extrinsic dimensionality* versus the *intrinsic dimensionality* [2]. The extrinsic dimensionality of a dataset is very well defined as the given number of features or variables. The intrinsic dimensionality, however, gives the number of variables that are needed to represent the dataset *after* all redundancy has been removed. As such, the reduced set of variables can be understood as a minimal representation of the data. This does not imply that the minimal representation is exactly identical to the original representation but it typically *does* imply that all of the “most relevant” aspects of the dataset are recovered. An important aspect of dimensionality reduction is to transform the original data into a new representation where (i) the intrinsic dimensionality can be easily seen and (ii) where the intrinsic dimensionality is—sometimes significantly—smaller than the extrinsic dimensionality.

15.1.3 Notations and Conventions

In the previous chapters, we always considered supervised learning methods where we needed both an set of input and output data. Here, we only require input data, and hence, we start with a brief overview over the used notations for this type of dataset. Assume that the data of interest in the dataset $\mathcal{D}$ consist of m records (e.g., measurements) and that the data itself may not be just scalar data but might either consist of vector-valued data or of data with several attributes (e.g., temperature, displacement, and stiffness could be three such attributes),

$$\mathcal{D} = \{\mathbf{x}_i\}_{i=1,\dots,m} \quad \text{with } \mathbf{x}_i = [x_{i1} \ x_{i2} \ \dots \ x_{in}] \in \mathbb{R}^n. \quad (15.1)$$

There, each of the $\mathbf{x}_i$ is considered a *row* vector and has n features or attributes. Note, that this notation requires some care (and the use of the transposition symbol) as we usually assume that vectors are column vectors. The dataset can be represented by a data matrix $\mathbf{X}$, which has n features in columns and m data records in rows. Explicitly writing all components gives:

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1- \\ -\mathbf{x}_2- \\ \vdots \\ -\mathbf{x}_m- \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}. \quad (15.2)$$

Often, we assume that the data for each feature are *centered* around the mean value, which can be achieved by subtracting the mean of the j -th column, μ_j , from each of its elements x_{ij} ,

$$\bar{x}_{ij} = x_{ij} - \mu_j. \quad (15.3)$$

Therefore, we denote by $\bar{\mathbf{X}}$ the centered data matrix where its elements are the $\bar{x}_{ij}$, the row vector $\bar{\mathbf{x}}_i$ is the centered data record, and the column vector $\bar{\mathbf{X}}_j$ is the centered feature number j .

15.2 Principal Component Analysis: Theoretical Background and Derivations

Principal component analysis (PCA) is a powerful method from the toolbox of statistical data analysis; it is one of the most well-known and well-studied methods used for *dimensionality reduction* of a dataset, i.e., for constructing a representation of the data that has an intrinsic lower dimensionality and at the same time preserves as much structure of the data as possible. The dataset itself consists of only the features X_i , no labels y_i as in supervised learning methods are used or required. The features are the coordinates of points in the feature space and can be represented in a two- or three-dimensional plot as point clouds. The coordinate system that is used for showing such data is arbitrary, and the Euclidean coordinate system is only one possible choice. However, coordinate systems and coordinate transformations play an important role in PCA, as we will see in the following.

The main idea of PCA is to find the best or most suitable coordinate system where a reduced number of directions (or basis vectors) contain as much information (or equivalently: as much structure of the data) as possible. Additionally, the dimensionality of the data can be reduced by projecting the data along these lower-dimensional, “optimal” directions, which are *embedded* into the original, high-dimensional space. In the following sections, we will see that PCA can answered many questions. Among those are the following:

- Are the input variables correlated such that they might contain redundant information?
- How can variables be decorrelated?
- Is there an alternative representation of the data in terms of “simpler” and fewer input variables?
- What does “simpler” mean in the context of [PCA](#).
- Which (possibly new) variables are *the most important ones* for representing the data such that the less important variables could be removed without losing too much of important information?

[PCA](#) is typically used for high-dimensional data. Examples of high dimensional data are images that can have millions of pixels with each pixel being a single feature, or datasets from molecular dynamics simulations containing the position of millions of atoms. [PCA](#) helps to reduce the dimensions of datasets and thereby to enable better generalization of (statistical) [ML](#) models. It helps to remove noise from images. And it ultimately helps to use computational resources more efficiently, e.g., in terms of computational time or memory requirement.

[PCA](#) does not require any previous knowledge about the dataset, which can be useful if we don't have that knowledge. Obviously, this also might be a drawback as it is not possible to include any information about the underlying physics of the problem that we might possess. As an example think of a dataset that contains values of potential and kinetic energy measured at many points within a closed system. In [PCA](#) it is not possible to enforce the law of conservation of total energy (this would be our “physical preknowledge”), and the lower-dimensional representation will very likely result in a very different total energy.

[PCA](#) is closely related to *singular value decomposition* ([SVD](#)) and is formulated in terms of eigenvectors, eigenvalues, using techniques and concepts from linear algebra. It is a *linear* method for dimensionality reductions since it relies on linear transformations of basis vectors. The mathematical details are given in [Appendix A](#), but the remainder of this section can be understood without comprehending all mathematical details.

15.2.1 A Simple Motivation from Solid Mechanics

In continuum mechanics, a related application case is to transform a coordinate system with respect to which a stress state is defined so that the coefficients of the stress tensor are simplified to the so-called *principal stresses*. Those readers who have an engineering or physics background most probably will have come across either the graphical construction method using the so-called “Mohr's circle” or directly used eigenvalues and -vectors to find the principal stress components in two- or three dimensions. Typically, the corresponding mathematical derivations of the equations for computing principal stresses are based on formulating equilibrium equations of forces: there, the vertical and horizontal force components are compared to equivalent forces acting parallel and perpendicular to an inclined plane.

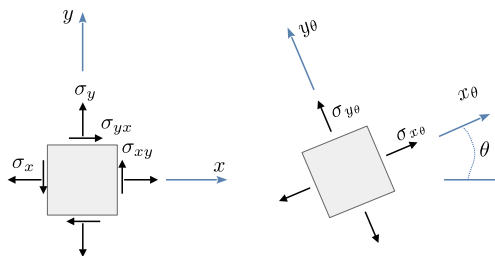


Fig. 15.1 In two dimensions, a stress state is given by two normal and one shear component. Principal stresses are obtained by rotating the reference coordinate system such that the shear stress vanishes. The stress tensor then consists of only two components, the principal stresses—a lossless “data compression” with a compression ratio of 3:2

Forces can be “converted” to stress components. Then, the inclination of that plane is found by requiring that the resulting shear stress acting in the plane vanishes—a minimization problem. From the normal and perpendicular forces then the two *principal stress components* are directly obtained, cf. Fig. 15.1.

In this case, the originally three components of the stress tensor are reduced to two different stress variables that are able to still represent the same overall stress state—this is roughly similar to what **PCA** is automatically doing for general datasets: automatically finding directions (i.e. “descriptors”) that represent features of the given dataset such that redundancy is removed and the data are “compressed” through this dimensionality reduction. For the example of principal stresses, the redundancy is the result of the *Poisson effect* through which normal strain components (and through this also stress components) along the x and y direction are coupled. Here, the set of new variables are *prescribed* and the coordinate transformation is computed. **PCA** is able to automatically find a suitable coordinate transformation (i.e., a new *basis system*). The variables w.r.t. these principal components, however, might not be interpretable in terms of any direct, physical meaning.

15.2.2 Further Examples of “Higher”-Dimensional Data Sets

The next applications and examples take a look at **PCA** from a point of view of data science and explain some of the main aspects:

1. **The Iris dataset:** One of the famous datasets in the **ML** literature is the Iris dataset. The dataset consists of 150 records each of which has four attributes characterizing different kinds of flowers; for more information, refer to the description of the DS-1 (Iris Flowers, Sect. 4.7) dataset in Chap. 14. The dataset is summarized in the six scatter plots in Fig. 15.2. **PCA** can be used to get rid of noise and redundancy, e.g., from the figure we observe, that the petal length and petal width are positively correlated and therefore already one of them might

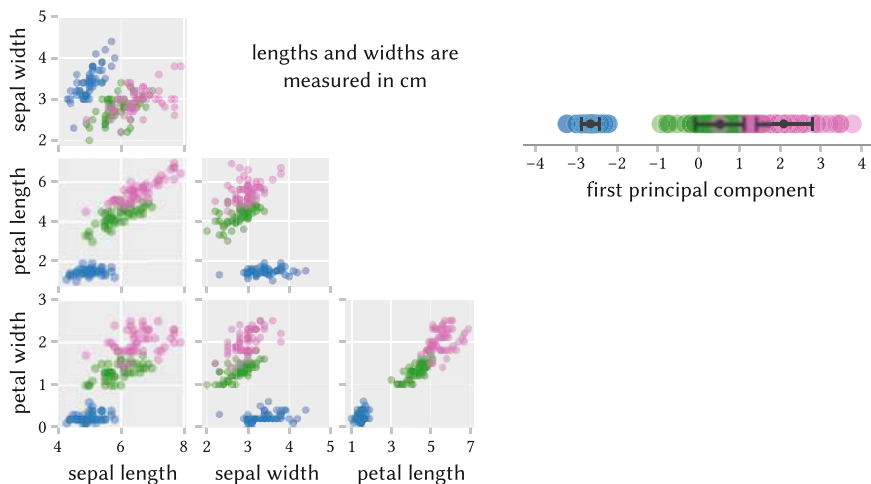


Fig. 15.2 The six scatter plots visualize the data of the four-dimensional “Iris Dataset” where the three different flowers are represented by the three colors. The right plot depicts the dataset reduced to the first principal value component. There, the “error bars” show the mean value (the dots) and the standard deviation (lines)

carry sufficient information. The right plot of Fig. 15.2 shows the data projected onto the axis of the first principal component. While some information is lost, it turns out that most of the important information, e.g., for a classification task is still retained, and only one principal component is already able to separate the three classes of different flowers fairly well. Only considering some or even only one principal components is a way of *data compression*—here to a compressed size of 25% the original dataset.

2. **Discovery of New Materials:** The power of [PCA](#) lies in reducing high-dimensional data to only few dimensions. There, the dimensionality reduction helps to escape the “curse of dimensionality” (Sect. 2.4) since sampling the reduced space typically requires significantly fewer sampling points. In computational materials science and theoretical chemistry, this has been exploited for the discovery of new materials with enhanced properties (see e.g. [5, 13]).
3. **Image Compression:** A different type of dimensionality reduction can be observed in the context of image data: an image with, e.g., $32 \times 32 = 1024$ pixel can be represented as a vector with 1024 entries by placing the values of all pixels on top of each other. This vector is thus defined in a 1024-dimensional space. [PCA](#) helps to represent these vectors in an alternative space where we can tune the accuracy (or, inversely, the level of compression) by including more or less dimensions into the new representation. [PCA](#) allows to reduce images to a few features, which then can be used for as input for other data mining or prediction methods. Examples include characterization of microstructures from

microscopy or simulations or the distribution of stress data from two-dimensional *finite element method* (FEM) simulations as, e.g., in [19].

15.2.3 The Algorithm of Principal Component Analysis in Brief

We will now summarize the method of PCA in a way that requires only basic knowledge of linear algebra and vector/matrix calculus. All proofs and further discussions can be found after this section.

The main idea of PCA is to find a number of k new coordinate axes or basis vectors that maximize the variance of the data. There, k is usually smaller than the original dimension of the dataset. Why is it important to maximize the variance of data in the new coordinate system? Imagine points on a single coordinate axis: the smaller the variance of the data, the closer the points get to each other (on average), which ultimately means that all points would coincide for a variance value of zero. The variance is a measure for the variability of the data, and the more spread out the data is the more “distinguishable” are details of the distribution and its structure. Therefore, maximizing the variance can be interpreted as maximizing the amount of contained information.

A two-dimensional example is shown in Fig. 15.3. There, the first step is to center the dataset such that the mean coincides with the origin of the coordinate system. Subsequently, the first direction of a new coordinate axis for which the variance is maximized is obtained. This is the first principal component or basis vector of the principal space, which here is one-dimensional. The data are then projected onto this direction. If the principal subspace consists of multiple components, then a new direction that is orthogonal to the previously direction is computed. The direction should not just be orthogonal to all previous axes, it also should again maximize the remaining variance of the data. This procedure is then repeated for as many times as there should be *principal components* (PCs).

In an additional postprocessing step, the projected data from the PC space can be transformed back to the real space (e) and (f) in Fig. 15.3, which also gives a first hint about how PCA can remove noise.

It will turn out that the principal directions are nothing but the eigenvectors of the data covariance matrix, which is a surprising and at the same time also very convenient fact. Eigenvectors and -values are typically sorted according to the magnitude of the eigenvalues such that the first eigenvalue corresponds to the largest possible variance of the data along one direction—the direction of the first eigenvector. The PCA algorithm can be summarized as shown in Algorithm 15.1.

15.2.4 Preparation: Orthogonal Projection of Data onto a Line

As a preparatory step for the derivation of the PCA method, we now derive the mathematical description of projecting the data $\mathcal{D}$ onto a line and investigate the consequences on the variance of the projected data. We again assume that the dataset

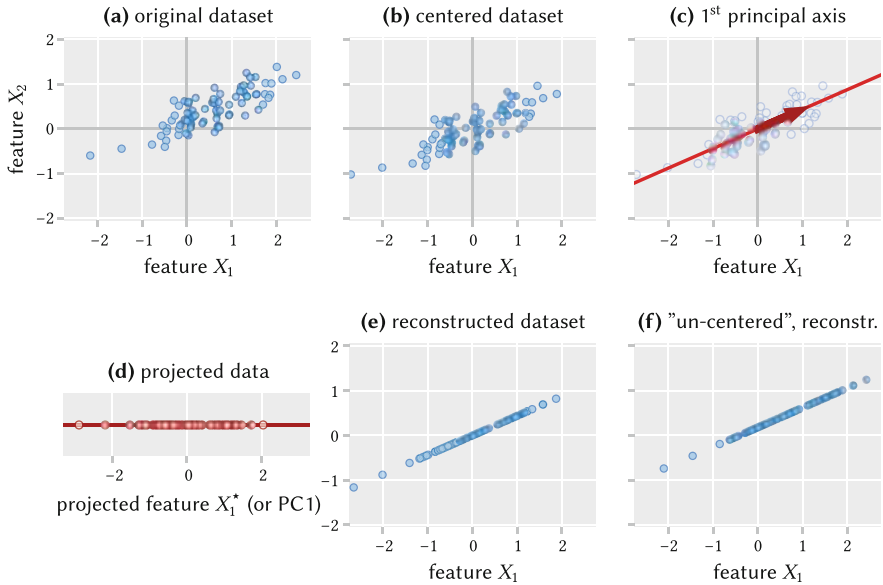


Fig. 15.3 PCA for a two-dimensional dataset (a) where X_1 and X_2 are correlated: after centering the original dataset (b), the first principal component is obtained ((c), vector shown in red). The data is then projected onto this direction (d). As a “postprocessing” step, the data from the “principal space” can be reconstructed in the real space (e) where the centering needs to be reverted (f)

is represented by the centered data matrix $\bar{\mathbf{X}}$ with features in columns and each row being a data record or sample. In this section, we are only concerned with the mathematical derivations for a single point. We therefore use as a abbreviation $\mathbf{x} := \bar{\mathbf{x}}_i$ in this section, which denotes one of the rows of the centered data matrix $\bar{\mathbf{X}}$.

A key idea of principal component analysis is to project data points orthogonally onto a suitable line or—by iterative construction—a plane, or a hyperplane that typically has a lower dimension than the space from which the data was taken. E.g., if $\mathcal{D}$ has two features, then the points are to be projected onto a line, given by a *column* vector $\mathbf{u}$. Figure 15.4 visualizes and introduces the most important variables and steps for the projection. The left panel shows that the process of projecting consists of decomposing the position vector $\mathbf{x}$ (which points to one of the data points and starts at the origin) into a vector that is parallel to the line given by $\mathbf{u}$ and a vector $\mathbf{n}$ that is perpendicular to $\mathbf{u}$. Here, $\mathbf{u}$ is a vector representing an line with arbitrary inclination; determining its “most suitable” direction will be done in the section below. The two vectors, $\mathbf{p}$ and $\mathbf{n}$, can be easily computed using vector calculus (the detailed derivation is given in Appendix A.1.8) as

Algorithm 15.1: Computation of the Principal Components

1. Ensure that the data matrix $\mathbf{X}$ contains n features in columns, each of the m rows is a record.
2. Ensure that $\mathbf{X}$ is centered, e.g., for each feature i we subtract the mean of all records $E[X_i]$ from each entry of the column vectors X_i :

$$\bar{X}_i := X_i - E[X_i]\mathbf{1} \quad \text{with} \quad \mathbf{1} = [1, \dots, 1]^T \quad (15.4)$$

Optionally, standardize the data for each feature: $\bar{X}_i := \bar{X}_i/s_i$ where s_i is the standard deviation of feature i .

3. Compute the centered data covariance matrix $\bar{\mathbf{S}}$ from the centered data $\bar{\mathbf{X}}$:

$$\bar{\mathbf{S}} = \frac{1}{m-1} \bar{\mathbf{X}}^T \bar{\mathbf{X}} \quad (15.5)$$

4. Compute the eigenvalues λ_i and the corresponding unit eigenvectors $\mathbf{u}_i$ of the centered data covariance matrix $\bar{\mathbf{S}}$.
5. Sort the eigenvalues and corresponding eigenvectors such that $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ with all $\lambda_i \in \mathbb{R}$ and $\lambda_i \geq 0$; $\mathbf{u}_1$ is then the first principal component, and $\mathbf{u}_i$ is the i -th principal component.
6. Assuming that $\mathbf{U}$ is the matrix of the first k eigenvectors. Then, the projected coordinates of the data in the system of the k largest principal components are

$$\bar{\mathbf{X}}^* = \bar{\mathbf{X}}\mathbf{U} \quad \text{with} \quad \mathbf{U} = \begin{bmatrix} | & & | \\ \mathbf{u}_1 & \cdots & \mathbf{u}_k \\ | & & | \end{bmatrix}. \quad (15.6)$$

$$\mathbf{p} = c\mathbf{u} \quad \text{with} \quad c = \frac{\mathbf{x} \cdot \mathbf{u}}{\mathbf{u}^T \cdot \mathbf{u}} \quad (15.7)$$

$$\mathbf{n} = \mathbf{x}^T - \mathbf{p} = \mathbf{x}^T - c\mathbf{u}, \quad (15.8)$$

where the dot “ $\cdot$ ” denotes the scalar product between two vectors. Recall, that $\mathbf{x}$ represents a row from the centered data matrix $\bar{\mathbf{X}}$ and therefore is a row vector while $\mathbf{u}$ is a column vector, as is the convention. The vector $\mathbf{u}$ in Eq. (15.7) can be understood as a *basis vector* for the one-dimensional space of a line. c can be understood as the coordinate w.r.t. that basis vector.

Intuitive explanation: An intuitive analogy of the vector $\mathbf{u}$ and the scalar c is our usual three-dimensional space with Cartesian coordinates where the “coordinate axes” are given by the unit basis vector $\mathbf{e}_1 = [1, 0, 0]$, $\mathbf{e}_2 = [0, 1, 0]$, and $\mathbf{e}_3 = [0, 0, 1]$. Each point $\mathbf{x}$ is represented by three scalar values c_1, c_2 , and $c_3 \in \mathbb{R}$ such that

(continued)

$$\mathbf{x} = c_1 \mathbf{e}_1 + c_2 \mathbf{e}_2 + c_3 \mathbf{e}_3 \in \mathbb{R}^3. \quad (15.9)$$

The only difference with regard to the above “projection coordinate system” is that $\mathbf{u}$ is not necessarily a vector with unit length (at least not here), and that we are dealing with only one dimension, which requires only one basis vector.

The right panel of Fig. 15.4 shows all resulting points projected onto the line that was given by $\mathbf{u}$. These points can be understood as an approximation for the original data. Clearly, there is an error involved, which would get smaller the closer to the line all points are located. At the same time, the larger the deviation from the line, i.e., the variance along the perpendicular direction, the less appropriate it is to use only one component.

Variance and Mean Squared Error as a Function of Line Orientation

So far, the lines’ angle given by $\mathbf{u}$ was arbitrarily assumed, but obviously there are angles of inclination of the new axis that are “better” suited than others (again, we still have to define what “better” means in this context). Figure 15.5a,b show two examples of different orientations together with the corresponding orthogonal projections of the dataset. We observe that the line in Fig. 15.5a as compared to the one in Fig. 15.5b looks like a very good fit to the data in the sense of a line fitting approach: the normal distance seems to be, on average, rather small. We can now sample the whole space of different orientations of the projection axis and compute two characteristics for the resulting configurations: the total variance as a function of the rotated direction, $\text{Var}_{\mathbf{u}}(\bar{\mathbf{X}})$ and the mean squared distance (i.e., the error *mean*

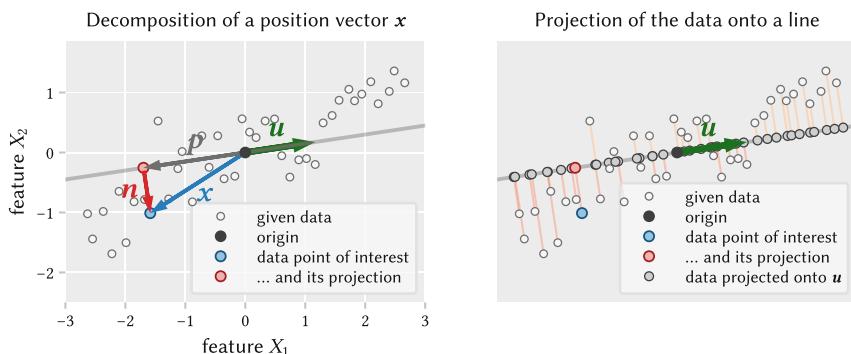


Fig. 15.4 The left panel shows data points in two dimensions. One of the points at $\mathbf{x}$ is projected on a line, which is given by $\mathbf{u}$. The figure shows how the vector $\mathbf{x}$ is decomposed into a two components, $\mathbf{n}$ and $\mathbf{p}$. The right panel visualizes a one-dimensional coordinate axis, spanned by the vector $\mathbf{u}$ together with all projected points of the dataset

squared error (MSE)) of the data from the line. The total variance of the projected data is defined as

$$\text{Var}_u(\bar{\mathbf{X}}) = \frac{1}{N_s - 1} \sum_{i=1}^{N_s} \|\mathbf{p}_i\|^2 = \frac{1}{N_s - 1} \sum_{i=1}^{N_s} c_i^2, \quad (15.10)$$

where $\|\mathbf{p}_i\|$ is the Euclidean distance (see Appendix A.1.7) of the projected point $\mathbf{p}_i$ from the origin in the one-dimensional coordinate system of $\mathbf{u}$, and the summations run over all N_s samples. The MSE value in this rotated coordinate system is obtained from the perpendicular distance $\|\mathbf{n}_i\|$ of the points to the line,

$$\text{MSE}_u(\bar{\mathbf{X}}) = \frac{1}{N_s} \sum_{i=1}^{N_s} \|\mathbf{n}_i\|^2. \quad (15.11)$$

The resulting plots are shown in Fig. 15.5c where the length of $\mathbf{u}$ was kept constant, and only its angle was changed. The lines show that MSE and the variance behave opposite to each other: the minimum of Var_u coincides with a maximum MSE_u value and vice versa. Taking a look at the above two equations, it can be seen that they have the same structure. The pre-factors are slightly different but will become nearly the same for larger number of samples, N_s . $\mathbf{p}$ and $\mathbf{n}$ are perpendicular to each other (cf. Eq. (15.8)) and therefore are, as a function of angle, offset by $\pi/2$, which explains the shift of the maxima and minima in the plot. Thus, finding a rotated coordinate axis that minimizes the MSE implies that the variance of the data is maximized.

Generalization and Matrix Notation

In a next step, we will now simplify some of the equations. At the same time, also the formulations are generalized such that they can be applied to the whole data matrix and that they can be easily implemented with Python and the numpy package. This requires a bit more of vector or matrix calculus but has the benefit of very concise and computationally efficient formulations.

We now assume without loss of generality that $\|\mathbf{u}\| = 1$ and discuss how vectors and matrices are transformed upon change of basis, i.e., change of the coordinate system. In what follows, only the most important aspects are summarized.

Assume that the current system is a two-dimensional Euclidean coordinate system. A record number i of the centered data matrix is given as the row vector $\bar{\mathbf{x}}_i$. Another unit vector $\mathbf{u}$ is given as a column vector, and we ask what the projection of $\bar{\mathbf{x}}_i$ onto $\mathbf{u}$ is.¹ The assumption that $\mathbf{u}$ has unit length simplifies the above equations, Eq. (15.7), such that the projected coordinate c_i of each point $\bar{\mathbf{x}}_i$ reduces to

¹ As mentioned before, the differentiation between row and column vectors can be tiring and would be unnecessary if tensor calculus is employed. Nonetheless, for most of the readers, tensor calculus makes the formulations rather inaccessible, and the implementation with Python is not always obvious. Hence, we have to put up with row and column vectors and plenty of “ $\mathbf{T}$ ” symbols.

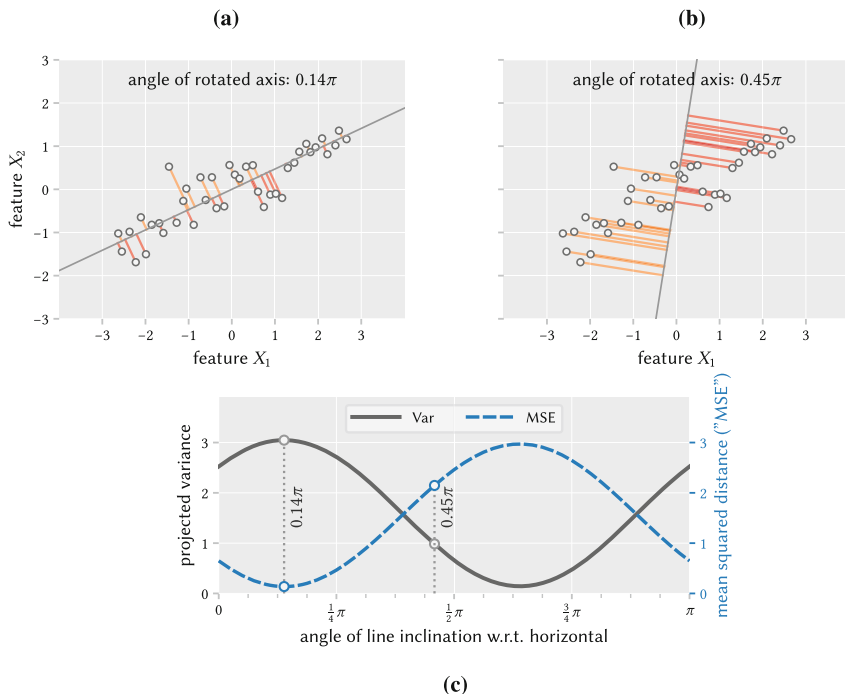


Fig. 15.5 The top left and top right panels depict the shortest distances of two-dimensional data to a line that is given by its angle w.r.t. the horizontal axis. The distances are shown as lines in two colors, to highlight on which of the two sides of the line the point is located. The bottom panel shows the projected variance and the mean square error (MSE) as a function of all angles. The points at the ends of the two dotted lines correspond to the configuration shown in the two top panels. (a) Optimal orientation. (b) Non-optimal orientation. (c) Variance and MSE in the projected system

$$c_i = \bar{\mathbf{x}}_i \cdot \mathbf{u} . \quad (15.12)$$

Rewriting this equation for all samples results in the projected data matrix, which we abbreviate in the following as $\mathbf{X}^*$:

$$\mathbf{X}^* := [c_1 \ c_2 \ \cdots \ c_{N_s}]^T = \bar{\mathbf{X}} \cdot \mathbf{u} . \quad (15.13)$$

Now it can be shown that the mean of each projected coordinate is zero for a centered dataset:

$$\mu_{\mathbf{u}} = \bar{\mathbf{X}}^* = \frac{1}{N_s} \sum_{i=1}^{N_s} c_i = \frac{1}{N_s} \sum_{i=1}^{N_s} (\bar{\mathbf{x}}_i \cdot \mathbf{u}) = \left(\frac{1}{N_s} \sum_{i=1}^{N_s} \bar{\mathbf{x}}_i \right) \cdot \mathbf{u} = \mathbf{0} \cdot \mathbf{u} = 0 \quad (15.14)$$

where we used the distributivity of the scalar product of vectors and, in the last step, the fact that the data are already centered with a mean value of zero for each feature, including the feature number i .

Last but not least, we also reformulate the expression for the total variance, Eq. (15.10), for the case that $\|\mathbf{u}\| = 1$. Point of departure is again Eq. (15.7) where we extend the expression for $\mathbf{p}$ from a single record $\bar{\mathbf{x}}_i$

$$\mathbf{p}_i = (\bar{\mathbf{x}}_i \cdot \mathbf{u}) \mathbf{u} \quad (15.15)$$

to the whole data matrix $\bar{\mathbf{X}}$:

$$\mathbf{P} = (\bar{\mathbf{X}} \cdot \mathbf{u}) \mathbf{u}^T \quad \text{with} \quad \mathbf{P} = \begin{bmatrix} - & \mathbf{p}_1 & - \\ & \vdots & \\ - & \mathbf{p}_m & - \end{bmatrix}. \quad (15.16)$$

This is already a bit more complicated. A good approach for understanding the details of these vector and matrix operations is to look at the dimensions of the involved variables: the product $\bar{\mathbf{X}} \cdot \mathbf{u}$ results in a column vector with m entries (m being the number of samples) where for each row i of $\bar{\mathbf{X}}$ the dot product with $\mathbf{u}$ is computed. Multiplication with the row vector $\mathbf{u}^T$ from the right is an outer product and results in a matrix $\mathbf{P}$ that has the same shape as $\bar{\mathbf{X}}$. Insertion into Eq. (15.10) gives:

$$\text{Var}_{\mathbf{u}}(\bar{\mathbf{X}}) = \frac{1}{N_s - 1} \sum_{i=1}^{N_s} c_i^2 = \frac{1}{N_s - 1} \sum_{i=1}^{N_s} (\bar{\mathbf{x}}_i \cdot \mathbf{u})^2 = \frac{1}{N_s - 1} \sum_{i=1}^{N_s} (\bar{\mathbf{x}}_i \cdot \mathbf{u}) (\bar{\mathbf{x}}_i \cdot \mathbf{u}) \quad (15.17)$$

$$= \frac{1}{N_s - 1} \sum_{i=1}^{N_s} (\bar{\mathbf{x}}_i \cdot \mathbf{u}) (\mathbf{u}^T \cdot \bar{\mathbf{x}}_i^T) \quad (15.18)$$

The parentheses in the last term can now be removed, the two $\mathbf{u}$'s taken out of the sum, and the term $\bar{\mathbf{x}}_i \bar{\mathbf{x}}_i^T$ is identified as an outer product. $\mathbf{u}$ does not depend on the record number i and therefore can be taken out of the summation. It follows

$$\text{Var}_{\mathbf{u}}(\bar{\mathbf{X}}) = \mathbf{u}^T \cdot \left(\frac{1}{N_s - 1} \sum_{i=1}^{N_s} (\bar{\mathbf{x}}_i^T \cdot \bar{\mathbf{x}}_i) \right) \mathbf{u} = \mathbf{u}^T \cdot (\bar{\mathbf{S}} \mathbf{u}) , \quad (15.19)$$

where we identified the term in the parentheses as the centered sample covariance matrix $\bar{\mathbf{S}}$. However, the final result is just a scalar due to the two-fold multiplication with the vectors $\mathbf{u}$.

In summary, we have managed to reformulated all relevant equations in a form that only consists of vector and matrix operations, no explicit summation has to be performed anymore. The advantage does not only lie in the superior computational

performance of any code, it is also a formulation that can be easily extended to the most general case of a multiple projection directions. In that case, we will use a matrix $\mathbf{U}$ that consists of a number of projection vectors and, e.g., Eq. (15.13) simply becomes

$$\bar{\mathbf{X}}^* := \bar{\mathbf{X}}\mathbf{U} \quad \text{with} \quad \mathbf{U} = \begin{bmatrix} | & & | \\ \mathbf{u}_1 & \cdots & \mathbf{u}_k \\ | & & | \end{bmatrix}, \quad (15.20)$$

where later on these $\mathbf{u}_i$ will be the first k *principal components* (PCs).

15.2.5 Maximizing the Variance along the First Principal Direction

We will now derive how to choose the new directions such as to maximize the covariance. We choose a *greedy maximization*, which means that we don't find all directions simultaneously but that we rather pick the direction that gives the most effective maximization first and then choose the subsequent direction such that the remaining variance is also maximized. It can be shown that this solution is also a solution for the global maximization problem. This approach is one of the more accessible derivations among a number of different derivation approaches. A related approach is to minimize the MSE, which is left for the reader as an exercise.

While we now obtained a formulation for the variance of the projected data as function of the line vector $\mathbf{u}$, we still don't know how to choose $\mathbf{u}$ as to maximize the projected variance. Simply finding the maximum, e.g., by taking a derivative of Eq. (15.19) and finding the $\mathbf{u}$ for which the derivative (or the gradient more specifically) becomes zero is not sufficient as this always would result in $\mathbf{u} = \mathbf{0}$ as a trivial solution. This, however, would violate that the length of $\mathbf{u}$ should be constant—we even postulate without loss of generality a value of 1 (see the previous section). If the length would not be fixed, then it would be very easy to change the variance just by changing the length. Hence, this “side condition” is important, leading to a *constrained optimization* problem:

The Constrained Optimization Problem of PCA:

Maximize the projected variance $\text{Var}_{\mathbf{u}}(\bar{\mathbf{X}})$ by simultaneously ensuring the constraint that $\|\mathbf{u}\| = 1$.

Solving such a class of problems can be done using the concept of *Langrangian multiplier*. It is based on formulating a *Lagrange function* $\mathcal{L}$ which returns in a scalar value. $\mathcal{L}$ consists of two terms, the term to be maximized (i.e., the variance from Eq. (15.19)) and a term that contains the constraint, formulated such that it is zero if the constraint is fulfilled and multiplied by a weighting factor—the so-called Lagrangian multiplier λ . The Lagrange function reads:

$$\mathcal{L}(\mathbf{u}; \lambda) = \mathbf{u}^\top \cdot (\bar{\mathbf{S}}\mathbf{u}) - \lambda (\mathbf{u}^\top \cdot \mathbf{u} - 1) . \quad (15.21)$$

Note, that $\mathbf{u}$ is the variable while λ is a parameter. The goal is now to maximize $\mathcal{L}(\mathbf{u})$, which is a different problem than maximizing the variance. For this we take the first derivative of $\mathcal{L}$ w.r.t. $\mathbf{u}$ and search for solution such that it becomes $\mathbf{0}$:

$$\frac{\partial \mathcal{L}(\mathbf{u}; \lambda)}{\partial \mathbf{u}} = \mathbf{0} \quad (15.22)$$

(which we sometimes write in short as $\partial_{\mathbf{u}} \mathcal{L}(\mathbf{u}; \lambda)$). Insertion of Eq. (15.21) gives the following expression

$$\frac{\partial \mathcal{L}(\mathbf{u}; \lambda)}{\partial \mathbf{u}} = \frac{\partial (\mathbf{u} \bar{\mathbf{S}} \mathbf{u}^\top)}{\partial \mathbf{u}} - \lambda \frac{\partial (\mathbf{u}^\top \cdot \mathbf{u} - 1)}{\partial \mathbf{u}} = 2 \bar{\mathbf{S}}\mathbf{u} - 2\lambda \mathbf{u} = \mathbf{0} . \quad (15.23)$$

There we used the fact that $\bar{\mathbf{S}}$ does not depend on $\mathbf{u}$ and therefore is considered as constant during differentiation. As $\mathbf{u}$ occurs as a quadratic term it is $\partial_{\mathbf{u}}(\mathbf{u} \cdot \mathbf{u}) = 2\mathbf{u}$. Dividing both sides by two and bringing the second term to the right-hand side of the equation, we obtain the result as

$$\bar{\mathbf{S}}\mathbf{u} = \lambda \mathbf{u} . \quad (15.24)$$

This is the *eigenvalue equation* that we already encountered in the context of visualizing the normal distribution. Further details can also be found in the appendix in Appendix A.3.1. The fact that we arrived at this equation has the following implications and consequences:

Implications and Consequences: There are two important results that directly follow from the fact that we obtained an eigenvalue equation:

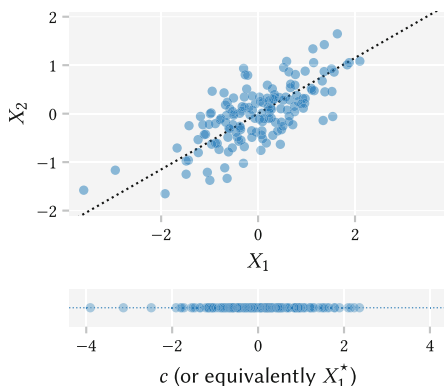
- The vector $\mathbf{u}$ is an eigenvector and
- The Lagrange parameter λ is an eigenvalue of the (centered) covariance matrix $\bar{\mathbf{S}}$.

The fact that $\mathbf{u}$ has unit length with $\mathbf{u}^\top \cdot \mathbf{u} = 1$ helps to solve the equation for λ by multiplying both sides with $\mathbf{u}^\top$:

$$\bar{\mathbf{S}}\mathbf{u} = \lambda \mathbf{u} \quad \Leftrightarrow \quad \mathbf{u}^\top \cdot \bar{\mathbf{S}}\mathbf{u} = \lambda \mathbf{u}^\top \cdot \mathbf{u} \quad (15.25)$$

$$\Rightarrow \quad \lambda = \mathbf{u}^\top \cdot \bar{\mathbf{S}}\mathbf{u} . \quad (15.26)$$

Fig. 15.6 The original data matrix $\mathbf{X}$ has two features (top panel), the dotted line shows the direction of the first principal axis as obtained from the first eigenvector $\mathbf{u}_1$. The bottom shows the projected data $c = \mathbf{X}\mathbf{u}_1$ along the first principal axis. Note that the two horizontal axes X_1 and c have a different scaling due to the different basis vectors



This is an important result as it allows us to conclude which is the direction of maximum variance: it is the direction of the eigenvector that corresponds to the largest eigenvalue of the data covariance matrix. The direction of an eigenvector is also called a *principal direction*, and the direction corresponding to the *largest* eigenvalue is called the *first principal component*.

Projecting each data point $\mathbf{x}_i$ onto the direction of the first principal component gives its coordinate in the coordinate system defined by the principal direction, $c_i = \mathbf{x}_i \cdot \mathbf{u}$. This is also known as the *principal component score value*. Already before we stated the equation in compact notation that holds for all points of a dataset, $\mathbf{X}^* = \bar{\mathbf{X}}\mathbf{u}$. Now, the new finding is that $\mathbf{u}$ is not just an arbitrary vector but a very special one: the first eigenvector.

An example application where these steps are used for reducing a two-dimensional dataset to one dimension is given in Example 15.2, a visualization is shown in Fig. 15.6 (also compare Fig. 15.3). The reader is encouraged to use this example as a starting point for own experiments, e.g., by using different distributions or computing the variance or MSE of the projected data.

Example 15.2 Computing the First principal component with Python ✂

Consider a dataset that consists of two-dimensional points, obtained from sampling a bivariate normal distribution that is characterized by given mean values and a covariance matrix:

(continued)


```
In [1]: import numpy as np
        from scipy.stats import multivariate_normal

        true_mean = np.array([3.1, 0.9])
        true_cov = np.array([[0.7, 0.3],
                             [0.3, 0.3]])

        rv = multivariate_normal(mean=true_mean, cov=true_cov)
        X = rv.rvs(size=150)
        X = X - np.mean(X, axis=0)

        sample_cov = np.cov(X, rowvar=False)
        sample_cov
```

```
Out [1]: array([[0.6329477 , 0.27707288],
                [0.27707288, 0.29295513]])
```

`x` contains two columns each of which has been centered. The output shows the data covariance matrix that has slightly different values than the population (“true”) covariance matrix. Each time this code is run, it will create slightly different values due to the random sampling. From `eig`, we obtain eigenvalues and -vectors and can compute the score values, i.e., the coordinates w.r.t. the first PC:

```
In [2]: eigenvals, eigenvecs = np.linalg.eig(sample_cov)
        if eigenvals[0] > eigenvals[1]:
            u = eigenvecs[:, 0]
        else:
            u = eigenvecs[:, 1]

        c = X @ u
```

Last but not least, we visualize the original data and the projected data

```
In [3]: import matplotlib.pyplot as plt

        fig, ax = plt.subplots(nrows=2, gridspec_kw={'height_ratios': [1, 0.1]})
        ax[0].plot(X[:,0], X[:,1], '.')
        ax[0].plot([-2, 2], np.array([-2, 2]) * u[1] / u[0])
        ax[1].plot(c, np.zeros_like(c), '.')
```

which results in a plot similar to the one shown in Fig. 15.6.

15.2.6 The Second Principal Direction

Principal component analysis is concerned with the k principal directions that represent as much detail from the data as possible. So far, we found the first principal

direction. How can we find the next principal direction? The next direction should be orthogonal to the previous direction, which is a requirement that is automatically fulfilled for the eigenvectors as they are mutually orthogonal. The new direction is supposed to maximize the variance along the new direction.

We will now denote the first, already known direction by $\mathbf{u}_1$ and the second, to be determined direction by $\mathbf{u}_2$. For now, we assume that $\mathbf{u}_2$ need not necessarily be an eigenvector; it can be an arbitrary unit vector orthogonal to $\mathbf{u}_1$ satisfying

$$\mathbf{u}_1^\top \cdot \mathbf{u}_2 = 0 \quad \text{and} \quad \|\mathbf{u}_2\| = \sqrt{\mathbf{u}_2^\top \cdot \mathbf{u}_2} = 1. \quad (15.27)$$

The main difference to the previous derivation is that the Lagrangian function now consists of *two* constraint terms (that is, the two equations in Eq. (15.27)), one ensuring the unit length and the other ensuring the orthogonality:

$$\mathcal{L}(\mathbf{u}_2; \lambda_2, a) = \mathbf{u}_2^\top \cdot \bar{\mathbf{S}} \mathbf{u}_2 - a(\mathbf{u}_2^\top \cdot \mathbf{u}_1 - 0) - \lambda(\mathbf{u}_2^\top \cdot \mathbf{u}_2 - 1). \quad (15.28)$$

Here, λ_2 and a are just two Lagrangian multiplier, and we do not need to assume that one of them is an eigenvalue (even though this is exactly what will come out below). The rest of the derivation is straightforward and follows along the line of the derivation for the first principal component. We start by computing $\partial_{\mathbf{u}_2} \mathcal{L}(\mathbf{u}_2; \lambda_2, a) = 0$, which gives:

$$2\bar{\mathbf{S}} \mathbf{u}_2 - 2\lambda_2 \mathbf{u}_2 - a \mathbf{u}_1 = \mathbf{0}. \quad (15.29)$$

By multiplication with $\mathbf{u}_1^\top$ from the left we obtain

$$2\mathbf{u}_1^\top \cdot \bar{\mathbf{S}} \mathbf{u}_2 - 2\lambda_2 \mathbf{u}_1^\top \cdot \mathbf{u}_2 - a \mathbf{u}_1^\top \cdot \mathbf{u}_1 = 0. \quad (15.30)$$

We realize that transposing the first term can be undone by additionally exchanging the position of the two vectors, $\mathbf{u}_2$ and $\mathbf{u}_1$, and $\bar{\mathbf{S}}$ is symmetric. $\mathbf{u}_2$ is by construction orthogonal to $\mathbf{u}_1$, which cancels the second term. The scalar product in the third term is one because $\mathbf{u}_1$ is a unit vector with $\|\mathbf{u}_1\| = 1$. With this we obtain:

$$2\mathbf{u}_2^\top \cdot \bar{\mathbf{S}} \mathbf{u}_1 - a = 0 \quad \Leftrightarrow \quad a = 2\mathbf{u}_2^\top \cdot \bar{\mathbf{S}} \mathbf{u}_1. \quad (15.31)$$

With insertion of the identity from Eq. (15.24) for the first eigenvector and -value we obtain

$$2\mathbf{u}_2^\top \cdot \bar{\mathbf{S}} \mathbf{u}_1 = 2\lambda_2 \mathbf{u}_2^\top \cdot \mathbf{u}_2 \quad (15.32)$$

$$\Leftrightarrow \quad 2\bar{\mathbf{S}} \mathbf{u}_1 = 2\lambda_2 \mathbf{u}_2. \quad (15.33)$$

With this, we can now simplify Eq. (15.29), which brings us again to an eigenvalue equation

$$\bar{\mathbf{S}} \mathbf{u}_2 = \lambda_2 \mathbf{u}_2 . \quad (15.34)$$

With the same argument as before, one can conclude that $\mathbf{u}_2$ is an eigenvector, corresponding to the eigenvalue λ_2 . $\mathbf{u}_2$ maximizes the (remaining) variance while at the same time its direction is orthogonal to $\mathbf{u}_1$. The vector $\mathbf{u}_2$ thus defines the *second principal component* .

15.2.7 The k -th and the Best k Principal Directions

The k -th principal direction

Common applications of PCA or related methods in ML analysis and data mining either treat the number of principal components as a hyperparameter or have as the sole aim to determine the number of PCs. Thus we would like to be able not only to obtain the first or first two principal directions but the first k principal directions.

The steps for obtaining the resulting equations for the k -th principal component are a generalization of the steps for obtaining the second component. We assume that the first $k - 1$ eigenvectors $\mathbf{u}_i$ and -values λ_i are already known. Again, the Lagrangian function consists of three groups of terms: (i) the function to be minimized, i.e., the variance projected onto the new direction $\mathbf{u}_k$, (ii) a constraint for ensuring that the vector governing the k -th direction has unit length, and (iii) a sum of constraints that enforce the mutual orthogonality between all directions:

$$\mathcal{L}(\mathbf{u}_k; \lambda_k, a_1, \dots, a_{k-1}) = \mathbf{u}_k^T \cdot \bar{\mathbf{S}} \mathbf{u}_k - \lambda_k (\mathbf{u}_k^T \cdot \mathbf{u}_k - 1) - \sum_{i=1}^{k-1} a_i (\mathbf{u}_k^T \cdot \mathbf{u}_i) . \quad (15.35)$$

As before, the variable of the Lagrange function, $\mathbf{u}_k$, is unknown and might be arbitrary; the same holds for the parameter λ_k . Again, it will turn out that these are the k -th eigenvector and -value. The derivation for this follows the same steps as before: Taking the derivative and solving the resulting equation $\partial_{\mathbf{u}_k} \mathcal{L} = 0$ results again in an eigenvalue equation from which follows that λ_k and $\mathbf{u}_k$ are in fact an eigenvalue of $\bar{\mathbf{S}}$ and the corresponding eigenvector. Furthermore, the set of the k eigenvalues of $\bar{\mathbf{S}}$ has a corresponding set of k eigenvectors each of which maximizes the variance along its respective direction.

The Best k Principal Directions

Now, we are able to successively compute all $n = \text{rank}(\bar{\mathbf{S}})$ eigenvalues of the covariance matrix. The maximum number of principal directions is also n , which usually is also the number of features of the original data matrix. Without loss of generality, the eigenvalues and the corresponding eigenvectors can now be sorted such that

$$\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_k \geq \dots \geq \lambda_n \quad \text{where all } \lambda_i \geq 0 . \quad (15.36)$$

Thus, we obtain the best k principal directions, which maximize the variance by simply choosing the first k eigenvectors. All steps for computing the k principal directions and the corresponding projection of the data are summarized in Algorithm 15.1 on page 397.

Projecting the data onto these directions is an approximation of the original dataset and therefore introduces an error—but we managed to systematically reduce the dimension of the dataset from originally n to $k \leq n$, which is why PCA is a method for *dimensionality reduction*. To estimate the quality of this dimensionality reduction, we will now take a look at how the *total variance* of the projected data changes as more and more principal directions are taken into account.

Total Variance of the Projected Data

To assess the overall variance of the projected data w.r.t. the principal directions, we compute the total variance. To do so, there is an elegant way that is based on the *eigendecomposition* of a square matrix $\mathbf{A}$, which implies that any symmetric, real-valued matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ can be factorized as

$$\mathbf{A} = \mathbf{U} \mathbf{D} \mathbf{U}^T, \quad (15.37)$$

where $\mathbf{D}$ is a diagonal matrix consisting of the eigenvalues λ_i of $\mathbf{A}$ such that $\mathbf{D} = \text{diag}(\lambda_1, \dots, \lambda_n)$. $\mathbf{U}$ is an orthogonal matrix which contains the n eigenvectors scaled to unit length as columns, and without loss of generality we assume that the eigenvalues and the according eigenvectors are sorted such that λ_1 has the largest, and λ_n has the smallest value, compare Eq. (15.36). From $\mathbf{U}$ being an orthogonal matrix it thus follows that $\mathbf{U} \mathbf{U}^T = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix. Further details on orthogonal matrices and the diagonalization can be found in particular in Appendix A.2.17 and A.3.2. Further information on eigendecomposition and Singular Value Decomposition, which is a related method, can be found in Appendix A.3.5

Above, we found out that the covariance data matrix $\bar{\mathbf{S}}$ fulfills these prerequisites for the eigendecomposition, and we can therefore obtain the corresponding diagonal matrix as

$$\mathbf{D} = \mathbf{U}^T \bar{\mathbf{S}} \mathbf{U}, \quad (15.38)$$

which consists of all sorted eigenvalues of $\bar{\mathbf{S}}$. The matrix $\mathbf{U}$ contains as columns all corresponding, mutually orthogonal eigenvectors of $\bar{\mathbf{S}}$. With this, any row vector $\mathbf{x}_i$ (i.e., a specific record i of the dataset) can be represented in terms of the basis system formed by the eigenvectors. The transformation from the Euclidean system to the eigensystem is simply done by the following linear transformation

$$\mathbf{x}_i^* = \mathbf{x}_i \mathbf{U}. \quad (15.39)$$

Similarly, we can also apply this transformation to the whole data matrix $\mathbf{X}$, as already introduced in Eq. (15.20)

$$\mathbf{X}^* = \mathbf{X}\mathbf{U} . \quad (15.40)$$

where column i contains the scores for all data points w.r.t. principal component number i and $\mathbf{U}$ acts as a projection matrix. Using the definition of the covariance matrix, we can now write down the formulation of the covariance matrix w.r.t. the principal components:

$$\bar{\mathbf{S}}^* = \mathbf{P}^T \bar{\mathbf{S}} \mathbf{P} . \quad (15.41)$$

Comparing this equation to Eq. (15.38) reveals that the data covariance matrix in the principal system is simply the diagonal matrix which contains all eigenvalues! Therefore, we can now very easily compute the *total covariance* of the projected data, $\text{Var}_{\mathbf{U}}^{\text{tot}}$ as the trace of $\bar{\mathbf{S}}^*$:

$$\text{Var}_{\mathbf{U}}^{\text{tot}}(\bar{\mathbf{X}}) = \text{tr}(\bar{\mathbf{S}}^*) = \sum_{i=1}^n \lambda_i . \quad (15.42)$$

If only the first k principal components are to be considered, then the total covariance comprises only the sum of the first k eigenvalues,

$$\text{Var}_{\mathbf{U}}(\bar{\mathbf{X}}, k) = \sum_{i=1}^k \lambda_i , \quad (15.43)$$

since the λ_i are sorted in decreasing order.

15.3 Application Aspects and Examples of PCA

This section will go in detail through several numerical and practical aspects of principal component analysis. This also helps to explain details of dimensionality reduction that can be generalized to other ML methods. The great benefit of PCA, however, is that it is rather simple in terms of its implementation.

15.3.1 Image Data and the Principal Component Analysis

So far, the governing equations of PCA were written for a number of features, i.e., vector-valued data or several individual scalar variables. However, a particularly interesting type of “data” are images. How can we treat images in the PCA framework as outlined in Algorithm 15.1? Grayscale images consist of an regular

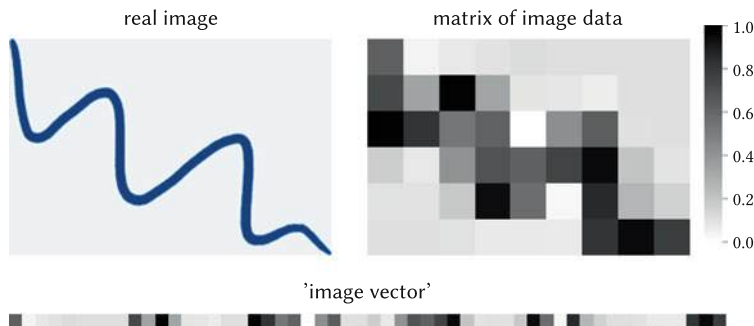


Fig. 15.7 Preprocessing of image data for the use with [PCA](#): the real image is represented as a two-dimensional matrix of values (here: strongly down-sampled). The values are here represented by different gray shades, and the gray values result from anti-aliasing of the real image. Then, the “image vector” is obtained by concatenating all rows of the matrix into a single vector where each gray value corresponds to a number, shown below the “image vector”

array of pixels each of which has a value in between 0 and 255 (or 0 and 1, depending on the underlying convention) binary images only contain values of 0 or 1. Such a two-dimensional array can be converted into a one-dimensional representation by *flattening* the image into a vector: there, each row of pixel is appended to a resulting vector as sketched in [Fig. 15.7](#). Thus, an image consisting of n rows and m columns of pixel is mapped to a vector with $n \times m$ elements. In case of a black and white image with 256 by 256 pixels, such a vector would be an element of a huge, 65536-dimensional vector space. In this space, each image is a single point. The combination of pixels that in fact results in a picture with interpretable, intelligible shapes or patterns is very rare, by comparison. This can be demonstrated by picking random constellations of pixels as shown in [Fig. 15.8](#). Even though these three images are from different points of the high-dimensional space, none of them results in meaningful patterns. This also implies that most of the non-random images are very particular. For example, images of human faces have some common features, which also show in the vectors of pixel values. Even images of very different features will exhibit some similarities and therefore only will occupy a probably quite small subspace of the high-dimensional space of all possible image vectors, the images are *embedded* into the high-dimensional space. Principal component analysis can help to group similar images together in the principal component space. As an example, we turn now to one of the classical datasets, the MNIST dataset, DS-2 (MNIST).

The MNIST Dataset

The dataset DS-2 (MNIST) is one of the famous and classical datasets for [ML](#), see [Sect. 4.8](#) for further information. In the context of [PCA](#), it has the advantage that the content of the dataset consists in very clear geometrical shapes that are easily accessible (we know very well how numbers look). Therefore, this dataset will help us to more easily interpret our results.



Fig. 15.8 The space of all possible pixel combinations mostly consists of random looking images. Images with recognizable patterns and shapes are very rare, by comparison. For the three examples, pixel values were drawn from a random uniform distribution of numbers [0, 1]

One of the objectives for analyzing the images of handwritten digits is to be able to assign labels to them, i.e., to infer which number was written. Classical *feature engineering* approaches try to identify characteristic features or typical structures, e.g., a “1” consists always of a more or less vertical line, sometimes with a short slash at the top (at least in many European countries). The handwritten 7 at least also has the vertical line, but it also might come with the “European cross bar”. Therefore, finding characteristics and quantifying them is in fact rather difficult, due to the large variety of different writing styles. **PCA** does not perform a classification of the images but does group similar images together, which can be useful, e.g., as a “preprocessing” step for a **ML** classification. Projecting images onto the first two principal components is particularly useful as this can be easily visualized.

Python Implementation and Results

We will now explain the steps required in Python to perform a **PCA** for the MNIST dataset. We assume that the MNIST training dataset has been already read into the variable `x` and the corresponding labels (i.e., the 10 digits) are contained in `y`. The first index of `x` is the number of the image, the second and third indices are the row and column of the pixel. `y` is a one-dimensional array. We start with a quick sanity check:

```
In [1]: import numpy as np

# [acquire data for X and y] -- this is not shown here for brevity!
print("X.shape:", X.shape)
print("y.shape:", y.shape)

Out [1]: X.shape: (60000, 28, 28)
        y.shape: (60000,)
```

The output shows that we have altogether 60,000 data records (i.e., images), each of which is represented by a matrix of 28×28 pixel. These are the training dataset, the MNIST database additionally consists of an additional 10,000 images as testing data. As a preparation, we convert each image into a one-dimensional vector, which is the “canonical form”. Additionally, we center the data:

```
In [2]: X = X.reshape(-1, 28 * 28)    # create image vector from 28x28 img.

mu = X.mean(axis=0)                  # mean for each column
X = X - mu                           # center each column of X
```

As an optional step, the features of the data matrix are standardized. As opposed to centering of the data, this is optional and mostly useful in situations where the centered features have very different ranges.

```
In [3]: sigma = X.std(axis=0)         # standard deviation for each column
mask = np.isclose(sigma, 0)           # ensure that all entries of sigma are != 0
sigma[mask] = 1e-12                   # and set only those to 1e-12 which are ~ 0
X = X / sigma                         # standardize X (this is optional!)
```

Here, we ensured that none of the entries of the standard deviation `sigma`—a vector with $28 \times 28 = 784$ elements—is zero for avoiding “division by zero” errors. Now, the data covariance matrix can be computed, considering that `x` contains features as columns. Additionally, we obtain eigenvalues and eigenvectors:

```
In [4]: data_cov = np.cov(X, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eig(data_cov)
```

Both eigenvalues and corresponding `-`vectors are sorted such that the eigenvalues are in descending order:

```
In [5]: # get the indices that would sort the eigenvalues
idx = np.argsort(eigenvalues)[::-1]

eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:,idx]    # these are column vectors!
```

There, the “`[::-1]`” reverses the order of the indices (similar to `numpy.flip()`) that are returned by `argsort`. Now we can create the matrix of the two principal components—the first two eigenvectors—and project the data onto the first two principal components:

```
In [6]: U = eigenvectors[:, :2]       # matrix of the first 2 eigenvectors
X_proj = X @ U                       # perform the change of basis
```

The symbol `@` denotes the product of two matrices. `U` has as many rows as the total number of pixel per image; here it has two columns. `X_proj` has as many rows as samples and as many columns as there are principal components, such that the data projected onto the first two principal components is now contained as the columns of `X_proj`. Visualization of the data can now be done with a scatter plot:

```
In [7]: import matplotlib.pyplot as plt
plt.scatter(X_proj[:,0], X_proj[:,1], c=y, cmap='tab10', alpha=0.3)
```

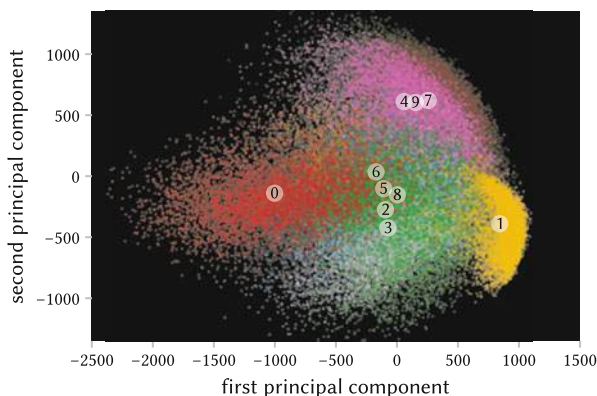



Fig. 15.9 All samples of the 10 digits contained in the MNIST training dataset projected into the latent space. The images of 28×28 pixels are projected onto the first two principal components. No standardization of the centered data was done. A circled number denotes the projected digit (e.g., the digit “0” is represented by the red marker), and the position of the number additionally indicates the center of the respective distribution

There, we made use of the fact that `matplotlib.scatter` can take a one-dimensional array as an argument for the color parameter `c`. This array contains the color values to be assigned to each of the points. The resulting distribution is shown in Fig. 15.9. We observe that already two components are able to “cluster” the data for each digit. Nonetheless, for many digits, there is still a large amount of overlap, which implies that in this embedding there is no unique map from a projected data point to a digit. Additionally, we also see that, e.g., the “1”s are much more localized than, e.g., the “0”s, which gives an idea about the variability of the original data.

Discussion

How can the distribution of points for this two-dimensional embedding be interpreted in terms of the original image data? Figure 15.10 shows a dataset that was reduced for ease of visualization: there, only the projected data for the digits 0 and 1, and 5 are shown, which—to some extent—occupy rather distinct regions with relatively small overlap. One can now pick a point and take a look at the corresponding input images, as shown in the figure. The distribution of “1”s is the most localized one. Taking a look at the three examples in the column of insets at the very right of Fig. 15.10, it can be observed that here, a higher value of the second component seems to imply a line inclined toward the left while a lower/negative second component implies an inclination toward the right.

The small images on the top show examples taken from the centers of the three distributions. The digits look upright, centered, and look relatively “average”. This conclusion, however, has to be taken with a grain of salt as there is some variation in the images of the points nearby.

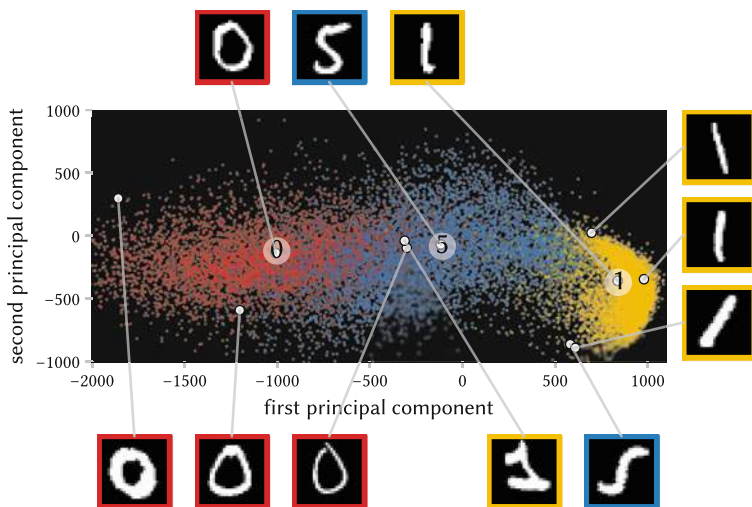


Fig. 15.10 Latent representation: The points are the projected images for the digits 0, 1, and 5 from the MNIST dataset. The latent subspace consists of the first two principal components. For chosen points of the dataset also the corresponding images (digits) are shown. The color of their frame indicates to which group of points they belong. Also shown are the centers of the distributions, indicated by the three circled numbers 0, 1, and 5. Note that the two axes do not have the same scale

How about the “extreme” positions? The three leftmost images on the bottom show data points that are far away from the mean. All of them contain some imperfections, untypical shapes or are particularly thick or thin, which seems to be an aspect of all extreme points. This can also be seen for the “1” in the bottom row of digits.

Finally, we take a look at two points in the [PC](#) space that are almost at the same location but belong to different digit classes. One of such examples are the points for “5” and “1” at the bottom right. The “5” can almost be taken for a “1” and has the same “average” orientation. Here, we might be tempted to conclude that nearby points are also visually similar. However, this is not always the case, as can be seen by comparing the “0” and “1” on the bottom and which are also located at nearly the same location in the embedding. Nonetheless, they are entirely different in shape.

Two principal components are obviously not sufficient to learn the difference between these two shapes. While it is certainly tempting to interpret the principal components and the representation of the data, it is important not to rely too strongly or too blindly on this and to remember that the principal components in all generality do not represent variables that can be physically interpreted.²

² There are cases, where principal components have been directly linked to features in images. An example is the so-called *eigenfaces* [16]: there the first eigenface represents the average gray value, the second eigenface accounts for the gray value contrast of face and hair, the third eigenface

15.3.2 Accuracy of the Representation

This leads to the question if or how the projected representation changes or how the accuracy is affected by increasing the numbers of principal components? Is it possible to represent differences better by using more PCs? One way of visualizing accuracy related aspects is a scree plot.

Scree Plot

Recall that one of the key ideas of PCA is to maximize the variance along each principal component—which implies to maximize the total variance. We already discussed that maximizing the variance is identical to maximizing the information content or minimizing the mean squared error of the representation.

In Eq. (15.43) we found that the total variance of the data projected onto the first k components is the sum of the first k , largest eigenvalues. This allows for an easy way for visualizing and analyzing the influence of the components. A *scree plot* as introduced by [6] is a line plot that shows the spectrum of the covariance matrix—or in other words: the magnitude of the eigenvalues obtained from the PCA, as shown as the solid line in the top panel of Fig. 15.11. As usual, the eigenvalues are sorted from largest to smallest, which are in total $28 \times 28 = 784$ (the number of pixel of each MNIST digit image) eigenvalues. Here we see, that the magnitude of the eigenvalues quickly falls below 5 after the first 25 eigenvalues, only the first ≈ 150 eigenvalues are larger than 1, and the λ_i are zero for $i > 712$. In the bottom panel of Fig. 15.11, a logarithmic scaling of the vertical axis was used, which in particular reveals more details in the regime where the eigenvalues are large and strongly changing. The scree plot helps to decide which are the components with the highest statistical significance. In such a plot, the “elbow” can be identified as the point of maximum curvature after which changes in the eigenvalues happen considerably slower. This approach should be used only as a rough estimate, though, as it is mathematically not well defined.

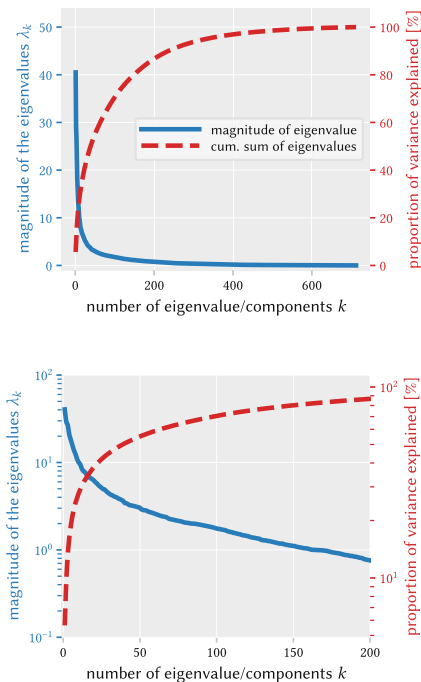
Scree and the Scree Plot



Scree plot is an unusual name that originates from the word *scree*, which is used to describe a mass of stone debris or broken rock fragments that has typically agglomerated at the foot of a hill or mountain. Why is this name chosen for the plot type? The largest eigenvalues are the top of the mountain and the big rocks while all the scree and the small pebbles are then at the foot of the mountain.

for lightning from the side, higher-order principal components represent features of the images of increasing complexity that are more and more difficult to interpret.

Fig. 15.11 Eigenvalues for all digits and the proportion of the total variance that is explained by the first i components. The left panel has a linear scaling of both axes while the right plot has a logarithmic scaling of the two vertical axes and additionally only shows the first 200 eigenvalues, revealing more details of the eigenvalues with larger magnitudes. The solid line in the left panel is what is typically called a “scree plot”, the dashed line shows the “amount of variance” that was explained by the first k components



An alternative representation is the cumulative sum of the eigenvectors as shown in the second curve in the two figures. The cumulative sum is scaled such that the maximum would be 100 percent because using *all* eigenvalues recovers the unreduced information content of the original data. This type of plot is particularly useful as the sum of the k first eigenvalues *is* the projected variance of the data in the k -dimensional embedding, compare Eq. (15.43). As such, it gives us the *proportion of variance that is explained* by the specific choice of k principal components.

$$\text{explainedVar}(k) = \frac{\text{Var}_u(k)}{\text{Var}_u^{\text{tot}}} = \frac{\sum_{i=1}^k \lambda_i}{\sum_{j=1}^N \lambda_j}, \quad (15.44)$$

where the total variance is given by Eq. (15.42), N is the total number of eigenvalues, and k is a given number of eigenvalues for which the coefficient should be obtained. Using numpy this value can be computed as follows:

```
In [1]: from numpy import cumsum, sum, arange
         cumsum_eigenvalues = 100 * cumsum(eigenvalues) / sum(eigenvalues)
```

There, we assumed that the eigenvalues are sorted in descending order and directly multiplied with 100 to convert into percent. It can be plotted by, e.g.,

```
In [2]: plt.plot(arange(1, cumsum_eigenvalues.size + 1), cumsum_eigenvalues)
```

where the `arange` expression ensures that the first eigenvalue starts with the index number 1 and not 0. In the top panel of Fig. 15.11, we can easily see that in order to represent 80% of the data (in terms of variance) at least the first ≈ 150 principal components need to be included, and for 90% of the variance explained 235 components would be required. Thus, with only 30% of the data we achieve to represent nearly the same amount of information as contained in the original dataset. Therefore, scree plots or related plots can help to determine the number of components that have to be included into a data analysis—in other words, they are a tool to determine the intrinsic dimensionality of a dataset.

15.3.3 Reconstruction

Clearly, referring to this or that percentage of “variance explained” is useful but at the same time also somewhat abstract in that sense that it is not clear what the error implies in terms of features of the images. To this end, we will now investigate how data from the lower-dimensional embedding can be transformed back into the original vector space of the 28×28 dimensional images.

Point of departure is the projection of the (centered) data matrix onto the principal components, which gives the coordinates in terms of the eigenvectors as derived in Eq. (15.20):

$$\bar{\mathbf{X}}^* = \bar{\mathbf{X}} \mathbf{U} . \quad (15.45)$$

In this equation, $\mathbf{U}$ is the matrix of eigenvectors which consists of all the eigenvectors in columns and is quadratic. $\bar{\mathbf{X}}$ is as always the centered data matrix with N_f in columns and N_s records. Because $\mathbf{U}$ is even an orthogonal matrix it is $\mathbf{U}^{-1} = \mathbf{U}^T$ and we obtain

$$\bar{\mathbf{X}} = \bar{\mathbf{X}}^* \mathbf{U}^T = \bar{\mathbf{X}} \mathbf{U} \mathbf{U}^T \quad (15.46)$$

where $\bar{\mathbf{X}}$ is the reconstructed data, which are here identical to the original data because $\mathbf{U}$ contains *all* eigenvectors. However, if only the first k eigenvectors are considered, we have to introduce a separate variable $\bar{\mathbf{X}}^*$ in Eq. (15.46) for the reconstructed data:

$$\bar{\mathbf{X}}^* = \bar{\mathbf{X}} \mathbf{U}_{[:,1:k]} (\mathbf{U}_{[:,1:k]})^T , \quad (15.47)$$

where $\mathbf{U}_{[:,1:k]}$ is a submatrix of $\mathbf{U}$ that consists of only the first k columns, cf. Sect. 3.2.4. As a final postprocessing step, only the centering and possibly also the standardization need to be undone. For the j th column, this reads

$$\mathbf{X}_j^* = \left(\bar{\mathbf{X}}_j^* \sigma_j \right) + \mu_j \mathbf{1}, \quad (15.48)$$

where σ_j and μ_j are the standard deviation and the expected value of the j -th feature, respectively. Writing this out for each element x_{ij} of $\mathbf{X}^*$ (for brevity we skip the superscript $*$), this is

$$\begin{bmatrix} x_{1,1} & \cdots & x_{1,N} \\ \vdots & & \vdots \\ x_{M,1} & \cdots & x_{M,N} \end{bmatrix} = \begin{bmatrix} x_{1,1}\sigma_1 + \mu_1 & \cdots & x_{1,N}\sigma_N + \mu_N \\ \vdots & & \vdots \\ x_{M,1}\sigma_1 + \mu_1 & \cdots & x_{M,N}\sigma_N + \mu_N \end{bmatrix} \quad (15.49)$$

where $\mathbf{X}^*$ has M records in rows and N features in columns.

Hence, given a point in the space of principal components, we can now transform it back into the original space. For the problem of the MNIST dataset, we can thereby “reconstruct”, e.g., an image from its representation in the principal component space as demonstrated in Fig. 15.12. The first three columns correspond to the three digits shown in Fig. 15.10, which were chosen from the means of the three distributions. Here we observe that already two components are sufficient to roughly reconstruct the “0” and the “1” such that they are legible. The “5” is quite blurry, has additional artifacts and therefore requires more components. Note that we could also take a look at the *weights*, i.e., the eigenvectors reshaped into a 28×28 pixel array, which helps to understand which structures are most relevant. Using 150 components for the reconstruction already results in very accurate images. The amount of data required for this representation is less than 20% of the original data. Therefore, PCA serves as a very effective compression method for these images.

Denoising

So far, we have already seen that PCA is a method that is able to perform a multitude of different tasks. We conclude this by a somewhat unusual application: *denoising* of images.

Denoising can be done by first learning the PCA components of a set of images with defects or noise, which serves to identify the basis for the transformation. Then, choosing a number of PCs that is—usually significantly—smaller than the total number of PCs, the data are reconstructed, and only the dominant characteristics from the images are pertained. In this way, the noise will be removed as a side effect of reducing the total variance.

As a demonstrator we again use the MNIST dataset. To set up the dataset, we chose in each image 40% of the pixel and replace their values by a value randomly sampled from the interval $[0, 255]$, which mimics image defects or other noise. Then, the dataset is centered by subtracting the mean value of all samples for each

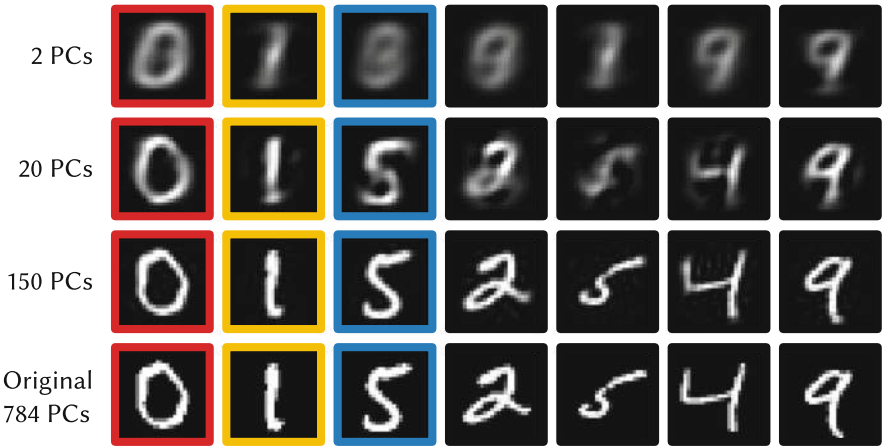


Fig. 15.12 Reconstruction of some MNIST digits using four different numbers of principal components (PCs). The first three digits are those that are shown on top of Fig. 15.10

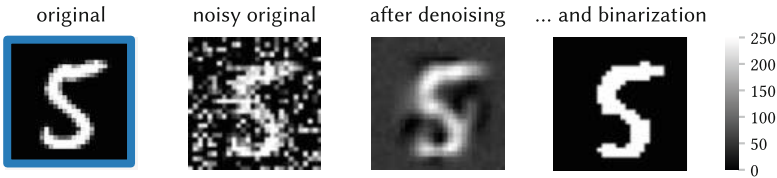


Fig. 15.13 Using PCA for denoising of images: the model was trained only with “noisy” version of the original MNIST training images. For further explanations, see the text

pixel i , μ_i . This is then used for a PCA to obtain the eigenvectors. We use the first $k = 40$ eigenvectors as a basis for the transformation of the noisy images into PC space and transform the result back to the original space. “Un-centering” the data is done by adding μ_i for each pixel. An example using the “5” from Fig. 15.12 can be seen in Fig. 15.13. The final thresholding step (pixel with values < 128 are set to zero, all other pixel are set to 255) is only for visualization purposes but shows already that the denoising can also be a very effective preparation for a binarization of an image.

The choice of the number of eigenvectors k is important: if k is too low, then the dominant structure is not represented properly, if it is too high, then more and more fluctuations can be represented as well. For the MNIST images, a value of $k = 25 \dots 50$ works relatively well, but there is also a certain dependency on details of the image as will be investigated as an exercise.

15.4 Further Methods for Dimensionality Reduction

PCA is a useful method, and it is always surprising how powerful this method is despite its simplicity. However, there are other methods as well some of which are able to overcome problems of **PCA**. Here, we only introduce a very incomplete selection. The first group of methods are methods related to **PCA** that are mainly based on matrix decomposition. The second group consists of the methods of the so-called *ensemble methods*.

15.4.1 The Relatives of Principal Component Analysis

For brevity, we will only cover two methods that are directly PCA-related and skip the classical methods of *independent component analysis* as well as *linear discriminant analysis*.

Randomized PCA

PCA becomes compute and memory intensive if the datasets become large (*large* being hundreds of thousands of records and above). *Randomized PCA* [10] is a method that uses random sampling in order to construct a reduced subspace into which the data are then projected. Randomized PCA is still a linear method.

Kernel PCA

Another extension of **PCA** is the method of *kernel PCA* which was originally created for classification problems that have a non-linear decision boundary. Similar to the concept of *support vector machines (SVMs)*, also here the *kernel trick* is utilized (cf. Sect. 14.6 for more information). Figure 15.14 shows the performance of the regular **PCA** in comparison to kernel **PCA** for the case of a two-dimensional dataset and a two-dimensional embedding. The dataset contains data of two categories that are not linearly separable. We observe that **PCA** is not able to separate the two groups of points as it is a linear method. In fact, the function between the original space and the embedding is close to the identity. Kernel **PCA**, on the other hand, is very well able to separate the data points because the transformation of the data using a kernel function (e.g., a RBF kernel consisting of products of Gaussian functions or a polynomial kernel, cf. Sect. 13.5.4) is non-linear. Reconstruction, i.e., projection of data back into the original feature space, however, is more complicated, due to the use of kernels. By comparison, the regular **PCA** is typically superior in this regards.

15.4.2 Manifold Learning Methods

PCA and related methods are parametric methods, as the mapping can be explicitly written in terms of a mathematical function. Unsupervised, *non-parametric dimen-*

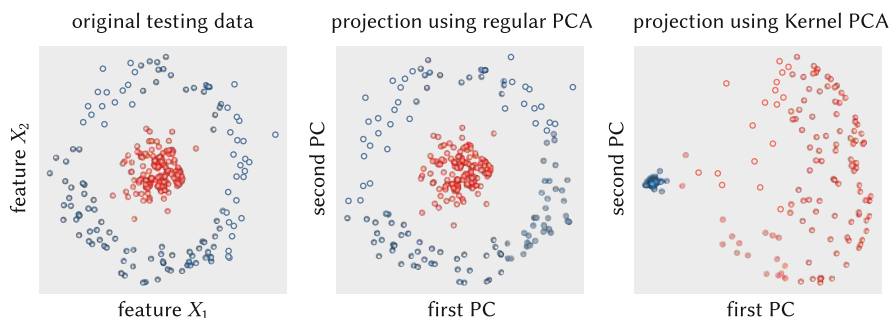


Fig. 15.14 Comparison of the “vanilla” PCA with kernel PCA

sionality reduction is an alternative, and is generally associated with the so-called *manifold learning*.

Among the most prominent examples are the isomap method [15], spectral embedding [1], multidimensional scaling [4], *t*-distributed stochastic neighbor embedding (*t*-SNE) [17], and *Uniform Manifold Approximation and Projection* (UMAP) [11]. In particular *t*-SNE as well as UMAP are very useful for data visualization and will be briefly explained in the following.

t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE is related to *multidimensional scaling* by the fact that both methods are based on pairwise distances of the data. Multidimensional scaling attempts to find distances between points in the embedding such that they are as close as possible to the original (high-dimensional) distances. However, in general it is not possible to preserve the high-dimensional distances in a lower-dimensional space. *t*-SNE (or in fact also the original SNE) takes an alternative approach and tries to preserve the structure of the neighborhood of points the original space when transformed to the embedding. This is achieved by introducing the measure of similarity of points (the closer the points the higher the similarity). A kernel function is introduced in the high-dimensional space, where the kernel size is adaptively chosen such that always the same number of points are covered. This results in one of the important hyperparameters, the *perplexity* parameter, which governs, roughly speaking, the number of desired nearest neighbors for a central point inside a cluster. Larger values of perplexity result in more nearest neighbors, which as a consequence might not show smaller structure. Smaller values of perplexity hide more of the global structure. Typical values are in the range of 5–50.

The low-dimensional similarity is defined based on the students-*t* distribution, hence the “*t*” in *t*-SNE (the original SNE just used a Gaussian kernel). The resulting loss function contains both similarity measures and can be numerically minimized using a gradient descent method.

The Relation of t-SNE to Molecular Dynamics



Interestingly, it turns out that the structure of the loss function is similar to that of a potential with repelling and attractive components. As a consequence, if we consider each data point a particle, then the gradient descent optimization implies to follow the trajectory of a multi-body simulation, as also done in molecular dynamics simulations. The resulting equilibrium configuration of the “data particles” in two dimension is the embedding.

The solution of this problem is not unique, and the minimization exhibits many local minima. To avoid “getting stuck” with a configuration that is far from optimal (e.g., in form of split clusters), a strategy called *early exaggeration* is used: for a number of steps, all attractive forces are scaled by a value (e.g., a factor of 4 for 100 or 200 iterations). Subsequently, the system is switched to the regular, non-scaled forces. Early exaggeration requires at least these two hyperparameters (a factor and the number of steps). Last but not least, we briefly mention that initialization plays an important role. For this, using a [PCA](#)-based initialization is a common method and also usually the default in most software packages.

[t-SNE](#) plots are not entirely trivial to read [18]: (i) hyperparameters have a strong influence on how the results look; (ii) the size of resulting clusters does not have a meaning; (iii) distances between clusters are possibly not interpretable; (iv) for interpreting plots in terms of topology, one might have to see several plots with different perplexity values. Figure 15.15 shows two examples of how t-SNE transforms different types of data from a two-dimensional space to a two-dimensional embedding. The upper row shows that for small numbers of perplexity, many small structures occur and there is no global structure. Clear clustering is observed for a perplexity of 30 and 50. For 100, the solution is rather extreme: the attractive forces between the particles of the “membrane” are in equilibrium with the repelling forces between the “core” and the “membrane”. Here, early exaggeration might help to separate the two group into to topologically simpler blobs. The bottom row shows again that a perplexity of 2 mainly preserves local structure, resulting in the circle. For increasing numbers of perplexity, more and more global structure is considered. The gap in the line, e.g., at perplexity of 5, is the result of repelling forces of points that are in the original space far away from each other.

Uniform Manifold Approximation and Projection (UMAP)

[UMAP](#) approaches the manifold learning from a slightly different angle and is based on RIEMANNIAN geometry and algebraic topology [11] for preserving both local and global structures in the data. In many situations [UMAP](#) performs computationally better than [t-SNE](#). Additionally, it is not restricted in terms of the embedding dimensions. Thus, UMAP *can* be used in a reasonable way for two- or three-dimensional visualization purposes, but it also can produce higher-dimensional data. This makes [UMAP](#) a good choice when the dimension-reduced data is supposed to

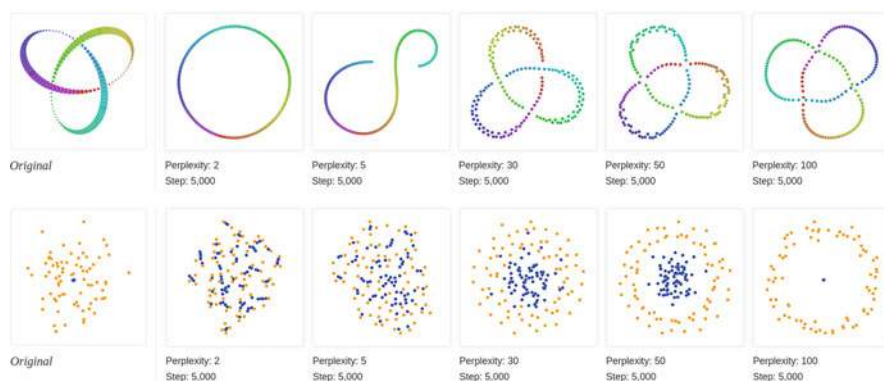


Fig. 15.15 Top row: 2 times 75 points with different dispersion. Only looking at the plot with perplexity 100 suggests that the orange points are a one-dimensional, circular structure, which is not the case. Bottom row: Also here, using different values of perplexity helps to show different types of information. (Diagrams taken from [18] with permission) (licensed under Creative Commons Attribution CC-BY 2.0, see <https://creativecommons.org/licenses/by/2.0/>)

be used as input for any other ML methods. There are a number of hyperparameters required for a UMAP-based dimensionality reduction. The four most important hyperparameters are: (i) the number of neighbors, (ii) the minimum distance between points, (iii) the number of components of the embedding (i.e., a value of one produces data for a line plot, a value of two produces data for a two-dimensional scatter plot), and (iv) the used metric. Figure 15.16 shows a comparison of t-SNE, PCA und UMAP applied to the MNIST dataset. First of all, there are significant differences in the computational time for performing the transformation on the dataset with 70,000 records. On the laptop with which this book was written, t-SNE required 7.30 min, UMAP took 1.30 min, and PCA was nearly instantaneously finished with a mere 2.5 s. Running Kernel PCA on this dataset was not possible, due to very extensive memory requirements. Here, UMAP is able to create more distinct “clusters”. If the goal would be to use this data as input for a clustering method or a supervised classification method, one would clearly choose UMAP. Nonetheless, if data visualization and data exploration are the main purpose, then it is certainly worth trying out both of them and additionally varying some of the main parameters.

UMAP $\neq$ t-SNE?



Even though the formulations in the respective publications of UMAP and t-SNE are rather different, it turns out that there is an interesting correspondences. Using t-SNE with early exaggeration for the MNIST data shown in Fig. 15.16 the embedding is nearly identical to what results from UMAP. Further discussions and investigations of this topic can be found in [7, 8].

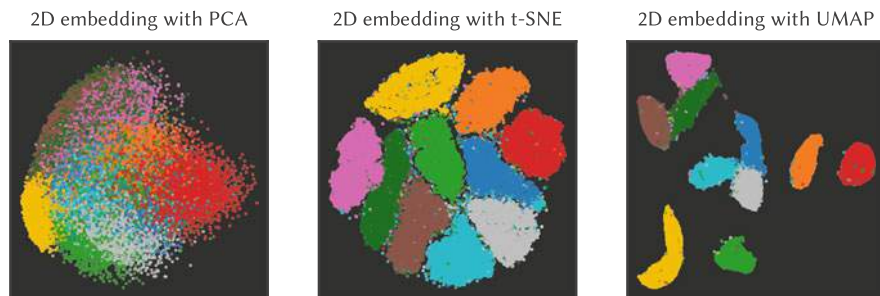


Fig. 15.16 PCA, t-SNE, and UMAP applied to the MNIST dataset. t-SNE was used with a perplexity value of 30, for UMAP we chose 15 neighbors as main parameter. For all other parameter the default values were used

15.5 Clustering

Cluster analysis is an ML technique for grouping similar objects into clusters. Clustering is an unsupervised method, and therefore, the dataset consists of only the N features,

$$\mathcal{D} = \{X_1, \dots, X_n\}_{i=1 \dots M} , \quad (15.50)$$

while the corresponding labels $Y_i, i = 1 \dots M$ are wanted for all M examples. How to mathematically or algorithmically characterize and detect a cluster, this is the point where the following methods all differ slightly. In the following, only a brief overview of clustering methods is given where we only give more details for the Gaussian mixture model. For a more detailed overview, see the review by [14] and the references therein.

15.5.1 k-Means and k-Medians

The k-means clustering algorithm has the goal to create k partitions from a given dataset, where hard boundaries delimit each cluster. This implies that there may be no overlap between clusters. This can be a strong restriction, as we already saw in the context of classification in Chap. 14. In fact, there is a relationship between the unsupervised k-means algorithm and the supervised *k-nearest neighbors* (kNN) method (cf. Sect. 14.4) as also the k-means method can be interpreted in terms of a Voronoi tessellation. The “means” in k-means stems from the fact that in this algorithm the centroids of Voronoi cells are determined from the mean values of the cluster data. As hyperparameter, k-means requires the number of clusters. The method *k-medians* is very similar to k-means; the difference is that it uses the medians for determining the Voronoi cell centers and not the means. Both methods have the restriction that the cells consist of polygons.

The algorithm consists of a iterative sequence of so-called “expectation steps” and “maximization steps” (more information will be given in Sect. 15.5.2 for the Gaussian mixture model).

Algorithm 15.2: k-Means Clustering



1. Decide on the number of clusters, k , e.g., based on physical knowledge of the underlying problem
2. Initialize the k centroids randomly
3. Expectation Step: Assign each data record to its nearest centroid
4. Maximization Step: Compute the new mean of each cluster. This is then the updated centroid.
5. GOTO 3 and repeat UNTIL convergence has been reached.

15.5.2 Gaussian Mixture Model

The *Gaussian mixture model* (GMM) [3, 12] is a probabilistic model, similar to the *Gaussian naive Bayes* (GNB) model that was introduced in the context of classification (Sect. 14.5), but the main difference is here that the labels are to be predicted.

Model Description

For the GMM it is assumed, that the *probability density function* (PDF) of a single cluster takes the functional form of a normal distribution $\mathcal{N}$. Multiple clusters are governed by multiple (possibly multivariate) normal distributions, all of which are additively superimposed. If a normal distribution of the cluster number j is characterized by the list of parameters θ_j (usually the mean μ_j and the covariance matrix Σ_j), we can write the weighted superposition as

$$F(\mathcal{D} \mid \Phi_k) = \sum_{j=1}^k \alpha_j \mathcal{N}(\mathcal{D} \mid \theta_j) \quad (15.51)$$

where k is the total number of clusters, which we have to provide, and the $\alpha_i > 0$ are the weights for each cluster with $\sum_j \alpha_j = 1$. In the following we adapt the commonly used notion of the *component*, which is identical to a cluster. The complete set of parameters for the Gaussian mixture model is then denoted by

$$\Phi = \{\alpha_1, \dots, \alpha_k, \theta_1, \dots, \theta_k\}. \quad (15.52)$$

For the **GMM**, “learning” refers to identifying the values of all model parameters such that the superimposed Gaussian distributions are the best fit to the training data.

Parameter Estimation from Maximum Likelihood

How can we obtain the parameters of the normal distributions? Recall, that during classification with the **GNB**, we could simply compute the mean and the standard deviation from the data that was already “sorted” according to the class labels. Here, this is not possible, because there are no class labels given. Therefore, a different approach needs to be taken. In **GMM** this is based on an error minimization, during which the parameters of the normal distributions are optimized with the objective that the resulting distribution F fits best to the data. The default approach for such a probabilistic model is to obtain the optimal distribution F based on the maximum likelihood method.

Maximum Likelihood Estimation

An important assumption is that all data points are independently distributed random samples. Then, one obtains the probability p of a single data point x_i from the superimposed Gaussian distributions as:

$$p(\mathbf{x}_i | \boldsymbol{\theta}) = \sum_{j=1}^k \alpha_j \mathcal{N}(\mathbf{x}_i; \boldsymbol{\theta}_j) , \quad (15.53)$$

from which the likelihood L of the entire dataset as the product of the individual probabilities of all M data records $\mathbf{x}_i$ is obtained:

$$L = \prod_{i=1}^M p(\mathbf{x}_i | \boldsymbol{\theta}_j) . \quad (15.54)$$

L can be understood as the joint probability of the combined likelihood of all the individual data points (note that L is not normalized, it can thus not be interpreted as a probability density f). As in the context of the **GNB**, for numerical purposes, we take the logarithm of the likelihood (see the comment at the end of Sect. 14.5):

$$\ln L = \sum_{i=1}^M \ln \left(\sum_{j=1}^k \alpha_j \mathcal{N}(\mathbf{x}_i; \boldsymbol{\theta}_j) \right) , \quad (15.55)$$

where we additionally made use of the fact that $\log(a \cdot b) = \log(a) + \log(b)$. $\ln L$ is now a function of the data and the model, given through the set of model parameters, i.e., $(\alpha_j, \mu_j, \sigma_j)$ for the one-dimensional case or $(\alpha_j, \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)$ for the general, multi-dimensional case.

Expectation–Maximization Algorithm

For obtaining the best fitting model parameters, a minimization problem is to be solved. There, the partial derivatives of $\ln L$ with respect to each parameter are required to be zero. To simplify the derivation, we only consider the case of univariate normal distributions here; the multivariate case follows the same line of argumentation but would require to deal with more complex derivatives of matrix products and covariance matrices. In the one-dimensional case, the j -th distribution is fully characterized by the mean μ_j and the standard deviation σ_j . Together with the third model parameter α_j , which governs the weighting of the individual distributions, we require that

$$\frac{\partial \ln L}{\partial \mu_j} = 0, \quad \frac{\partial \ln L}{\partial \sigma_j} = 0, \quad \text{and} \quad \frac{\partial \ln L}{\partial \alpha_j} = 0, \quad (15.56)$$

which results in a system of $(3k - 1)$ nonlinear equations.

For solving this system of equations, the so-called *expectation maximization algorithm* is employed, which is an iterative, numerical algorithm. In the *expectation step* (commonly called “E step”), we assume, that the model parameters (μ_k, σ_k) are fixed for each of the k sub-model. Then, $p(r | x_i)$ gives the probability that point x_i belongs to cluster number r .,

$$p(r | x_i) = \frac{\alpha_r \cdot \mathcal{N}(x_i; \mu_r, \sigma_r)}{\sum_{j=1}^k \alpha_j \cdot \mathcal{N}(x_i; \mu_j, \sigma_j)}. \quad (15.57)$$

The *maximization step* (“M step”) starts with computing the derivative of the log-likelihood w.r.t. μ_j (see Eq. (15.56)) as

$$\frac{\partial \ln L}{\partial \mu_j} = - \sum_{i=1}^M p(j | x_i) \frac{\partial}{\partial \mu_j} \left[\ln \sigma_j + \frac{(x_i - \mu_j)^2}{2\sigma_j^2} \right] = 0, \quad (15.58)$$

and the other two partial derivatives are written analogously. By solving these equations, we obtain new estimators for the parameters of the j -th normal distribution,

$$\mu_j = \frac{\sum_{i=1}^M p(j | x_i) x_i}{\sum_{i=1}^N p(j | x_i)} \quad \text{and} \quad \sigma_j^2 = \frac{\sum_{i=1}^M p(j | x_i) (x_i - \mu_j)^2}{\sum_{i=1}^N p(j | x_i)}, \quad (15.59)$$

as well as for the weighting factor

$$\alpha_j = \frac{1}{N} \sum_{i=1}^M p(j | x_i) . \quad (15.60)$$

Therein, M denotes the number of data records used for the fit. With these improved estimates for the parameter, we can then return to Eq. (15.57). Once the parameter values do not change anymore (up to a tolerance), we can then terminate the iteration. Algorithm 15.3 summarizes the GMM algorithm for the case of univariate Gaussian distributions. The generalization to a higher-dimensional feature space can be done by extension.

Algorithm 15.3: Univariate Gaussian Mixture Model with k Components



We assume that the number of components k is known, guessed, or given.

I. Initialization

1. Randomly choose mean values $\mu_1, \dots, \mu_k$, e.g., by randomly sampling k values without replacement from the dataset.
2. Define the variances σ_j^2 of each components j as the sample variance, e.g.,

$$\sigma_j^2 = \frac{1}{M} \sum_{i=1}^M (x_i - \bar{x})^2$$

where $\bar{x}$ is the sample mean of $\mathcal{D}$.

3. Set all k weighting factors $\alpha_r = 1/k$.

II. The Expectation Step (E Step)

1. Compute the probability that point x_i belongs to cluster number r according to Eq. (15.57).

III. The Maximization Step (M Step)

1. Calculate for each parameter and for all components a new estimator according to Eq. (15.59).
2. GOTO 1 and repeat UNTIL convergence has been reached.

15.5.3 Model Selection with the BIC Criterium

Model selection is a process where from a set of candidate models the most suitable is chosen. The candidate models are obtained by varying model hyperparameters.

For example, in the case of the [GMM](#), the number of components k is such a hyperparameter. For each value k , a candidate model is trained. Then, the optimal model can be chosen using an appropriate selection criterion. The *Bayesian information criterion* ([BIC](#)) is one of the commonly used selection criteria [9] and is given by:

$$\text{BIC} = -2 \ln L + d \ln M . \quad (15.61)$$

where d denotes the total number of parameters of the model, M is the number of data records, and L is the maximum likelihood achieved by the model. The first term of the equation is the maximized likelihood of the model. An increase of the values of L results in a reduction of the BIC value. Therefore, the most likely model is that which has the lowest BIC value. The second term represents a penalty for the number of parameters, as the chances of overfitting grow with the number of parameters.

OCCAM’S RAZOR and the Curse of Dimensionality



The penalty in proportion to the number of parameters can also be interpreted in terms of OCCAM’S razor Sect. 1.2.5 where it is stated that the simpler model should be the one that is more likely to be the true one. Overfitting for higher-dimensional parameter spaces is the consequence of the fact that the “data coverage” is no longer sufficient.

The [BIC](#) is closely related to the *Akaike information criterion* ([AIC](#)). It reads

$$\text{AIC} = -2 \ln L + 2d . \quad (15.62)$$

The two terms have the same functionality as in case of the [BIC](#); however, the penalty term has no dependency on the number of data records.

In both cases, the goal is to minimize the [BIC](#) or [AIC](#), respectively; the model that gives such a minimum is considered the best suited model. An advantage of the [BIC](#) is that, if one of the candidate models is the “true model” (i.e., one that was used to generate the data, cf. Chap. 12), then the probability that [BIC](#) will select the true model increases with the size of the training dataset. The [AIC](#), on the other hand, is insensitive to the size of the training dataset.

15.5.4 Density-Based Methods: DBSCAN

Density-based methods do not have any restriction in terms of the shapes of the clusters. A cluster is understood as a region in which the density of data point (i.e.,

number of points per area, volume or hyper-volumes) exceeds a given threshold. One of the most well-known methods is DBSCAN.

The algorithm searches for regions with very high densities, while regions of low density are related to either noise or outliers. The original algorithm requires two parameters: the size of the neighborhood ε and the minimum number of points that indicate a “cluster region”. DBSCAN is very efficient and is suitable also for large datasets. As an additional advantage, it does not require the user to specify the number of clusters beforehand, as opposed to the k-means method.

15.6 Materials Science Examples

We will now turn to two examples: dimensionality reduction for time evolving microstructures and clustering of indentation data.

15.6.1 Dimensionality Reduction for Microstructures from the Ising Model

The first investigated dataset, MDS-2 (Ising Model), contains simulation data of microstructures at different temperatures. What are the features that PCA would use, and how do these patterns change over the course of the simulations? For visualization purposes, only the first two principal components of the whole dataset MDS-2 (Ising Model) are used. visualization of the temperature of the system as a function of these two components results in the plots shown in Fig. 15.17. There, high temperatures correspond to strong randomness without any structure. We also clearly see in the right panel that there is a significant difference between the data with temperature below the CURIE temperature $T_C \approx 2.26$ and the data above T_C . Higher temperatures result in more noisy microstructure, which strongly localizes the data in the principal space. As the typical “Ising microstructure” forms, the variance increases, which shows in the large scatter of the data in the principal space. Could we take an image of a microstructure and, based on that, predict the corresponding temperature? Only very roughly. For example, for principal components (30, −30), it is clear that the temperature must be below the CURIE temperature, but for other points in the principal space, this is not possible.

As a last step in the PCA analysis, we will take a look at a scree plot (cf. Fig. 15.18). We observe that after an initial phase of $k \approx 30$ components, the proportion of the variance explained rises up to approximately 70%. After this “elbow,” the curve’s inclination is much lower, and it takes another ≈ 130 components to reach 100%.

When we use t-SNE to visualize the structure of the dataset, we do not see any particular structure in the data, see Fig. 15.19. Also, varying the perplexity parameter of the t-SNE algorithm does not produce any significant changes.

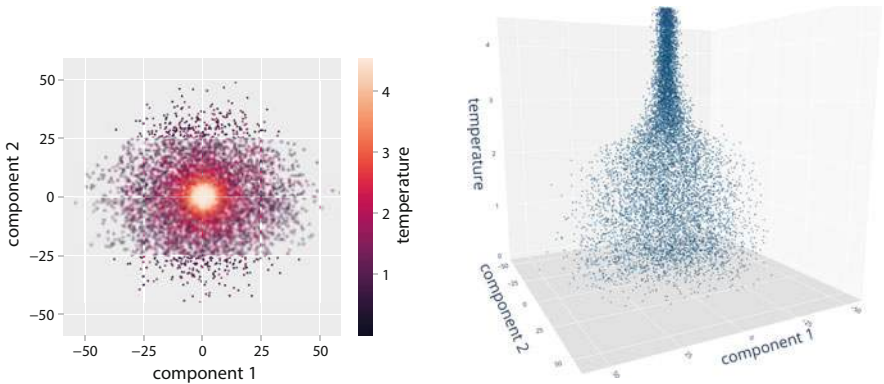


Fig. 15.17 Principal component analysis of microstructure data from MDS-2 (Ising Model). Shown is the temperature as a function of the first two principal components. In the left panel, the temperature is shown by the color of the marker, while in the right panel, the vertical axis is the temperature

Fig. 15.18 Scree plot for the data of the Ising model

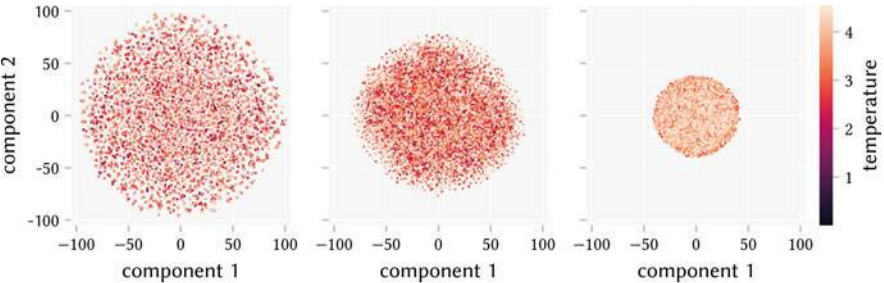
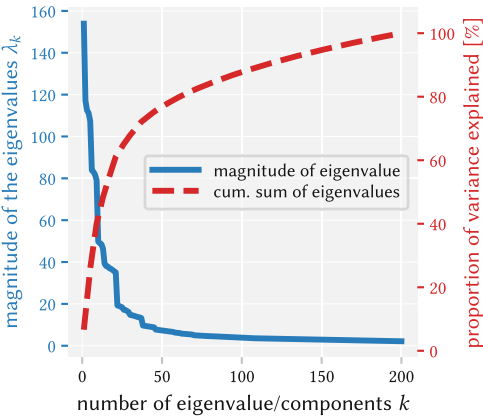


Fig. 15.19 Visualization of the MDS-2 (Ising Model) using t -SNE with three different perplexity values: 3, 12, and 40 (from left to right)

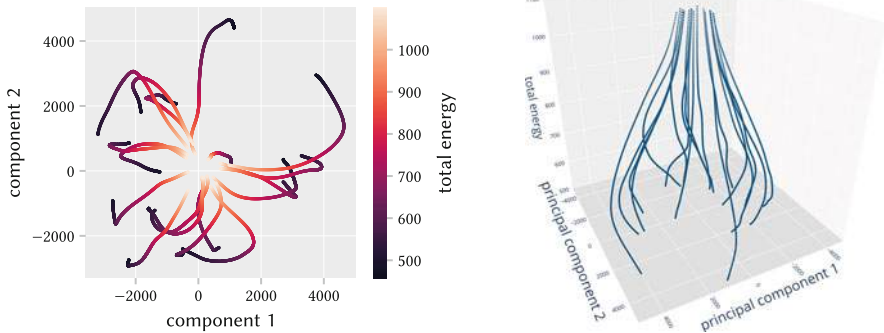


Fig. 15.20 Principal component analysis of microstructure data from MDS-3 (Cahn-Hilliard Model). Shown is the temperature as a function of the first two principal components. The left panel shows the temperature as indicated by the color, while in the right panel, temperature is plotted on the vertical axis

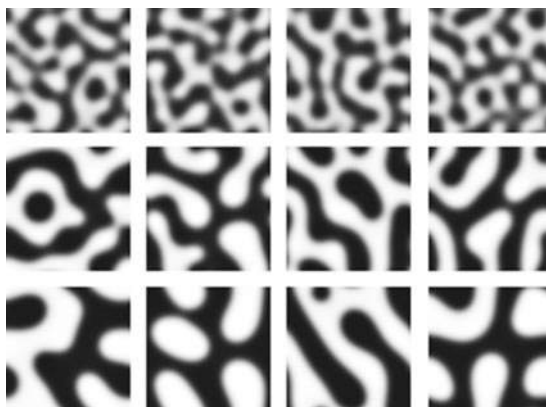
15.6.2 Dimensionality Reduction for Microstructures from the CAHN-HILLIARD Model

The dataset MDS-3 (Cahn-Hilliard Model) contains the data of phase microstructure evolutions together with the resulting total energies. Altogether 20 simulations are contained in this dataset. Taking a look at the first two components and the corresponding energy in Fig. 15.20 reveals a very interesting structure. The temporal evolution of each individual simulation can be exactly traced in the latent space. All simulations start at (0, 0), which is reasonable as all phase microstructures (the concentrations) of all simulations are initialized with uniform random distributions. From these line-like structures, we observe that (i) in between a few time steps, the microstructure does not change too much; (ii) each simulated microstructure seems to have a very particular structure since many of the lines occupy distinct places in the PCA space. Comparing these results to that of the MDS-2 (Ising Model) dataset from above, we clearly see in the dataset the absence of any randomness within the simulation model. Figure 15.21 shows the microstructure at three different stages of four different simulation runs.

Visually, it is not obvious that each microstructure evolution is entirely different or distinct. It is also not obvious that all microstructures seem to become increasingly different, at least in terms of the principal components.

From this part of the analysis, we can conclude, that at least in some aspects the dataset is statistically not representative because (a) most of the simulations do not intersect in the latent space and (b) the simulation data only cover a very small part of the latent space. Furthermore, each simulation is unique in that sense that at an early stage, there is a bifurcation point after which two simulation datasets strongly diverge and which determines much of the following microstructure evolution. Here,

Fig. 15.21 Microstructure evolution as obtained from the CAHN-HILLIARD model (MDS-3 (Cahn–Hilliard Model)). Each column is a different simulation. The top row is a state close to the beginning of the simulation, the middle row is an intermediate state, and the bottom row shows the microstructure at the point of time when the simulation was stopped



PCA might serve as a tool to monitor the data creation process and to help us judge if the number of simulations is already sufficient.

15.6.3 Clustering of Indentation Data

As a last example, we investigate the dataset MDS-5 (Nanoindentation) and present part of an analysis that has been published in [20, 21]. The first question was to get a baseline for how well the GMM can be expected to work for the quality and the amount of available data. For this purpose, we investigated two pure metals, Cu and Cr for which there should be only one material property along with the corresponding scatter. Figure 15.22 shows the PDF. Taking a look at the BIC, we see that the minimum value is reached for one component, indicated by the star, which is in line with our expectation that for a pure material there should be no other (harder or softer) “phases”.

Next, we turned to the composites and again analyzed the GMM for a number of different components k , Fig. 15.23. For these materials, we observe an entirely different behavior. Taking a look at the Young’s moduli (the two left panels in the figure), we can very well observe that the histogram fits to the superposition of three Gaussian distributions. Taking a look at the BIC criterion for model selection, this is supported by the minimum at $k = 3$. The three Gaussian distributions can then be related to the properties of Cu, the properties of Cr, and the properties of inter-phase region separating Cu-rich from Cr-rich regions.

Further details can be found in [20]. There, also a bivariate Gaussian mixture model was applied, to leverage the combined features of hardness and Young’s modulus.

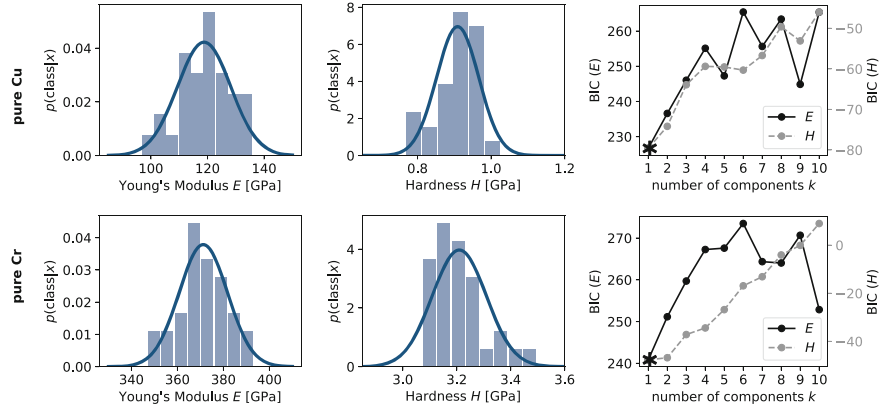


Fig. 15.22 Probability density functions (left and middle column) and plot of BIC (right column) of pure Cu and Cr. The BIC values are shown for both, Young's modulus (left "y" axis) and hardness (right "y" axis), where the star indicates the respective minimum. The top row shows the data for pure Copper, the bottom row for pure Chromium

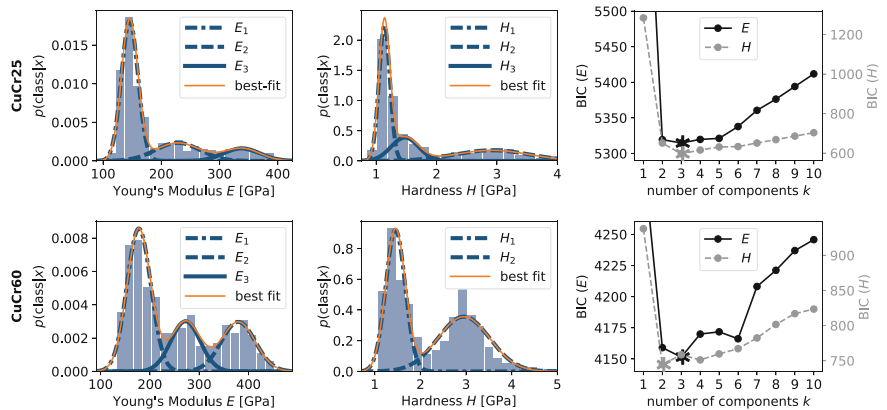


Fig. 15.23 1D GMM results of CuCr25 (top row) and CuCr60 (bottom row) at 1 μ m depth. The left panels show a histogram of the E_i and the fitted individual Gaussian as well as their superposition. The panels in the second column show histograms of the H_i and the (superimposed) Gaussians. The two panels on the right show the BIC of H and E , where the star indicates the respective minimum

15.7 Exercises

15.1. Consider the derivation of the principal component formulations and in particular Eq. (15.15). Why do we need p_i and c_i and can not just take the norm of p_i in order to obtain c_i ?

15.2. Assume that a dataset consists of k features and the principal component analysis that is performed on the dataset consists of k components. How much of the variance of the original dataset can be explained by the k components? Explain.

15.3. Do the PCA analysis for the MDS-3 (Cahn-Hilliard Model) dataset as done in Sect. 15.6.2 and create a scree plot. Compare it to the scree plot of the Ising model and discuss the differences.

15.4. Use the MDS-4 (Properties of Chem. Elements) dataset for a principal component analysis and plot the first two components. Color the marker according to whether they are metals or nonmetals. Repeat the same with the scikit-learn implementation of t-SNE and vary the perplexity parameter. Which method would you recommend as input for a cluster analysis?

15.5. Visualize the “weights” of the PCA of the DS-2 (MNIST) dataset. For this, take the eigenvectors and reshape them into images of 28×28 pixels. Which are the patterns or shapes that are the most important or most obvious ones?

15.6. This is a continuation of Exercise 15.5. Do the same study for the MDS-3 (Cahn–Hilliard Model) dataset and discuss the results in view of the results from the DS-2 (MNIST) dataset.

15.7. PCA for the MDS-2 (Ising Model): do a PCA of only two components and pick four points in the PC space that are located at extreme points (e.g. the maximum of component 1 and a zero value of component 2). Their temperature should be approximately the same. Compare the corresponding real space microstructure images among each other and with an image that has a zero value for both components.

15.8. Perform a visualization using the scikit-learn implementation of t-SNE for the MDS-3 (Cahn-Hilliard Model) dataset

References

1. M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 15(6):1373–1396, jun 2003. ISSN 0899-7667. DOI <https://doi.org/10.1162/089976603321780317>.

2. R. Bennett. The intrinsic dimensionality of signal collections. *IEEE Transactions on Information Theory*, 15(5):517–525, 1969. DOI <https://doi.org/10.1109/TIT.1969.1054365>.
3. C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.
4. I. Borg and P. J. F. Groenen. *Modern Multidimensional Scaling*. Springer New York, 2005. DOI <https://doi.org/10.1007/0-387-28981-x>.
5. M. Boubchir, R. Boubchir, and H. Aourag. The Use of Principal Component Analysis for the Prediction of Double Halide Perovskites A_2BX_6 . *Journal of Multiscale Modelling*, 12(02), 2021. DOI: <https://doi.org/10.1142/s1756973721500049>.
6. R. B. Cattell. The scree test for the number of factors. *Multivariate Behavioral Research*, 1(2):245–276, 1966. DOI https://doi.org/10.1207/s15327906mbr0102_10. PMID: 26828106.
7. S. Damrich, J. N. Böhm, F. A. Hamprecht, and D. Kobak. From t-sne to UMAP with contrastive learning, 2023.
8. A. Draganov, J. R. Jørgensen, K. S. Nellesmann, D. Mottin, I. Assent, T. Berry, and C. Aslay. Actup: Analyzing and consolidating tSNE and UMAP, 2023.
9. S. Gideon. Estimating the dimension of a model. *The annals of statistics*, 6(2):461, 1978.
10. N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):17–288, 2011. DOI <https://doi.org/10.1137/090771806>.
11. L. McInnes, J. Healy, and J. Melville. UMAP: Uniform manifold approximation and projection for dimension reduction, 2020.
12. D. A. Reynolds and R. C. Rose. Robust text-independent speaker identification using gaussian mixture speaker models. *IEEE transactions on speech and audio processing*, 3(1):72–83, 1995.
13. F. Saidi, S. Khetari, I. S. Yahia, H. Y. Zahran, T. Hidouri, and N. Ameer. The use of principal component analysis (PCA) and partial least square (PLS) for designing new hard inverse perovskites materials. *Computational Condensed Matter*, 31:e00667, June 2022. DOI <https://doi.org/10.1016/j.cocom.2022.e00667>.
14. A. Saxena, M. Prasad, A. Gupta, N. Bharill, O. P. Patel, A. Tiwari, M. J. Er, W. Ding, and C.-T. Lin. A review of clustering techniques and developments. *Neurocomputing*, 267:664–681, 2017. ISSN 0925-2312. DOI <https://doi.org/10.1016/j.neucom.2017.06.053>. URL <https://www.sciencedirect.com/science/article/pii/S0925231217311815>.
15. J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, Dec. 2000. DOI <https://doi.org/10.1126/science.290.5500.2319>.
16. M. Turk and A. Pentland. Eigenfaces for recognition. *J. Cognitive Neuroscience*, 3(1):71–86, jan 1991. ISSN 0898-929X. DOI <https://doi.org/10.1162/jocn.1991.3.1.71>.
17. L. van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008. URL <http://jmlr.org/papers/v9/vandermaaten08a.html>.
18. M. Wattenberg, F. Viégas, and I. Johnson. How to use t-sne effectively. *Distill*, 2016. DOI <https://doi.org/10.23915/distill.00002>. URL <http://distill.pub/2016/misread-tsne>.
19. C. Yang, Y. Kim, S. Ryu, and G. X. Gu. Prediction of composite microstructure stress-strain curves using convolutional neural networks. *Materials and Design*, 189:108509, Apr. 2020. DOI <https://doi.org/10.1016/j.matdes.2020.108509>.
20. C. Zhang, C. Bos, S. Sandfeld, and R. Schwaiger. Unsupervised learning of nanoindentation data to infer microstructural details of complex materials, 2023a.
21. C. Zhang, C. Bos, S. Sandfeld, and R. Schwaiger. Nanoindentation Datasets of Copper-Chromium Composites, Sept. 2023b. DOI <https://doi.org/10.5281/zenodo.8336072>.



I don't get wrapped up in technique and the like.

Fay Godwin (1931–2005)

British photographer

16.1 Feature Engineering and Feature Importance

How can we obtain create features for *machine learning* (ML) models and how can we show that these are possibly better than other choices? The next two subsections provide some fundamental knowledge about useful approaches and techniques.

16.1.1 Feature Engineering

Feature engineering is an ML technique that utilizes the existing variables to create new variables. The new variables are not contained in the original dataset and typically relate to the existing features in a nontrivial manner. Such a relationship can be, e.g., a simple variable transformation, e.g., using the square or the logarithm of an existing feature. In the context of feature engineering, this is called *feature transformation*. The motivation for performing this transformation is very similar the reason why variables are transformed in physics and in numerical maths: to simplify equations and to increase the stability and robustness of a model

Words of advice

Often, *feature engineering* is understood as everything that is part of the preprocessing (cf. Sect. 11.5). This would include the treatment of outliers, data imputation, as well as scaling and variable transformation. In this text, we only consider those techniques as feature engineering that actually result in new features. Data cleaning is not part of this.

A method related to feature transformation is *categorical encoding*, which is a kind of a transformation. The main idea is to transform categorical features into numerical features. The simplest way of doing this is to assign each different label an integer number; this technique is also called *label encoding*. *one-hot encoding* (OHE) is a method for turning different categories into a binary representation, cf. Example 16.1. One of the advantages of this technique is that an ML model can output different probabilities for each label, which reveals more information than just a single label output (also see Sect. 18.5 for a Python example).

Example 16.1 (One-Hot Encoding and Microstructure Classification) 

One-hot encoding is commonly used for classification with *deep learning* (DL) models. The idea is to convert different categorical class labels into a sequence of ones and zeros. For example, imagine a multi-class instance segmentation where a DL model detects different types microstructural features in a micrograph, e.g., a grain boundary, a twin, a dislocation, a precipitate, and the rest: background. The encoding is then

grain boundary	→	[1, 0, 0, 0]
twin	→	[0, 1, 0, 0]
dislocation	→	[0, 0, 1, 0]
precipitate	→	[0, 0, 0, 1]
background	→	[0, 0, 0, 0]

The output of a DL model would consist of four individual output “nodes,” each of which represents the binary value zero and one.

Feature engineering is also a good opportunity to introduce *domain knowledge* into an ML model, such that the feature represents physical variables or more complex, mathematical terms. In the context of feature engineering, this is called *feature creation*.

16.1.2 Permutation Feature Importance

Permutation Feature Importance quantifies the importance of a specific feature in the relationship between all input variables and the output. This technique was first introduced in 2001 by L. BREIMANN [1].

The first step is training a model with the true features. In a second step, the prediction error for the original model is compared with the prediction error after randomly permuting the data of an input variables one at a time (see Example 16.2 for an example of the permutation process).

Example 16.2 (Permutation of Features)



How does “permutation of the features” work in practice?

A given feature matrix **X** has the data shown in the left table while a permutation of the first feature is shown in the middle table, a possible permutation of the second feature is shown on the right:

original features:	perm. of 1st feature:	perm. of 2nd feature:																																				
<table><tr><td>X_1</td><td>X_2</td></tr><tr><td>1</td><td>10</td></tr><tr><td>2</td><td>11</td></tr><tr><td>3</td><td>12</td></tr><tr><td>4</td><td>13</td></tr><tr><td>5</td><td>14</td></tr></table>	X_1	X_2	1	10	2	11	3	12	4	13	5	14	<table><tr><td>X_1</td><td>X_2</td></tr><tr><td>3</td><td>10</td></tr><tr><td>1</td><td>11</td></tr><tr><td>3</td><td>12</td></tr><tr><td>5</td><td>13</td></tr><tr><td>4</td><td>14</td></tr></table>	X_1	X_2	3	10	1	11	3	12	5	13	4	14	<table><tr><td>X_1</td><td>X_2</td></tr><tr><td>1</td><td>10</td></tr><tr><td>2</td><td>14</td></tr><tr><td>3</td><td>11</td></tr><tr><td>4</td><td>15</td></tr><tr><td>5</td><td>12</td></tr></table>	X_1	X_2	1	10	2	14	3	11	4	15	5	12
X_1	X_2																																					
1	10																																					
2	11																																					
3	12																																					
4	13																																					
5	14																																					
X_1	X_2																																					
3	10																																					
1	11																																					
3	12																																					
5	13																																					
4	14																																					
X_1	X_2																																					
1	10																																					
2	14																																					
3	11																																					
4	15																																					
5	12																																					

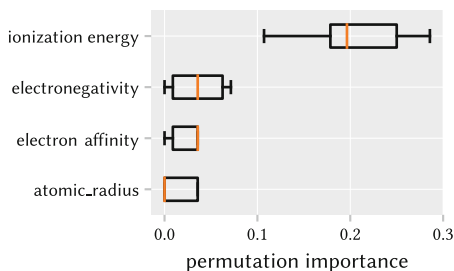
Assume that a model was trained with the original features, resulting in the following hypothesis of a regression problem

$$f(X_1, X_2) = 0.5 + X_1 \cdot \exp(X_2 - 1) \text{ .}$$

Then, *this* hypothesis is used for predictions based on the two permuted feature matrices.

A feature is considered more “important” than another feature, if the prediction using the permuted feature results in a higher error than for making prediction with another permuted feature. The argument for a higher feature importance is that a larger error indicated that the model “relies” more strongly for making predictions on that feature than on any other feature. If the original error is given by $\mathcal{E}^{\text{orig}}$ and the error after permuting the data of feature i is given by $\mathcal{E}^i$, then the feature importance FI^i is defined as any of the two measures

Fig. 16.1 Feature permutation importance for a classification problem with dataset MDS-4 (Properties of Chem. Elements). The whiskers in the box plot result from the fact that for each feature, a number of random permutations were performed



$$FI^i = \frac{\mathcal{E}^{\text{orig}}}{\mathcal{E}^i} \quad \text{or} \quad FI^i = \mathcal{E}^{\text{orig}} - \mathcal{E}^i. \quad (16.1)$$

This value is then computed for all features. To increase the reliability of that result, several different random permutations of the same feature are used and averaged. Typically, the permutation feature importance of all features is summarized by sorting the feature importance according to magnitude. Figure 16.1 shows the feature importance for the dataset “MDS-4 (Properties of Chem. Elements).” We observe that the ionization energy is by far the most important feature. It is also obvious that computing feature importance is also useful for *feature selection*, e.g., by dropping less important features. In this case, if we were to reduce the number of features, the atomic radius would certainly be the first feature to remove.

Words of advice



Care needs to be taken in cases where features are strongly correlated. While each individual feature might have a low feature importance, dropping *all* of them might result in a model that behaves significantly worse than the original one.

16.2 Data Splitting, Cross-Validation, and Statistical Resampling

In Sect. 8.4, we introduced how through *statistical sampling* a set of examples, the *sample* can be chosen from a *population*. Sampling implies very often that an investigated dataset is only a subset of the real, possibly even infinitely large population. Therefore, a good statistical sampling is one that creates a dataset through which the relevant statistical properties of a population are reflected. We have already discussed a number of sampling methods and systematic strategies for this purpose and also had a look at the source for different types of sampling errors.

In the context of ML model validation, we encounter another situation where sampling strategies are required: Training a model without validation may give wrong results. In Sect. 12.5.3, we saw that there can be big differences between the training performance of a model and the behavior with respect to new, previously unseen data. Remember, that in ML we are *not* interested how well the method works

on the training data, we *are* interested in the accuracy of the predictions when the method is applied to new data. This is the goal of *cross-validation*. It is useful for identifying problems such as overfitting or selection bias (i.e., a systematic error due to non-random sampling). However, before we introduce the cross-validation methods, we first need to discuss the difference between training, testing, and validation datasets.

16.2.1 The Difference Between the Testing and Validation Dataset

There is an important conceptual difference between the *testing dataset* and the *validation dataset*. What we have learned so far was that to evaluate the performance of a model, we have to use a subset of the whole dataset that has not been used for the training. Typically, this is called the *testing data*. As long as the whole dataset is only split into these two subsets, there is no confusion about which is which.

However, when a *hyperparameter study* is performed for, e.g., determining the most suitable model by varying parameter of the hypothesis function, care needs to be taken. In that case, the held-out dataset would be used for selecting the most suitable *candidate model*, e.g., by comparing the performance on the held-out dataset for each candidate model. We cannot use any of the data that were already used during the model selection, in the final performance evaluation. In such cases, *three* distinct datasets are required: the training dataset, the validation dataset, and the testing dataset, which are defined as follows:

Definition 16.1 (Training Dataset). The training dataset consists of all data records used for training a model.

Definition 16.2 (Validation Dataset). The validation dataset is a held-out dataset that is entirely separate from the training dataset. It is used for evaluating the performance of the different models during hyperparameter tuning.

After validation of all candidate models, we will choose the best-performing model, which is then called the *final model*. This is the point where the testing dataset comes into play:

Definition 16.3 (Testing Dataset). The testing dataset is a dataset that is entirely separate from the training and the validation dataset. Its purpose is to be used for evaluating the final model, i.e., after the model selection was finished.

In other words, if there is no hyperparameter optimization and there is just *the* model that we are going to train, using the concept of splitting the data into training and testing data is, in general, perfectly fine (which also explains why in such situations the differentiation between validation and testing set is not strictly followed).

As soon as a model needs to be selected from a set of candidate models, then we have to keep a validation and a testing dataset separated from the training data (and from each other).

The following sections introduce a number of methods that are commonly used for the purpose of cross-validation. Note, that most of these are explained for regression problems, e.g., using the *mean squared error* (MSE) as error measure. This can be easily transferred to classification problem by replacing the MSE with a suitable classification measure such as the relative number of falsely classified examples.

Words of advice



... on the notions of validation and testing: In the following subsections, we do not consider a split into a training, a validation, and a testing dataset but only a training and a testing dataset. Beware that in this case, the latter is often synonymously called testing or validation set. Below, you will encounter both notions, which might lead to some confusion.

16.2.2 The Holdout Method—Or the Train/Test Split

The *holdout method* is the simplest method for testing the performance of a model. It is sometimes also simply called “train-test split” and is probably the most commonly known method. The idea behind this method is as follows:

1. Partition the sample of data into two complementary subsets or *partitions* by randomly assigning data points to the subsets. The two partitions are the training and the validation set.
2. Perform the training (i.e., build the model) on the training set.

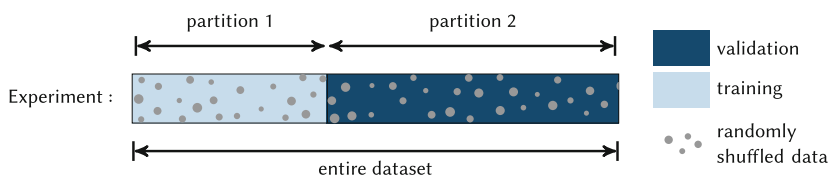


Fig. 16.2 The holdout method (i.e. the “common train/test split”) splits the full dataset (symbolized by the “data dots”) into disjunct training and validation datasets, each of which contains random samples from the full dataset. The performance of the trained model is then estimated based on the so far “unseen” validation data

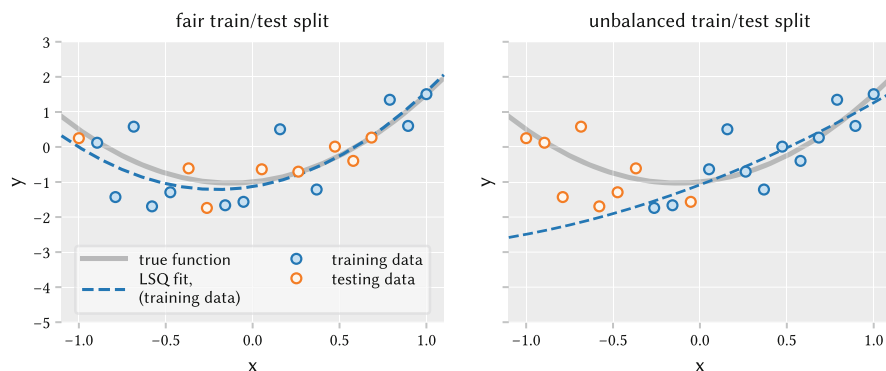


Fig. 16.3 Two scenarios of splitting a dataset into 60% train data and 40% test data. Left: all data points have the same probability to end up as train or test data. Right: test data are chosen with a higher probability from the left region of the dataset. The true function is the function used to create the data without noise. The dashed lines show a least square fit of a quadratic polynomial to the train data, showing the prediction of this model

3. Validate the analysis (i.e., validate the model performance) on the other subset (the validation or testing set¹).

Usually, the training set is larger than the validation/testing set. Typically used fractions of the total dataset size are 60%–80% of the dataset for training and 40%–20% for testing/validation. The concept is shown in Fig. 16.2.

The method requires only a single validation run and is therefore rather fast. However, depending on the sample size or the type of sampling, the holdout method can be very sensitive with respect to the particular choice of the partitioning of the data, and for small datasets, this method is strongly influenced by the selection bias. As an example, in Fig. 16.3, the effect of a rather extreme selection bias on the train/test split is shown. There, the noisy data are created by sampling the “true function” given by an quadratic equation, which was superimposed with random

¹ In particular for the holdout method, the terms validation and testing are often used interchangeably in the literature.

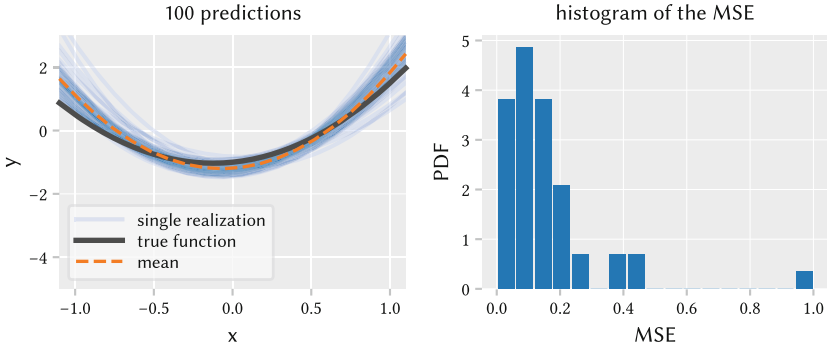


Fig. 16.4 100 random realizations of a 60% training/40% test data split in analogy to the example shown in the left panel of Fig. 16.3. Shown are the resulting fitted lines in blue, plotted on top of each other (left panel); the dark line is the true function; the dashed line is the mean of all 100 experiments. The right panel shows the histogram of all MSE values

noise. The dashed line is obtained from a least square fit to the training dataset, and the resulting quadratic function then represents the learned model. A statistically “unbalanced” train/test split results in two sub-datasets that are statistically not representative of the whole dataset. While the performance of the model for the training data is good, it does not perform well for the test data, as can be seen by the big deviation between fitted line and test data. As for the unbalanced training dataset, almost no data for the left third of the plot are available, it is not surprising that there the model performance is worse than for the balanced train/validation data. This “selection bias” is a particular problem for small datasets or for those that have a very heterogeneous distribution of data, e.g., with a strong clustering in some regions.

Quantitative Analysis

Quantitative analysis of the performance can be done for such a regression problem based on the MSE as error measure. For example, for the above example, the true function is known, and based on that the difference between prediction and true values can be obtained and used in the MSE. Figure 16.4 shows an ensemble of 100 different, random train/test splits (60%/40%) together with the corresponding distribution of the MSE. There, it can be seen that for some realizations there is a substantial error, and even the mean of all 100 realizations shows significant deviations from the true curve in some regions. But most importantly, the variance of the predicted data is quite large. Therefore, based on a single experiment, with the holdout method, it is difficult to judge the quality of the prediction.


```

1 import numpy as np
2
3 def train_test_split(X, y, split=0.6, seed=None):
4     """Splits a dataset given by X and y into training and testing data
5
6     This function uses numpy's function 'shuffle' and preserves the order of elements in X by
7     randomly picking elements and "moving" those to the test data set. If you also want the
8     two data sets themselves to be shuffled you have to do this afterwards.
9
10    :param X: feature data (a 1D numpy array)
11    :param y: target data (a 1D numpy array)
12    :param split: percentage of training data (e.g., 0.6: 60% of the DS is used as training data)
13    :param seed: either None or an integer used as seed from the random number generator
14    :returns: X_train, y_train, X_test, y_test
15    """
16    assert (X.ndim == 1) and (y.ndim == 1)
17
18    # total number of records and number of points for training
19    n_records = y.size
20    n_training_records = int(np.round(split * n_records))
21
22    # get random indices for the data used for learning
23    mask = np.empty(n_records, dtype=bool)
24    mask[:n_training_records] = True
25    mask[n_training_records:] = False
26    rng = np.random.default_rng(seed)
27    rng.shuffle(mask)
28
29    # get training data
30    X_train = X[mask]
31    y_train = y[mask]
32
33    # ... and what's left is used for testing
34    X_test = X[~mask]
35    y_test = y[~mask]
36
37    return X_train, y_train, X_test, y_test

```

Python Listing 16.1 An implementation of the train-test split

Things to Remember about the Holdout Method (or just: train/test split)★

- easy to implement
- fast in terms of computation (only one pass needed)
- can be strongly dependent on the random choice of the partitioning (depending on structure or variance of the data and in particular for small sample sizes)
- size of the training dataset is significantly reduced $\leadsto$ often, test error might get overestimated
- gives only a rough estimate of the true performance of a model

In Python Listing 16.1, a function for the train-test split is shown, which takes a dataset in form of a one-dimensional X and y array

16.2.3 Leave-One-Out Cross-Validation

The biggest shortcomings of the holdout method are (i) a dependency on a selection bias, (ii) the dependency on the structure of the dataset in case of nonuniform sampling and the significant reduction of data records available for training. The *leave-one-out cross-validation* (**LOO CV**) tries to remedy some of these problems through an “extreme” partitioning strategy:

1. Keep only a single data point for validation and use all other $n - 1$ observations for training the model.
2. Use the trained model for making a prediction for the single, left out data point and compute the error for this point, e.g., the **MSE** for a single point (which is the squared difference).
3. Perform the split of the dataset, followed by training, prediction, and error calculation altogether n -times until all data records were once “left out” as validation points.
4. If each experiment results in the MSE_i , then the resulting cross-validation error is

$$\text{CV} = \frac{1}{n} \sum_{i=1}^n \text{MSE}_i \quad (16.2)$$

Figure 16.5 shows the first five of these experiments. As compared to the holdout method, the **LOO CV** exhibits a strongly reduced bias because each experiment is trained on virtually the entire dataset. Additionally, this measure does not contain any random effects since all possible partitions are considered and reflected in the average cross-validation estimate. While the holdout method gives a different value whenever it is computed (due to the randomness of the train/test split, cf. Fig. 16.4), the **LOO CV** is a deterministic method that can only give as many different values as there are records in the dataset. For least squares regression using polynomial functions, there exists a formulation for the averaged CV that effectively has the same computational cost as the holdout method [3]. However, this cannot be generalized to arbitrary regression or classification methods; for those, there are still n experiments to be performed, which is why the computational cost is increasing in proportion to the number of observations.

Things to Remember about the leave-one-out cross-validation



- relatively easy to implement
- for n samples, n experiments need to be performed $\leadsto$ high computational cost
- deterministic cross-validation error, no influence from random sampling

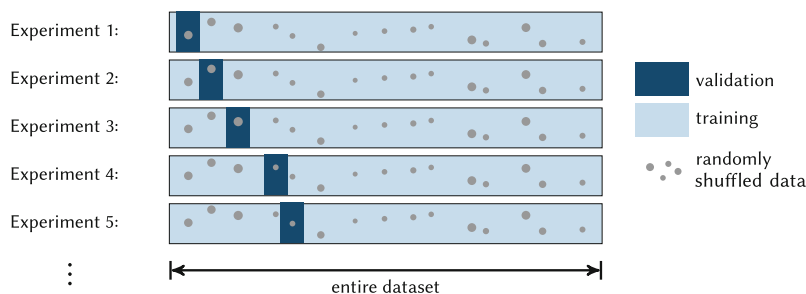


Fig. 16.5 Shown are the first five experiments for the **LOO CV** strategy. During each experiment, only one data point is used for validation, the rest is used for training, resulting in an as large as possible training dataset

16.2.4 *k*-Fold Cross-Validation

k-fold cross-validation (***k*-fold CV**) combines the advantages from both the holdout method and leave-one-out cross-validation: it creates a sequence of different training/validation partitions—the so-called *folds* and based on these, computes an average cross-validation error. The validation partition contains, unlike in the **LOO CV** method, more than one observation. Here is a brief summary of the strategy:

1. Split the whole dataset into k partitions by randomly assigning data points to the partitions. Fold number i consists of partition i , used as validation set; all remaining partitions are used together as training set.
2. Train the model for with training fold i and based on its predictions compute an error measure for this fold i , e.g., the **MSE**.
3. Perform the training, prediction, and error estimate for all k folds.
4. If each experiment results in the MSE_i then the resulting cross-validation estimate is

$$\text{CV} = \frac{1}{k} \sum_{i=1}^k \text{MSE}_i. \quad (16.3)$$

A sketch of this method is shown in Fig. 16.6 for five different partitions and folds. The holdout method corresponds to the evaluation of a single fold. Obviously, in the limit case of n observations and $k = n$ folds, this approach is identical to **LOO CV** introduced above. Comparing the two methods, it turns out that often the resulting **MSEs** is very close. For practical purposes, $k = 3 \dots 10$ are reasonable number of folds where $k = 3$ is feasible for small dataset and $k = 10$ for larger ones. For very small dataset it is important to use a value of k that is a divider of the number of observations such that all partitions have the same number of data points.

How to decide on whether we deal with a small or large dataset and the fold sizes are representative? A first impression can be obtained by calculating summary

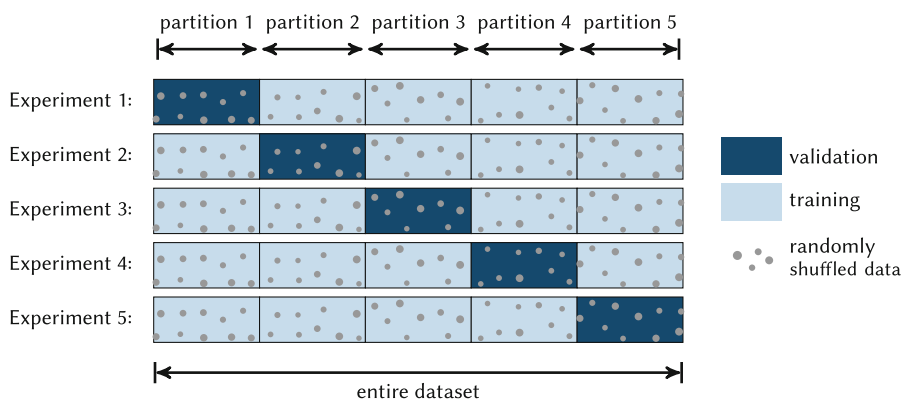


Fig. 16.6 This diagram shows the strategy for a 5-fold cross validation: The complete dataset (symbolized by the “dots” in a row) is split into five equal amounts of data, the *partitions*. During five separate experiments, each of these five partitions is used as validation dataset while the remaining four datasets are used for training. The partitioning for each experiment stays the same (note that in each experiment the position of the “data dots” is not changing)

statistics and observing (i) how pronounced the variance of the **MSEs** is and (ii) how the **MSE** compares to that of the whole dataset.

As compared to the **LOO CV**, the computational cost is significantly lower. Nonetheless, training a model or, to a lesser extent, making predictions on very large datasets four or eight times might not always be feasible. In these cases, the holdout method might be the only choice.

Things to Remember about the k -fold cross-validation



- more complex to implement
- for k folds, k experiments need to be performed $\leadsto$ medium computational cost, does not scale badly with the number of observations
- non-deterministic cross-validation error as the observations are randomly assigned to the folds, but it is a robust performance estimate

16.2.5 Repeated k -Fold Cross-Validation

The idea behind this method is to repeat a single **k -fold CV** and to analyze the variation among the repetitions. In between each repetition, the data are randomly shuffled so that for each repetition, different samples are contained in the folds. Then, one of the two different cases can be expected: (i) The model are “well-behaved” and stable such that for the same model parameters, small variations in the training dataset still give the same prediction for the validation dataset. (ii) If

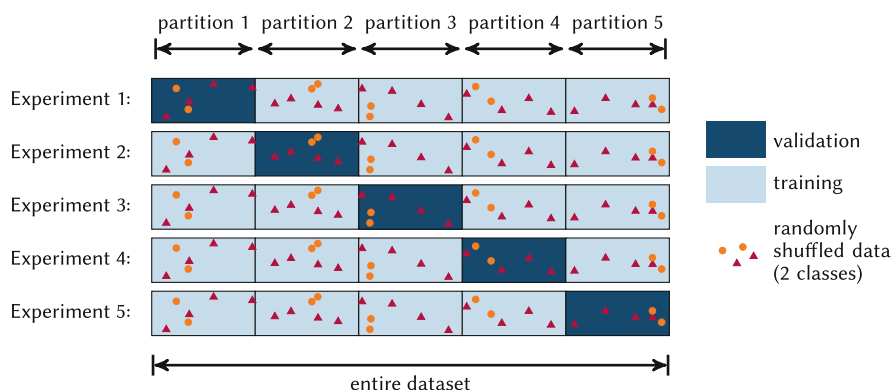


Fig. 16.7 The stratified 5-fold cross validation is a method for handling classification problems with class imbalance. *Stratified* corresponds to the fact that each fold contains samples with identical class statistics. This is symbolized by the two different markers: each fold has the same number of each marker, and the ratio between “triangle” examples and “circle” examples is the same as in the entire dataset. Apart from this all of the above concerning the *k*-fold CV applies

a model has a high sensitivity with respect to small variations in the training data, this also results in more pronounced variations of the predictions across different repetitions. The aspect of *model stability* can be investigated through repeated *k*-fold CV.

16.2.6 Stratified *k*-Fold Cross-Validation

So far, all discussed methods can be applied to both regression problems as well as to classification problems (in the latter case, instead of the *MSE* suitable error measures need to be employed, such as the relative number of misclassified observations).

A specific problem occurring only for classification problems is the problem of *class imbalance*, i.e., strong variations of the number of examples per class (for regression problems a similar phenomenon can occur if different regions have a different density of data, e.g., see the distribution of energy for the “MDS-3 (Cahn-Hilliard Model)” dataset). *Stratified sampling* implies that each fold contains approximately the same distribution of samples per class as contained in the total dataset. This is an important aspect as the goal is to make predictions for the whole population based on a set of samples. Figure 16.7 gives a visualization of the method

16.2.7 Bootstrap

So far, in all above approaches, it was important to split the dataset into disjunct partitions. This is the most obvious difference to the *bootstrap*.² For choosing data points for the training dataset samples are randomly drawn *with replacement* from the entire dataset. This may result in a *bootstrapped* dataset with a number of identical data points, compare the example in Example 16.3.

Meaning of “bootstraps”



Originally, in the literature, the phrase *by one's own bootstraps* denotes the self-starting process that is able to proceed by itself and without any extra help.

The bootstrap of the original dataset $\mathcal{D}$ can be performed a number of times, say n -times, and each time one obtains a different dataset $\mathcal{D}_n^*$. Each of the bootstrapped datasets can now be used to train a model, resulting altogether in an *ensemble of models*. Within this ensemble, the average model predictions can now be obtained from

$$\langle y \rangle = \frac{1}{n} \sum_{i=1}^n y(\mathbf{x}^i) \quad \text{where } \mathbf{x}^i \text{ denotes all } x \in \mathcal{D}_i^* \quad (16.4)$$

Then, e.g., the so far not used samples can be used for validation purposes. The training datasets are then *overlapping* subsets of the entire dataset.

Bootstrapping is a method originally used in statistics and belongs to the class of *resampling* methods. It was first described in 1979 by Efron [2], a statistician from Stanford University, as an enhancement and generalization of the *Jackknife method*. Both methods are also used for “regular” random sampling in statistics and there in particular for estimating the median, bias, and variance of a dataset.³

² In the ML community it is often just “bootstrap”, “bootstrapping”, or “the bootstrap” but originally called “the bootstrap method”

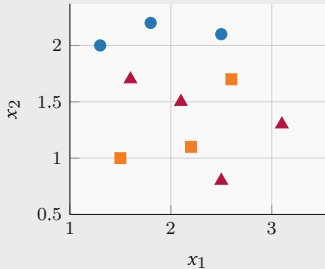
³ A curious side note about the origin of the name “Bootstrap” for this method: at the end of [2], B. Efron acknowledged discussions about the naming of his method: “[...] I also wish to thank the many friends who suggested names more colorful than Bootstrap, including Swiss Army Knife, Meat Axe, Swan-Dice, Jack-Rabbit and my personal favorite, the Shotgun, which, to paraphrase Tukey, ‘can blow the head off any problem if the statistician can stand the resulting mess’.”

Example 16.3 (Bootstrapping a Dataset of Categorical Data.)

✖

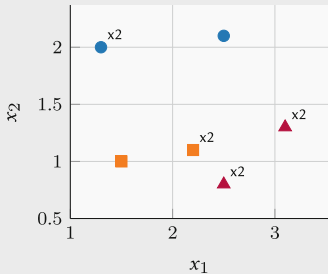
Assume a training dataset, $\mathcal{D}$, which consists of 10 records. Each record is governed of two real valued features, x_1 and x_2 , and a class label y as output where y is one of the three marker symbols. The dataset is shown in the table below; additionally, the figure shows a visualization of the dataset.

#record <i>n</i>	Input		Output
	x_1	x_2	y
1	1.6	1.7	▲
2	2.2	1.1	■
3	3.1	1.3	▲
4	2.5	0.8	▲
5	2.6	1.7	■
6	1.3	2.0	●
7	1.5	1.0	■
8	2.1	1.5	▲
9	1.8	2.2	●
10	2.5	2.1	●



The bootstrapped dataset $\mathcal{D}^*$ is obtained by drawing 10 samples from $\mathcal{D}$ with replacement, i.e., any record could be drawn multiple times. One could now use, e.g., the `numpy` function `random.randint(1, 11, 10)` to draw 10 numbers from the half-open interval $[1, \dots, 11)$ which might give the to be drawn number of sample $n = \{6, 4, 6, 7, 2, 3, 2, 3, 10, 4\}$. There, the sample number 6 occurs twice, sample number 7 occurs only once, sample number 8 was not drawn. This is shown in the following, already compacted table in which the last column gives how many times a sample was drawn.

#record <i>n</i>	Input		y	Output
	x_1	x_2		repetition
2	2.2	1.1	■	2x
3	3.1	1.3	▲	2x
4	2.5	0.8	▲	2x
6	1.3	2.0	●	2x
7	1.5	1.0	■	1x
10	2.5	2.1	●	1x



The scatter plot on the right indicates the number of drawings of the same sample by the small superscripts, e.g., the triangle with the superscript $\blacktriangle^{x_2}$ denotes two “coinciding” data points. The new, bootstrapped dataset is similar to the original one in some aspects but, e.g., misses some data points. Therefore, it has slightly

16.3 Baseline Models

Sometimes, we need to get a quick and rough estimate of what “good performance” actually means. A baseline or a *baseline model* is usually a very simplistic model or estimation that serves as a rough orientation and that can be created without having to invest a significant amount of time. This does not necessarily have to be a non-ML model, it can also be any other reference model that is already available. However, the emphasize is on simplicity and certainly excludes an involved training process, let alone hyperparameter optimization. Typically, a good model should perform better than the baseline model.

A good first step is, to take a look at the data, maybe perform a *exploratory data analysis* (EDA), which is also useful for determining a performance metric.

Baseline Models for Classification

As an example, consider a classification problem with two classes each of which are equally likely. The simplest strategy is then to use a classifier that randomly assigns class label to new data. This will give the correct results in 50% of all cases, and the base line accuracy would be 50%. Any ML that we train should perform better than that; if this is not the case, it might be a hint that there is something wrong with the implementation. This approach can also easily be generalized to multi-class problems. If the dataset has a class imbalance, then one can use the most frequently occurring class label and from that compute accuracies for the whole dataset.

Baseline Models for Regression

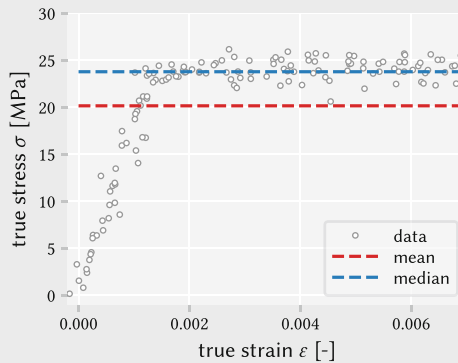
A very simple approach for obtaining a regression baseline model is to use some of the information about the expected value of the data distribution itself. E.g., using the mean (or median) of all data records will give us at least a rough idea about a range in which some of the data will be located. Another approach is to use any constant value that is chosen, e.g., based on our domain knowledge of a problem. An example is shown in Example 16.4.

Example 16.4 (Regression Baseline for the Tensile Test Dataset)



The goal is to create a regression model for the MDS-1 (Tensile Test) dataset that is able to predict the stress–strain curve. We only focus on the high-temperature regime of 600 °C and strains up to 0.007. A simple baseline model is to take the mean or the median of all stress values as shown in the figure.

(continued)



As can be seen, the median has a high value since there are many examples in the plastic regime while the mean is more balanced. The MSE value for the median baseline is 59, for the mean baseline it is 46.

If we implement a simple linear regression model, then we get a MSE value of 24—not great but better than the baseline, which also gives us some confidence that our implementation is probably correct.

References

1. L. Breiman. *Machine Learning*, 45(1):5–32, 2001. DOI <https://doi.org/10.1023/a:1010933404324>.
2. B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1), Jan. 1979. DOI <https://doi.org/10.1214/aos/1176344552>.
3. G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer New York, 2013. DOI <https://doi.org/10.1007/978-1-4614-7138-7>.

Part IV

Artificial Neural Networks and Deep Learning

From the Perceptron to Artificial Neural Networks

17

A single neuron in the brain is an incredibly complex machine that even today we don't understand. A single 'neuron' in a neural network is an incredibly simple mathematical function that captures a minuscule fraction of the complexity of a biological neuron.

Andrew Ng (born 1976), American computer scientist and technology entrepreneur

17.1 A First Model of a Neuron

The biological neuron is often used to explain the motivation behind the computational model(s) of neurons. While, in the early days of the computational neurons, it was certainly true that they were designed to mimic biological neurons, hoping to create an “artificial intelligence”. However, later on, with the emergence of more powerful models this was changed, and it is no longer the goal to imitate the biological neuron (unless we are diving into “neuromorphic computing”). In the following, we trace the historic development *artificial neural networks* (ANNs) and thereby understand how, why, and when these single neuron models became the “launch pad” for today’s powerful and complex deep learning architectures.

17.1.1 The Biological Neuron

The name *neuron* was coined in the late nineteenth century by the anatomist HEINRICH WILHELM WALDEYER [11], who already speculated 1881 that biological neurons might be the basic building blocks of the nervous system. Generally, neurons are nerve cells for processing and transmitting chemical and electrical

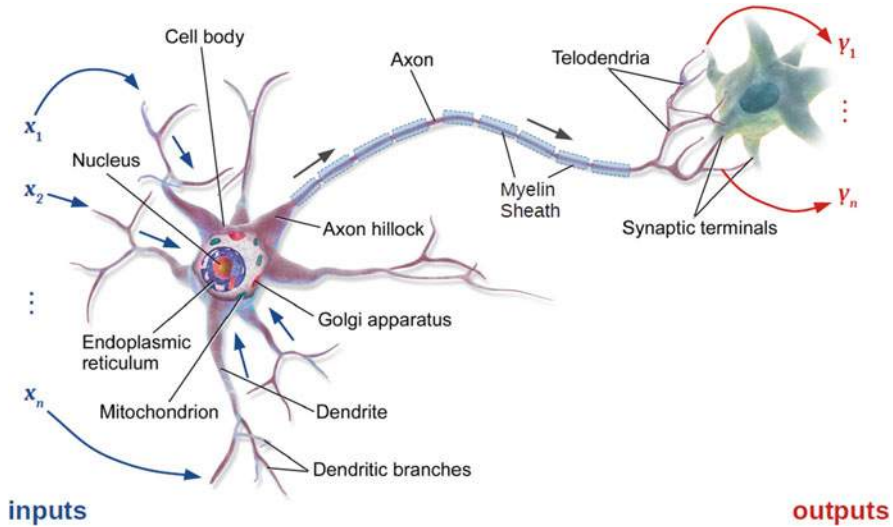


Fig. 17.1 Anatomy of a multipolar neuron: left is the presynaptic cell, right is the postsynaptic cell. The input signal arrives at dendrites/dendritic branches, is processed in the soma/cell body, which passes the signal through the axon to the synapses or other neuron's dendrite (indicated by the red arrows). (Image adapted from [2], licensed under CC BY 3.0 license¹)

signals. Typically, a large number of neighboring neurons (even up to 200,000) are interconnected with each other and thereby exchanging information.

A typical neuron consists of a cell body, the *soma*, whose purpose is to process the received signals. The links between the neurons consist of *dendrites* (from the Greek δένδρον, pronounced dendron = “tree”) and the *axon* (from the Greek ἄξων, pronounced axōn = “axis”); through the dendrites, the signal is received from other neurons, and the *axon* transmits the output. The *synapse* is located at the end of the axon and acts as connection point between two neurons; mostly, it “translates” electrical signals into chemical signals, the so-called *neurotransmitters*. A particular type with one axon and two or more dendrites/dendritic branches is called *multipolar neuron*, see the sketch in Fig. 17.1. Neurons are most commonly encountered in the central nervous system and spinal cord of almost all animals.

Neurons react to electrical signals, or more precisely to the change of voltage over time. If this change within a small time span is sufficiently large, then the neuron “fires” by emitting an electrochemical pulse. This pulse is transmitted by the axon, getting weaker with increasing distance, and eventually reaches the synaptic connection where it causes an excitatory or inhibitory response. The particular kind of response can be modified in a number of different ways. Hence, the signal transfer can be adapted by certain types of stimuli—a process that is called

¹ <https://creativecommons.org/licenses/by/3.0/deed.en>

“synaptic plasticity”.² It was only about 10 years after the neuron got its name from H. W. WALDEYER that EUGENIO TANZI, an Italian neuropsychiatrist, hypothesized that during repetitions of a learning process, a repeated neuronal activity was initiated that reinforces the current synaptic connections and makes the crossing of the signal easier [1]. Further information on the biological process from the point of view of computer science can be found in, e.g., [6].

17.1.2 Introducing...the McCulloch-Pitts Model

Two persons are responsible for the *McCulloch-Pitts* (MCP) model, which is named after them: WARREN. S. MCCULLOCH and WALTER PITTS. The MCP model [3] was developed in 1943 and is also called *Threshold Logic Unit* (TLU) model. It was the first computational model designed to resemble or imitate a biological neuron. Its main goal was to make a decision $y \in \{0, 1\}$, i.e., to answer a question with “yes” or “no” based on some input $x_i \in \{0, 1\}$. There, the values 0 and 1 typically correspond to the Boolean values `False` and `True`.

Example 17.1 (A decision that the MCP model could make)



A decision made by the MCP model might be answering the question “Is it going to rain in the afternoon?”. The decision could be based on the input information x_1 : “The sun is shining right now” (“yes” or “no”); x_2 : “The air humidity is above 70%” (“yes” or “no”). The output is then $y = 1$ (“yes, it will rain in the afternoon”) or $y = 0$ (“no, it will not rain in the afternoon”).

In analogy to the example of the biological neuron in Fig. 17.1, the flow of data would take place from left to right: the input values x_i from the left (dendritic branches) are processed or *aggregated* by the cell body and transferred to the right through the axon to the synapse, which then fires (or not). The MCP model is based on the assumption that the time for the information transfer along the axon is negligible. The aggregation of the input data x_i happens in the form of an (unweighted) sum while the fact that the synapse fires only above a certain threshold is modeled through a nonlinear *activation function* φ . Mathematically, the aggregation of the input happens in the *aggregation or transfer function* $a(\cdot)$,

$$a(\mathbf{x}) = \sum_{j=1}^n x_j, \quad (17.1)$$

² This notion is the reason why an online search for the word “plasticity” not only shows materials science results on *plastic deformation of metals* but also shows—an equally large number of—seemingly unrelated search results from the field of neuro science.

yielding the *net input* of the unit. The binary output y is then obtained by applying the activation function φ to the aggregated data

$$y = \varphi(a(\mathbf{x})) . \tag{17.2}$$

In the case of the MCP model, the activation function is taken as a unit step function with the threshold value θ ,

$$\varphi(a) = \begin{cases} 1 & \text{IF } a \geq \theta \\ 0 & \text{ELSE ,} \end{cases} \tag{17.3}$$

as depicted in Fig. 17.2a. It is important that the comparison is $\geq$ and not just $=$ as can be seen in the next section. A typical sketch of the MCP model is shown in Fig. 17.2b; Fig. 17.2c shows the commonly used short version.

Words of advice

In a number of publications and online resources, the MCP model is not always presented correctly: there, the input values are often individually weighted with factors, which, according to the publication of McCulloch and Pitts from 1943 [3], is not correct. In the original MCP model, the inputs are not individually weighted and represent Boolean values.

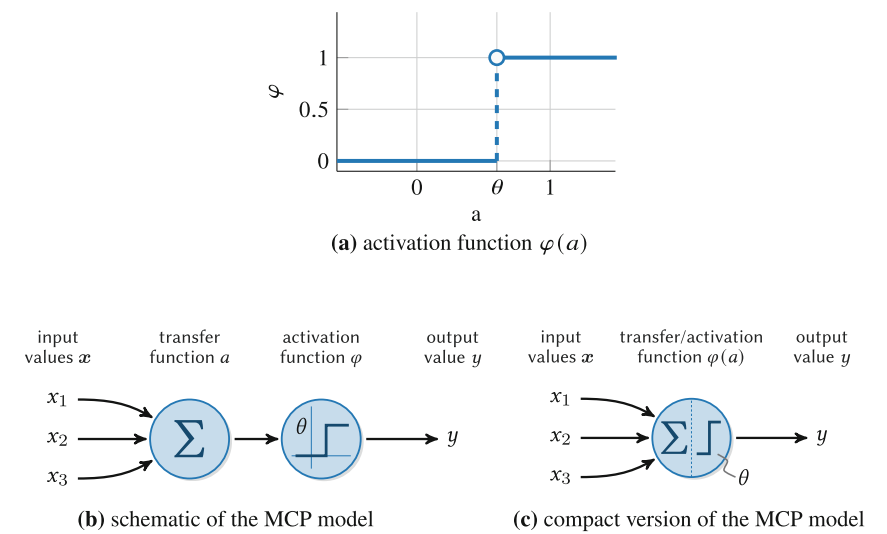


Fig. 17.2 The MCP model takes the input x_i , sums them up in the transfer function, and “makes a decision” with the activation function. For this model, the activation is the unit step function shown in (a) and has a threshold θ , which is also indicated in the detailed symbolic representation of the model in (b). (c) show a compact representations of the model for three inputs

Table 17.1 Representation of the logical AND by a MCP model with two inputs and a threshold value $\theta = 2$. The two output columns show the model output as well as the true output $x_1 \wedge x_2$

Input		Data Processing			Output
x_1	x_2	$a(\mathbf{x}) = x_1 + x_2$	$a \geq \theta ?$	$y = \varphi(a)$	$x_1 \wedge x_2$
0	0	0	$0 \geq 2 ?$	0	0
1	0	1	$1 \geq 2 ?$	0	0
0	1	1	$1 \geq 2 ?$	0	0
1	1	2	$2 \geq 2 ?$	1	1

17.1.3 Boolean Decision Problems

Many problems can be turned into a *Boolean decision problem*, i.e., decisions where the input data and the outcome (i.e., the decision) can be reduced to statements with “yes” or “no” answers. In other words, the output data are *categorical data* and thereby solve binary classification problems as there are only two possible outcomes (0 or 1), which, however, can be interpreted, e.g., as “the testing specimen shows failure” or “the testing specimen is intact.”

In the MCP model, both the input values x_i and the output y can only take the values 0 or 1, which can then be interpreted as the Boolean values $\text{True} \equiv 1$ and $\text{False} \equiv 0$. Therefore, the model has the potential to solve certain Boolean decision problems, hence the alternative name “Threshold Logic Unit” (TLU).

We will now investigate if the MCP model would be able to represent the fundamental Boolean operations AND, OR, and NOT, and thus if the model would be able to make such a decision. A numerical, automated strategy for finding the most suitable value of θ is a different task and will be addressed for subsequent, more complex models below.

• Logical AND operation

This logical operator is represented using the $\wedge$ symbol. For example, for two input values x_1 and x_2 , it is $x_1 \wedge x_2$. The result of the logical AND is True if and only if all input values are True . For a neuron with two inputs, this is the case if the threshold value is chosen as $\theta = 2$; for a logical unit with n inputs, θ must be set to n .

This can be easily seen in Table 17.1 by comparing the column “ $a = x_1 + x_2$ ” with the column “ $x_1 \wedge x_2$ ”: only when a changes from 1 to 2 the output should change from 0 to 1 and thus $\theta = 2$. Thus, the MCP model is able to represent the logical AND function.

• Logical OR operation

This logical operator is represented using the $\vee$ symbol. For example, for two input values x_1 and x_2 , the OR function is $x_1 \vee x_2$. The result of the logical OR is True if at least one of the input values is True . For a neuron with two and more inputs, this is the case if the threshold value is chosen as $\theta = 1$ as shown in Table 17.2.

Table 17.2 Representation of the logical OR by a MCP model with two inputs and a threshold value $\theta = 1$. The two output columns show the model output as well as the true output $x_1 \vee x_2$

Input		Data Processing		Output
x_1	x_2	$a(\mathbf{x}) = x_1 + x_2$	$a \geq \theta ?$	$y = \varphi(a)$
0	0	0	$0 \geq 1 ?$	0
1	0	1	$1 \geq 1 ?$	1
0	1	1	$1 \geq 1 ?$	1
1	1	2	$2 \geq 1 ?$	1

Thus, the MCP model is able to represent the logical OR function.

- *Logical NOT operation*

This logical operator is represented using the $\neg$ symbol. For example, for an input x , the NOT function is $\neg x$. The result of the logical NOT function is True if the input is False (and vica versa). The NOT function is the counterpart to an *inhibitory synapse*.

For representing the NOT function, though, it would be necessary to weight the input with -1 and adjusting threshold alone is not sufficient, which can be shown as follows: Assume that $x = 0$. It follows:

$$x = 0 \Rightarrow y = \neg x = 1 \quad (17.4)$$

As for a single input, the transfer function becomes the identity function, $a(x) = x$, we are looking for the θ that fulfills the inequality

$$a(x) = x = 0 \geq \theta \Rightarrow \theta \leq 0. \quad (17.5)$$

Assuming $x = 1$ it is $y = 0$, and we are looking for values of θ that *do not* satisfy the inequality, $a = x \geq \theta$, such that y evaluates to 0 (False):

$$x \not\geq \theta \Rightarrow \theta > 1. \quad (17.6)$$

As it is impossible to fulfill Eqs. 17.5 and 17.6, the NOT operation cannot be represented by the MCP model.

Another Boolean operator is the XOR operator, which is represented by the symbol $\oplus$ or also by $\dot{\vee}$. Its result is True if and only if $x_1 \neq x_2$, i.e., one input is True while the other one is False. The XOR operator can be replaced by a combination of AND and OR operations together with the NOT operator. As the MCP model cannot represent the NOT operator it is also not able to represent the XOR operator. Thus, a more flexible model is required that, as it turns out, needs to at least be able to “weight” the inputs with individual and adaptable factors.

Things to Remember about the McCulloch-Pitts (MCP) model:

- The MCP (or TLU) model is the simplest possible model motivated by the biological neuron.
- The input data have the values 1 (True) or 0 (False) and are aggregated (summed up); ...
- ...if this value is larger than a threshold value, the output is 1 (or True); otherwise, it is 0 (or False).
- The model can only make the Boolean decisions “AND” and “OR” while operations “NOT” and “XOR” cannot be represented by the model.

17.2 The Rosenblatt Perceptron

After the pioneering development of the McCulloch-Pitts model, it took another 15 years until the *perceptron* as an enhancement of the MCP model was developed by the psychologist Frank Rosenblatt: in 1957, he published the report “The Perceptron: a perceiving and recognizing automaton,” whose aim he specified as to “[...] *formulate a brain analogue useful in analysis*” [7]. In this publication, the word “perceptron” originally denoted an interconnected network *structure* and not, as it is typically understood today, the individual unit. The first hardware implementation was the Mark I perceptron, which was assembled in 1957 at Cornell Aeronautical Laboratory in Buffalo, New York. It was a pure hardware realization of the perceptron “network” where weights were represented as potentiometers, which were adjusted by electro-motors. A short overview about the work of Rosenblatt can be found in [5].

The name *perceptron* was derived from the fact that the inputs consisted of 20×20 cadmium sulfide photo cells for “perceiving” a 400 pixel image; the output consisted of signal lights that could be activated when a threshold—similar to the MCP model—was exceeded. Activation was mutually exclusive such that two signals would not be activated at the same time. Further details can be found in the original publication [7].

In the following, we introduce one of such units and will subsequently denote this as a *perceptron*; if more than one of these units are combined, then we will denote this as a network of perceptrons.

17.2.1 The Mathematical Model of the Rosenblatt Perceptron

The first main difference in the MCP model is that the inputs x are weighted with individual values. The second difference is that the input values can take real values while for the MCP they could only be Boolean, i.e., 0 or 1. Additionally, a *bias* acting as offset for the weighted and aggregated inputs was introduced. With this

the transfer function takes the shape of a linear combination,

$$a(\mathbf{x}) = b + \sum_{j=1}^n w_j x_j, \quad (17.7)$$

where the w_j are variable weights and b is the bias. This equation can be reformulated if the bias is also interpreted as weight such that the new input and weight vector looks

$$\tilde{\mathbf{x}} = [1, x_1, \dots, x_n]^T \quad \text{and} \quad \tilde{\mathbf{w}} = [b, w_1, \dots, w_n]^T. \quad (17.8)$$

With this, the transfer function can be rewritten using the scalar product between two vectors, which is computationally efficient as already mentioned by Rosenblatt:

$$a(\mathbf{x}) = \tilde{\mathbf{w}}^T \cdot \tilde{\mathbf{x}}. \quad (17.9)$$

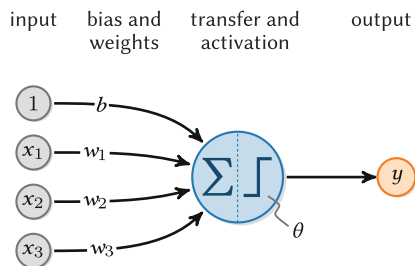
The activation function is again the unit step function with the threshold value θ (as in Eq. 17.3),

$$\varphi(a) = \begin{cases} 1 & \text{IF } a \geq \theta \\ 0 & \text{ELSE} \end{cases}. \quad (17.10)$$

A schematic representation of a perceptron with three inputs is depicted in Fig. 17.3.

Note that b does not always occur in the above equation, as it can be “brought to the other side” of Eq. 17.10, and then it just contributes to the threshold (alternative, the threshold can also be included in the bias). The mathematical model formulation contains the original MCP model as a special case, as can be seen by setting $\theta = 0$ and all $w_i = 1$:

Fig. 17.3 Schematic of the Rosenblatt perceptron with three adjustable weights w_1, w_2 and w_3 and an adjustable bias b . θ is the threshold parameter of the activation function



$$a(\mathbf{x}) = [0, 1, \dots, 1] \cdot \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^n x_i. \quad (17.11)$$

This is the formulation for the transfer function of the MCP model; the activation function is the same for both models.

Python Code for the Perceptron Model

The above can be easily implemented in Python, if we assume that the inputs and weights are all numpy arrays. For this, the code of the perceptron model is shown in Python Listing 17.1. The function `predict` makes use of the fact that the

```

1 import numpy as np
2
3 def predict(x, weights, bias, threshold):
4     # Evaluate the transfer function: we're using numpy's scalar product `dot`
5     a = bias + np.dot(x, weights)
6
7     # Activation function: check if the perceptron fires
8     # (For 1 data record a is a scalar and we could simply use
9     # `y = int(a >= threshold)`. For multiple data records, a is a 1D array
10    # --> use numpy's `where(condition, x, y)` function which returns x if
11    # the condition holds and otherwise it returns y.)
12    y = np.where(a >= threshold, 1, -1)
13
14    return y

```

Python Listing 17.1 Implementation of the Rosenblatt perceptron for making a prediction. Here, the prediction will be a value of either 0 or 1. However, it is also possible to use, e.g., -1 and $+1$ (see Sect. 17.2.5 for an explanation)

aggregation function can be computed as a scalar product between $\mathbf{x}$ and $\mathbf{w}$. In the code this is done using Numpy's function `dot` which takes two vectors (or rather: 1D Numpy `array`'s) as arguments. The example usage is for two inputs; for more inputs only the number of elements in the weights and input vectors need to be adapted.

Example 17.2 (Prediction with the perceptron model for two inputs) ✖

Assume a perceptron with two inputs and a threshold of $\theta = 1$. Furthermore, given are the weights $w_1 = 1.5$ and $w_2 = 2$ together with the bias $b = 0.5$. We can then make predictions, e.g., for the input vector $\mathbf{x} = [1, 1]$:

(continued)

```
In [1]: weights = np.array([1.5, 2])
        bias = 0.5
        x = np.array([1, 1])
        y = predict(x, weights, bias, threshold=1.0)
        print(f'x = {x}, y = {y}')
```

```
Out [x]: = [1 1], y = 1
```

`x` contains a single data record with two features. For several data records, `x` has to be a two-dimensional array with features in columns and records in rows.

17.2.2 Again: Boolean Decision Problems

As the MCP model is contained as a special case of the Rosenblatt perceptron, this enhanced model is still able to represent the OR and AND operations. We will now take a look, if the NOT and XOR operators can be represented:

Logical NOT Operation

Starting with the definition of the NOT operator

$$x = 0 \Rightarrow y = \neg x = 1 \quad (17.12)$$

$$x = 1 \Rightarrow y = \neg x = 0 \quad (17.13)$$

and the transfer function for a perceptron with a single input x

$$a(x) = b + wx \quad (17.14)$$

we are looking for the θ that fulfills the inequality

$$a(x) = b + wx \geq \theta \quad (17.15)$$

For $x = 0$ this inequality should be True, i.e., it is $b \geq \theta$. As we have two equations (or rather inequalities) and three unknown variables, θ , b and w , we need to fix at least one of them, and we choose the threshold as $\theta = 0$. For $x = 1$ the inequality Eq. 17.15 should *not* be fulfilled and it reads

$$b + w \not\geq 0 \quad (17.16)$$

$$\Leftrightarrow b + w < 0 \quad (17.17)$$

$$\Leftrightarrow w < -b. \quad (17.18)$$

Table 17.3 Representation of the logical NOT by a perceptron model with one input. The parameters are chosen as $\theta = 0$, $b = 0$ and $w = -1$. The two output columns show the perceptron output and true output

Input		Data Processing		Output
x_1	$a(x) = b + wx$	$a \geq \theta$?	$y = \varphi(a)$	$\neg x$
0	0	$0 \geq 0$?	1	1
1	-1	$-1 \geq 0$?	0	0

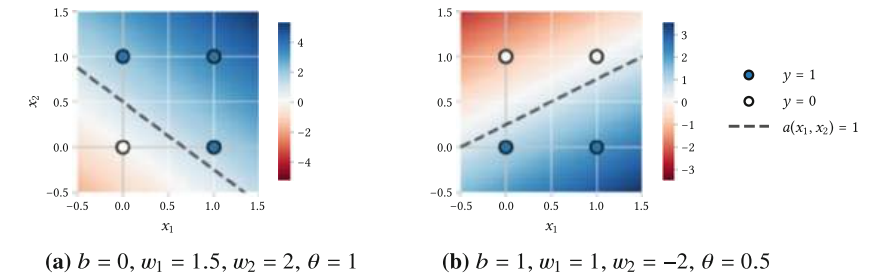


Fig. 17.4 Visualization of response of the perceptron for two different parameter sets. The blue and red color shades show the value of the transfer function $a = b + x_1 w_1 + x_2 w_2$, the dashed line is the “decision boundary” that separates points with values below the threshold θ from those that have values above the threshold. The markers show four particular points; blue filled markers denote those input values x_1, x_2 for which the perceptron would be activated ($y = 1$). (a) $b = 0, w_1 = 1.5, w_2 = 2, \theta = 1$. (b) $b = 1, w_1 = 1, w_2 = -2, \theta = 0.5$

Choosing $b = 0$, it follows that the weight w should be negative, e.g., $w = -1$. With these parameters we get the response as summarized in Table 17.3.

Thus, the perceptron is able to represent the Boolean operation NOT—and therefore potentially also to *learn* such a response (even though so far, it is not clear how “learning” is done for such models).

Logical XOR Operation

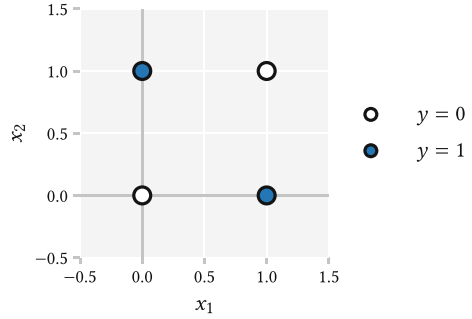
The mathematical proof that the XOR operation is beyond the capability of the perceptron is left as an exercise. Instead of a formal proof we will use a geometrical interpretation in Sect. 17.2.3 to explain why the XOR operation cannot be represented.

17.2.3 Geometric Interpretation

The mathematical model of the perceptron for two inputs can be visualized and interpreted based on geometrical relation. Figure 17.4 shows as a two-dimensional contour plot the values of the transfer function a as a function of x_1 and x_2 for two particular parameter sets b, w_1, w_2 , and θ .

If the inequality of the activation function is further restricted to an equality, i.e.,

Fig. 17.5 If the perceptron would be able to represent the logical XOR operation, then only the points shown as dark markers would have output values of $y = 1$ as shown. However, there exists no straight line that could completely separate the dark from the light markers



$$\varphi(a) = \begin{cases} 1 & \text{IF } a = \theta \\ 0 & \text{ELSE,} \end{cases} \quad (17.19)$$

and we only consider the case $a = \theta$ then the set of equations can be simplified to

$$w_1 x_1 + w_2 x_2 + b = \theta \quad (17.20)$$

$$\Rightarrow x_2 = -\frac{w_1}{w_2} x_1 + \frac{\theta - b}{w_2}. \quad (17.21)$$

This is the equation for a line where the slope is $-w_1/w_2$ and the offset or intercept is $(\theta - b)/w_2$. It is called *decision boundary* as it separates the regions where the neuron fires ($\varphi = 1$) from those where it doesn't fire ($\varphi = 0$) (see Chap. 14 where the notion of the decision boundary was introduced in detail). In analogy, with three (or more) inputs, the separating line turns into a separating plane (or *decision hyperplane*). The decision boundary is shown as dashed line in Fig. 17.4. The visualization of $a(x_1, x_2)$ also reveals that a is a *linear* equation. This has implications for the type of problems that can be represented by the perceptron model: only *linearly* separable data can be learned and properly represented.

What does that imply for the XOR problem? To understand this, we take a look at the left plot of Fig. 17.4. There, only the marker at $x_1 = x_2 = 0$ has a value of $y = 0$, all other points indicate a value of $y = 1$. In terms of logical operations this would represent the OR operation. From this we also can directly see why there is some freedom in choosing the parameters as there exist a multitude of different lines, all of which separate the three blue markers from the white marker. The XOR operator has $y = 1$ if and only if there is exactly one of the input value that has the value 1, i.e., only $x_1 = 1$ and $x_2 = 0$ and $x_1 = 0$ and $x_2 = 1$ result in the output $y = 1$. Figure 17.5 visualizes this configuration.

If the perceptron would be able to represent (and eventually “learn”) the XOR Boolean decision, then there must exist a straight line that separates the points with $y = 1$ from those with $y = 0$. As the figure shows, this is not possible.

Even though, the perceptron is more powerful than the MCP model, it is still not able to make an “either-or” decision, and it is limited to classification problems of linearly separable data.

Historical Comments



In the original publication by ROSENBLATT [8] from 1958 most of the used notions were named in analogy or motivated by the (human) eye and our ability to detect visual patterns:

- What we today denote as *input nodes* was originally named *sensory units* (S-points), e.g., that of a retina which ...
- ... can be *excitatory* or *inhibitory* depending on if they increase or reduce the resulting signal—in our model this is represented by the sign of the respective weight.
- The signals are transmitted to the *association cells* (A-units)—our combined transfer/activation unit.
- The *response units* (R-points) are the output node(s), activated once the threshold is overcome.

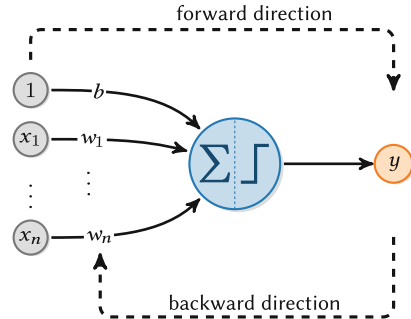
Already ROSENBLATT [7,8] used as variable for the threshold the Greek letter θ , which is why this is still the most common choice.

17.2.4 Supervised Learning with the “Perceptron Learning Rule”

As opposed to the MCP neuron, the concept of the Rosenblatt perceptron includes a feedback loop, where through the so-called *perceptron learning rule* the error is *back propagated*, allowing the weights and bias to be automatically adjusted. The “regular” evaluation of the perceptron, i.e., computing $y^{\text{pred}} = \varphi(\mathbf{x})$, is the *feed forward* step. Figure 17.6 shows the forward calculation direction and the backward direction where the error is “propagated” from the right to the left.

Back-propagation is a crucial component of any *deep learning* (DL) network and will then include the calculation of gradients, but we will come to that later. These are two notions also used in the deep learning networks introduced in one of the following chapters.

Fig. 17.6 The feed-forward consists of evaluating the transfer function and the threshold for given weights, bias, and input values. During back propagation, the weights are automatically adjusted based on the output of the perceptron



Binary Classification



The Rosenblatt perceptron and the MCP model are able to solve classification problems: based on some input data, they can make a decision, and their output data are of categorical type. It is a *binary classification* as there are only two possible outcomes (0 or 1), which, however, can be interpreted, e.g., as “the sample shows failure” or “the sample is intact.”

The perceptron learning rule is a simple approach through which by comparing the desired output with the current output the weights are iteratively changed by small amounts and in proportion to that difference. How strong the influence of the difference is on the updated values is controlled by a scalar factor, the *learning rate*, which we denote in the following by the Greek letter η . If the learning rate η is 0, the weights and bias should not be changed; equally, if the current output value is already the desired output value, the weights and bias should not be changed. Even though it will turn out that this learning rule is unsuitable for a robust learning strategy, the Rosenblatt perceptron is the first attempt of *supervised learning* in the context of early neuronal networks.

Perceptron Learning Rule

Assume that a number of training records in the form of the input features $\mathbf{x}^{\text{train}} = \{x_1^{\text{train}}, \dots, x_n^{\text{train}}\}$ and the response y^{train} are given, and the perceptron should be trained to be able to make predictions. Then, for the training the algorithmic steps shown in Algorithm 17.1 need to be performed.

Algorithm 17.1: Perceptron Learning Rule (for 1 Training Epoch)



1. Initialization:

- initialize the weights $w_1, \dots, w_n$ and bias b with zero
- choose a suitable learning rate $\eta \in (0, 1]$
- prepare a set of data records (e.g., rows from the data matrix)

2. **Feed-forward step:** Choose a new data record and make a prediction for $\mathbf{x}^{\text{train}}$ with the current values of the weights and bias:
 - a. compute the weighted sum $a(\mathbf{x}^{\text{train}}; \mathbf{w}, b)$
 - b. evaluate the activation function: $y^{\text{pred}} = \varphi(a; \theta)$

3. **Backward propagation:** use difference between predicted and true output to compute the update increments for weights and bias

$$\Delta \mathbf{w} = (y^{\text{train}} - y^{\text{pred}}) \mathbf{x}^{\text{train}} \quad \text{and} \quad \Delta b = y^{\text{train}} - y^{\text{pred}} \quad (17.22)$$

Update the weights and bias to obtain the new values

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \Delta \mathbf{w} \quad \text{and} \quad b \leftarrow b + \eta \Delta b \quad (17.23)$$

4. **GO TO step 2 and repeat** with a new record $(\mathbf{x}^{\text{train}}, y^{\text{train}})$ until all data records were used exactly once.

Epoch

Any full iteration, consisting of both forward steps and weight/bias updates for *all* data records, is called an *epoch*. A complete training process usually consists of several epochs where each new epoch starts with the updated weights and bias values from the previous epoch. Typically, for a very small learning rate, more epochs are required than for larger learning rates. Often, it is the goal to use a learning rate that is as large as possible so that only few epochs are required for training, which makes the training process fast. At the same, if the learning rate is too high, numerical instabilities will occur and the training process will not be able to converge.

Termination Criterion

The training is terminated either if an error is sufficiently small (see “error measures” below) or if a maximum number of training epochs is reached. The latter is helpful for avoiding that the algorithms encounter a situation where the error cannot be reduced sufficiently and where one would end up in an “infinite loop.” An obvious choice is termination if the error is zero. However, it can be formally proven that the perceptron learning algorithm only converges if the dataset consists of two linearly separable classes. Therefore, a more generous minimal error is often more useful and should be chosen based on the dataset and the application.

The Peculiarity of the Perceptron Learning Rule



The perceptron learning rule is somewhat unusual as it does an individual forward/update step for *each* of the training data records. Therefore, in particular for small datasets, the order of the data records matters. This is an important difference to, e.g., the ADALINE algorithm introduced in Sect. 17.3: there, the weights and bias update is determined based on all data records at once.

Error Measures

How can the error be measured for the perceptron learning rule? Assuming that the number of data records is M , the *absolute error* (AE) for the classification can be easily computed as the total number of misclassified data,

$$\text{absolute classification error AE} = \sum_{i=1}^M \left| y^{\text{true}}_i - y^{\text{pred}}_i \right|, \quad (17.24)$$

where y^{true}_i is the true class label of data record number i and y^{pred}_i is corresponding predicted class label. This error measure can only take integer values that represent the number of misclassified data and can therefore be easily interpreted. Another useful error measure, that we already encountered in Sect. 11.7, is the mean absolute error (MAE), i.e., the number of mislabeled data divided by the number of samples

$$\text{mean abs. classification error MAE} = \frac{1}{M} \sum_{i=1}^M \left| y^{\text{true}}_i - y^{\text{pred}}_i \right|. \quad (17.25)$$

This allows to specify an error tolerance that is independent of the size of the dataset, which can be used to improve the termination criterion of the algorithm. For example, one could weaken the originally very strict requirement that all samples need to be correctly predicted and postulate that not more than 5% of the data should be misclassified.

A comment on “Learning”



As we saw in the previous chapters, *learning* typically requires a number of *examples* from which the *machine learning* (ML) model is supposed to generalize to new data. This has already been formulated by ROSENBLATT [8], who in his early publications was wondering about the amount of

(continued)

information needed to recognize a geometrical shape: the strength of a good and predictive model is *not* to learn *all* possible sizes and shapes of a triangle but rather to learn the most important characteristics from a finite number of examples.

Python Implementation of the Perceptron

A numerical implementation that uses the absolute classification error as a termination criterion can be found in Python Listing 17.2. The function `train_1_epoch`

```

1 import numpy as np
2 from perceptron_prediction import predict
3
4
5 def train_1_epoch(X_train, y_train, weights, bias, threshold, learning_rate):
6     n_samples, n_features = X_train.shape
7     y_pred = np.zeros_like(y_train)
8
9     for j in range(n_samples):
10         # feed forward direction
11         y_pred[j] = predict(X_train[j], weights, bias, threshold)
12
13         # backwards direction ("perceptron rule")
14         weights += learning_rate * (y_train[j] - y_pred[j]) * X_train[j]
15         bias += learning_rate * (y_train[j] - y_pred[j])
16
17     return weights, bias
18
19
20 def train(X_train, y_train, threshold, n_epochs, learning_rate):
21     n_features = X_train.shape[1] # the features are the columns of X_train
22     weights = np.zeros(n_features)
23     bias = 0.
24
25     for i in range(n_epochs):
26         weights, bias = \
27             train_1_epoch(X_train, y_train, weights, bias,
28                           threshold, learning_rate)
29
30         y_pred = predict(X_train, weights, bias, threshold)
31         if np.sum(np.abs(y_pred - y_train)) == 0:
32             break
33
34     return weights, bias
35
36
37 # The following two lines show how to use the code:
38 X_train, y_train = create_training_data() # X_train is a 2D array, y_train is 1D
39 weights, bias = train(X_train, y_train,
40                       threshold=0.4, n_epochs=50, learning_rate=0.5)

```

Python Listing 17.2 Implementation of the perceptron learning rule. The code reuses the function `predict` from Python Listing 17.1. Here, the training is stopped if the error is zero. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact

performs a complete training step consisting of the prediction and the back-propagation for all samples of the dataset and returns the updated weights and bias. The function `train` does the training for `n_epochs`. `x_train` is a 2D array and `y_train` is a 1D array; `x_train.shape` returns the number of rows (number of samples) and columns (number of features) of the 2D array. For each epoch, the updated weights and bias values are computed and, based on that, the `AE` calculated. For this, numpy's `abs` functions compute the absolute value *element-wise* for each entry of the vector `y_pred - y_train`. These values are then summed up. If the error is zero, then the training is stopped and the `break` statement quits the `for` loop. Note that changing the threshold can be counteracted by changing the bias. This is why by fixing the bias to zero, one can say, that the algorithm is also learning the threshold.

To check if the code works properly, a first “sanity check” is to predict the training data: for `x_train` we already know that the prediction should be `y_train`.

Convergence Theorem

Taking a look at Eq. 17.22, one observes that once a solution $\mathbf{w}$ has been found that correctly predicts all training data, then $\mathbf{w}$ will never be changed again. A formal proof is given by [9] and shortly afterward by [4] including a formal discussion of the limitations of the Rosenblatt perceptron.

17.2.5 Geometric Interpretation of the Perceptron Training

Interpretation of the Weights Vector

Point of departure for this part is the mathematical representation of lines. Already above we saw that the decision boundary for the case of a zero threshold is given by

$$\mathbf{w} \cdot \mathbf{x} + b = 0, \quad (17.26)$$

which is the equation of a line (or hyperplane if $\mathbf{x}$ is a position vector in an arbitrary dimensional space). A non-zero threshold value can be included in an adjusted value of b . From this form the normal of the line can be easily inferred: two vectors $\mathbf{a}$ and $\mathbf{c}$ are normal if and only if the scalar product $\mathbf{a} \cdot \mathbf{c} = 0$ (cf. Appendix A). To find a normal vector for a given vector $\mathbf{x}$ we therefore have to find a vector $\mathbf{a}$ such that $\mathbf{a} \cdot \mathbf{x} = 0$. Neglecting b in Eq. 17.26 we see that such a vector would be $\mathbf{w}$. This is still true if we add b , as this only acts as an offset of the line but does not change the direction. Therefore, $\mathbf{w}$ is a vector that is perpendicular to the line given by Eq. 17.26.

Choice of y Values

Sometimes the two classes are not chosen to be represented by the values 0 and 1 but rather by -1 and 1 . Both variants work equally well and result in identical training behavior if the learning rate is scaled accordingly. The former convention is rather motivated by Boolean decision problems where 0 and 1 have a clear correspondence

to the Boolean values of `False` and `True`. Additionally, for problems with more classes it is easy to simply count them starting from 0. The second convention is motivated by the fact that for the geometrical interpretation, the normal to the decision boundary, $\mathbf{b}$, points into the area, where correctly classified y values are *positive*. Those values in the area pointed at by the vector $-\mathbf{b}$ would be negative. For the rest of this section, we follow the second convention.

Geometry of the Perceptron Update

How is this important for understanding how the perceptron learning rule works? Figure 17.7a shows a hypothetical configuration with a given training dataset $\mathcal{D}$. According to Algorithm 17.1, a point number i (or equivalently the data record i) is chosen, and the corresponding position vector is $\mathbf{x}^{(i)}$. The current weights and bias result in the shown decision boundary and the normal vector $\mathbf{w}$. The update from the above algorithm reads for data record number i

$$\mathbf{w}^{\text{new}} := \mathbf{w}^{\text{old}} + \eta \left(y^{(i)} - \hat{y}^{(i)} \right) \mathbf{x}^{(i)}, \quad (17.27)$$

where we used the short form $y^{(i)} := y^{\text{true}}$ and $\hat{y}^{(i)} := y^{\text{pred}}$ to avoid too lengthy superscripts.

The point i is misclassified with $\hat{y}^{(i)} = +1$ and the term in parenthesis evaluates to $1 - (-1) = 2 > 0$. With $\lambda = 2\eta$ the new weight vector is then $\mathbf{w}^{\text{old}} + \lambda \mathbf{x}^{(i)}$ —addition of two vectors as shown in Fig. 17.7b. At the same time, the value of the bias is incremented by λ . This results in a rotation of the decision boundary in clockwise direction, cf. Fig. 17.7c. If the misclassification of a triangular marker would have been considered, then the factor λ would be negative, which results in a rotation of the decision line in the opposite, counter clockwise direction. If a point was correctly classified, then the decision boundary does not change. A perceptron model without a bias would have a hyper-plane that always goes through the origin.

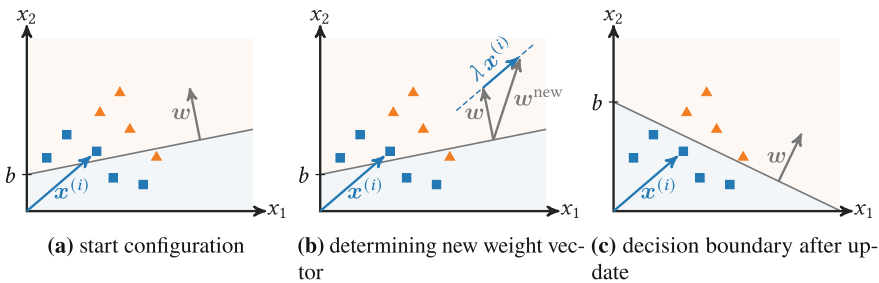


Fig. 17.7 Illustration of the geometrical relations implied by the perceptron learning rule. The decision boundary is given by $\mathbf{w}$ and b , cf. (a). There, three points are misclassified: two rectangular and a triangular point. An arbitrary point $\mathbf{x}^{(i)}$ is chosen that is then used for the update step. The new weight vector in (b) is the sum of the current $\mathbf{w}$ and a vector along the direction of $\mathbf{x}^{(i)}$: its length is proportional to the training rate as indicated by the factor λ . Additionally, the sign of the line direction is determined by the difference $y^{\text{true}} - y^{\text{pred}}$

17.3 The ADALINE Model

The *Adaptive Linear Neuron* (ADALINE) (also sometimes called “Adaptive Pattern Classification Machine”) was developed in the 1960s by BERNARD WIDROW and TED HOFF [12]. It is also based on the idea of the MCP neuron and the Rosenblatt perceptron. However, the novelty of ADALINE is an enhanced training strategy as compared to the perceptron. Below, we will see that the training strategy is in fact an efficient gradient descent method for finding optimal weights.

17.3.1 ADALINE Training Strategy

The key difference between the Rosenblatt perceptron and the ADALINE model is the way how the weights are adjusted: while the Rosenblatt perceptron used the output y to determine the training increments for bias and weights, the ADALINE model directly uses the agglomerated and weighted input values $\mathbf{w} \cdot \mathbf{x} + b$. This can be best seen in the schematic shown in Fig. 17.8.

There, after summation we have now an activation function, which here is just taken as the identity function $\varphi(x) = x$. The output is squeezed through a “quantizer”³—nothing but a threshold function, which, however, gives as output either -1 or $+1$ (see below for the motivation). The backward direction computes the error based on the output of the activation function according to the so-called *Widrow & Hoff learning rule*, which consists of the following increments

$$\Delta \mathbf{w} = (y^{\text{train}} - a(\mathbf{x}^{\text{train}}; \mathbf{w}, b)) \mathbf{x}^{\text{train}} \quad \text{and} \quad \Delta b = (y^{\text{train}} - a(\mathbf{x}^{\text{train}}; \mathbf{w}, b)). \quad (17.28)$$

$a(\mathbf{x}^{\text{train}}; \mathbf{w}, b)$ is the output of the (identity) activation functions for given weights and bias. Update of the weights and bias is then done using the learning rate η by the update step

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \Delta \mathbf{w} \quad \text{and} \quad b \leftarrow b + \eta \Delta b \quad (17.29)$$

where these equations correspond to the perceptron’s Eqs. 17.22 and 17.23. Additionally, the update is not done incrementally for each record or sample of the dataset. Instead, the two increments are computed for the whole dataset such that the update is done for all records simultaneously. Everything else stays the same as for the perceptron learning rule. A summary of the whole algorithm can be seen in Algorithm 17.2.

³ In signal processing, “quantization” denotes the process of mapping continuous data to discrete data.

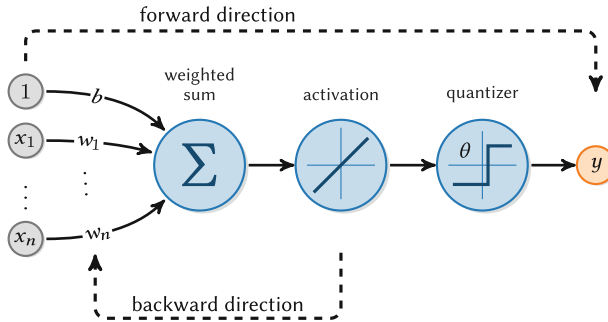


Fig. 17.8 Schematic of the ADALINE model: The forward direction implies evaluating the transfer, activation, and “quantizer” function for given weights, bias, and input values. For the backward direction, the error from the activation function (here: the identity function) is used to automatically adjust weights and bias

Algorithm 17.2: ADALINE Training with the Widrow & Hoff Learning Rule



1. **Initialization:** (training data must have $y^{\text{train}} \in \{-1; 1\}$)
 - a. initialize the weights $w_1, \dots, w_n$ and bias b with zero
 - b. choose a suitable learning rate $\eta \in (0, 1]$
 - c. define a maximum number of epochs (=iterations) as termination criterion; define an error tolerance as convergence criterion
2. **Forward step:** Make *simultaneous* predictions for *each* record of the dataset with current weights and bias:
 - a. compute the weighted sum $a(\mathbf{x}^{\text{train}}; \mathbf{w}, b)$
 - b. evaluate the activation function (here: identity function): this gives again the weighted sum a
 - c. evaluate the quantizer function: $y^{\text{pred}} = \varphi(a)$
3. **Backward direction:** use difference between true output, y^{train} , and the output values of the activation function, a , to compute update increments for weights and bias

$$\Delta y = y^{\text{train}} - a(\mathbf{x}^{\text{train}}; \mathbf{w}, b) \quad (17.30)$$

$$\Delta \mathbf{w} = \Delta y \mathbf{x}^{\text{train}} \quad \text{and} \quad \Delta b = \Delta y \quad (17.31)$$

Update the weights and bias to obtain the new values

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \Delta \mathbf{w} \quad \text{and} \quad b \leftarrow b + \eta \Delta b \quad (17.32)$$

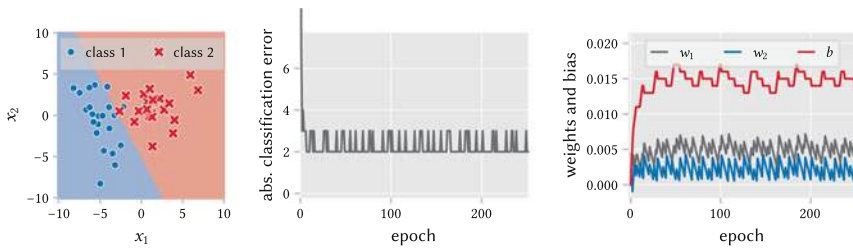
4. **GO TO step 2 and repeat** until the norm of the error, $\|\Delta y\|$, is small enough or until the maximum number of iterations is reached.

17.3.2 Example: Comparison Between the Perceptron and the ADALINE Model

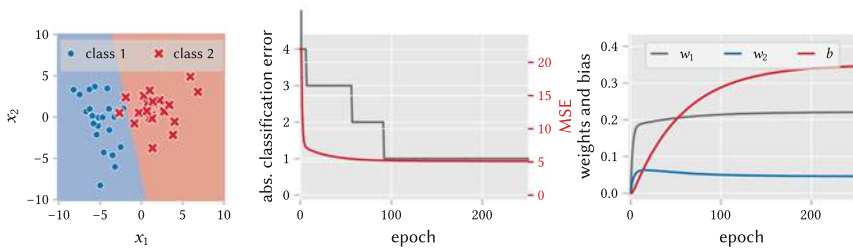
We now take a look at how well the perceptron and the [ADALINE](#) perform for a dataset that consists of two data classes, which have a small amount of overlap and are therefore, strictly speaking, not linearly separable. This was motivated by the fact that often in many real problems, some of the assumptions used for the models are slightly violated, but obviously a good model has a certain degree of tolerance.

In Fig. 17.9, we show the training results for the perceptron and the [ADALINE](#) model. All parameters, i.e., the training data itself, learning rate, initial values for bias and weights, are chosen identically. There, it becomes obvious that the [ADALINE](#) is numerically much more stable, which is the main side effect of using the output of the (continuous) activation function and not the output of the step functions.

For the perceptron, the error keeps fluctuating and does not reach the same small error as the [ADALINE](#). The decision boundary keeps jumping and rotating, which is also not fundamentally changing if the learning rate is further decreased. Also, the [ADALINE](#) model is not able to perfectly classify all the data, but this is not surprising



(a) Training of the Perceptron



(b) Training of the ADALINE model (additionally, the MSE error is shown in the middle panel)

Fig. 17.9 Comparison of the training results for the Perceptron and the [ADALINE](#) model. Top panel: final classification state, middle: error, bottom panel: evolution of weights and bias. The perceptron oscillates and only reaches a larger error than the [ADALINE](#) model. The [ADALINE](#) model is also numerically more stable. (a) Training of the Perceptron. (b) Training of the ADALINE model (additionally, the MSE error is shown in the middle panel)

as it is a *linear* model. Overall, it looks numerically much more well-behaved and converges to a set of weights and bias that result in the minimum possible error.

Is it surprising or problematic that the weights and bias values of the two models have different magnitudes? For almost all problems, there is no unique solution to the problem of finding a decision boundary. Thus, there exists a potentially large range of parameters that work equally well. Additionally, a mathematical line is fully determined by two parameters, whereas the two studied models contain two weight parameters and the bias and therefore have one degree of freedom more than needed.

17.3.3 The Relation to Gradient Descent Optimization

The **ADALINE** training uses the linear activation function a for computing an error. This has the advantage that a is differential, and that we can define a cost function $J(\mathbf{w})$ that can then be minimized. Here and in the remainder of this paragraph, we assume that the vector $\tilde{\mathbf{w}}$ contains both the weights and the bias, $\tilde{\mathbf{w}} := [w_1, \dots, w_n, b]^T$ as introduced in Eq. 17.8, which also required a new definition of the input vector as $\tilde{\mathbf{x}} := [1, x_1, \dots, x_n]^T$. Then we define the cost function as the sum of the squared errors of all M data records,

$$J(\tilde{\mathbf{w}}) = \frac{1}{2} \sum_{i=1}^M (y_i - a(\tilde{\mathbf{x}}_i; \tilde{\mathbf{w}}))^2. \quad (17.33)$$

$\mathbf{x}_i$ is the i -th data record, and the factor of one half is here to simplify the formulation after taking the derivative below and does not have any further influence. Minimizing the cost function w.r.t the weight components then yields the values of the trained weights. In the following, we use the simple gradient descent method (cf. Chap. 12 for further details). There, we should follow the direction of the steepest descent, which is $-\nabla J(\mathbf{w})$. Rewriting this component-wise lets us explicitly perform the differentiation along the j -th component:

$$\frac{\partial J}{\partial \tilde{w}_j} = \frac{1}{2} \frac{\partial}{\partial \tilde{w}_j} \left(\sum_{i=1}^M (y_i - a(\tilde{\mathbf{x}}_i; \tilde{\mathbf{w}}))^2 \right) \quad (17.34)$$

For taking the partial derivative of the lengthy expression, we note that the derivative of a sum equals the sum of the derivatives. Therefore, we can move the partial differentiation into the summation and apply it to the $(\dots)^2$ term using the chain rule. This results in

$$\frac{\partial J}{\partial \tilde{w}_j} = \sum_{i=1}^M (y_i - a(\tilde{\mathbf{x}}_i; \tilde{\mathbf{w}})) \left(-\frac{\partial a(\tilde{\mathbf{x}}_i; \tilde{\mathbf{w}})}{\partial \tilde{w}_j} \right). \quad (17.35)$$

For taking the partial derivative of a with respect to any $\tilde{w}_j$, recall that with the extended notation of $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{w}}$ it is $a_i = \sum_j \tilde{w}_j \tilde{x}_{ij}$ (hint: recall the definition of the data matrix, Sect. 3.3.3). Therefore, there is only one element of the sum that depends on $\tilde{w}_j$, which would be the j -th element. Then, it is $\partial a_i / \partial \tilde{w}_j = \tilde{x}_{ij}$ with which follows that

$$\frac{\partial J}{\partial \tilde{w}_j} = - \sum_{i=1}^M (y_i - a(\mathbf{x}_i; \tilde{\mathbf{w}})) \tilde{x}_{ij} = - \sum_{i=1}^M (y_i - \tilde{\mathbf{x}}_i \cdot \tilde{\mathbf{w}}) \tilde{x}_{ij} \quad (17.36)$$

This equation gives the sensitivity of the cost function with respect to the j -th weight. Writing this in compact vectorial form for *all* weights eliminates the occurrence of the index j and reads:

$$\frac{\partial J}{\partial \tilde{\mathbf{w}}} = - \sum_{i=1}^M (y_i - \tilde{\mathbf{x}}_i \cdot \tilde{\mathbf{w}}) \tilde{\mathbf{x}}_i. \quad (17.37)$$

This is, by construction, a vector that points into the direction of the steepest descent. For the iterative update of the search direction, the gradient descent method then takes a small step along this new search direction, resulting in the weights increment

$$\Delta \tilde{\mathbf{w}} = -\eta \nabla J(\tilde{\mathbf{w}}) = \eta \sum_{i=1}^M (y_i - \tilde{\mathbf{x}}_i \cdot \tilde{\mathbf{w}}) \tilde{\mathbf{x}}_i \quad (17.38)$$

$$\Leftrightarrow \Delta \tilde{w}_j = -\eta \frac{\partial J}{\partial \tilde{w}_j} = \eta \sum_{i=1}^M (y_i - \tilde{\mathbf{x}}_i \cdot \tilde{\mathbf{w}}) \tilde{x}_{ij}. \quad (17.39)$$

Note that for comparison with the delta rule of the [ADALINE](#) model, we have to convert the extended notation $\tilde{\mathbf{w}} = [w_1, \dots, w_n, b]^\top$ back to $\mathbf{w}$ and θ . This is the same update formula that is used from the [ADALINE](#) model above which also shows why [ADALINE](#) is indeed based on the gradient descent optimization method.

We conclude this derivation by mentioning that $\partial J / \partial \mathbf{w}$ is not necessarily the direction of the minimum. However, for problems that exhibit only a global minimum and no local minima, the gradient descent method can be shown to always converge to the global minimum. For cost functions that have local minima, this guarantee is only restricted to the vicinity of the global minimum.

17.3.4 Limitations of the Perceptron and the ADALINE Model

We have seen that both models—even the simplistic perceptron—work very well for linearly separable data. This gave rise to great hope that with such models truly artificial intelligent machines could be created. This hope was vastly destroyed in 1969 by Minsky and Papert [4], who in their book formally proved that no linear,

single layer model is able to learn the simple logical XOR operator. This limitation (among others) was so severe, that after the rapid progress and enthusiasm of the 1950s and 1960s, further developments were dramatically reduced until the 1980s, a period of which is sometimes called *AI winter*.

17.4 Increasing the Complexity: Assemblies of Neurons

From an early point in time on, it was clear that the perceptron and the [ADALINE](#) model were very limited in terms of learnable the degrees of complexity. In fact, already the very early units were combined to form assemblies of perceptrons to increase the complexity. Again, the motivation came from the biological nervous system. In the following, we first analyze networks of neurons and discuss what kind of functions can be represented by them. In a next step, we then turn to the investigation of aspects related to the training of such assemblies.

17.4.1 A First Network of Two Computational Neurons

The obvious approach is to combine multiple perceptrons in order to increase the complexity of the model and—hopefully—thereby also to increase the complexity of the predictable phenomena. In Fig. 17.10a, two neurons with n input values/features are combined.

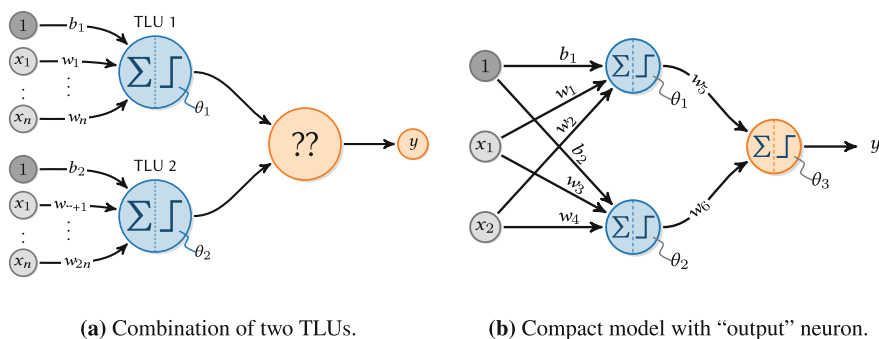


Fig. 17.10 Combination of two perceptrons: (a) The output of two perceptrons with n input nodes will be combined in the “??”-node. This gives then the output y , a real value or a categorical value (class). (b) is a more compact notation with input nodes that are “reused” for each neuron; their output is summed and tested against a threshold value (the large orange node), yielding the output y that is used for classification. For a multi-class classification problem, more output nodes would be used; for regression problems, there would be no thresholding component in the output node

Example with 2 Perceptrons

The output of each of the two perceptrons is fed as input into an output unit that—depending on whether a regression problem, a binary or multi-class classification problem is to be solved—has different functionality.

We turn now to a concrete example: a network of two perceptrons, each of which has two inputs, to be used for a classification problem. This is shown in Fig. 17.10b where we also changed to a commonly used depiction: the two identical input nodes of the perceptrons are merged, and from every input node connections go to both units; their output is not depicted as an individual node.⁴ We assume that each of the three shown units consists of a summation and activation function of the form $a(\mathbf{w} \cdot \mathbf{x}; \theta)$ where θ is again a threshold (the notation with the semicolon indicates that θ is a constant parameter while everything before the semicolon is considered a variable. Sometimes, the parameters are omitted for brevity). We can then start from the right of Fig. 17.10b to recursively derive the formulation for y :

$$y = a_3(y_1 w_5 + y_2 w_6; \theta_3) \quad (17.40)$$

where y_1 and y_2 are the output values from the two TLUs. They can be computed as

$$y_1 = a_1(b_1 + x_1 w_1 + x_2 w_2; \theta_1) \quad (17.41)$$

$$y_2 = a_2(b_2 + x_1 w_3 + x_2 w_4; \theta_2) \quad (17.42)$$

The a_i are the transfer and activation functions with the threshold values θ_i . The above equations are already sufficient to perform predictions if the weights and the bias values are given. In Example 17.3, a step-by-step calculation is performed for the case of two perceptrons.

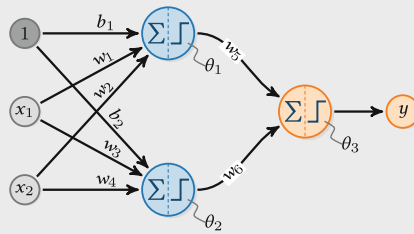
Example 17.3 (Prediction with two Perceptrons)



The following figure depicts a simple network of two combined perceptrons and an output unit:

(continued)

⁴ This is a convention that exists mainly for historical reasons. Additionally, in the context of connected neurons, only one node is needed for storing the value, e.g., in case that the output of the one is the input of the other.



It requires altogether six weights and two bias values, which are given as:

$$\begin{aligned} w_1 &= -2.0, & w_2 &= 2.0, & b_1 &= 0.5 \\ w_3 &= 1.5, & w_4 &= -2.1, & b_2 &= 0 \\ w_5 &= -1.8, & w_6 &= -2.2 \end{aligned}$$

together with the threshold values $\theta_1 = -1$ and $\theta_2 = -1$ for the two perceptrons and $\theta_3 = -2.5$ for the output unit (which we define to have values of either 0 or 1). What is the output y for a given input of $x_1 = 0.8$ and $x_2 = 0.3$?

We start by evaluating Eq. 17.41:

$$y_1 = a_1(0.5 + 0.8 \cdot (-2.0) + 0.3 \cdot 2.0) = a_1(-0.5)$$

To compute the first activation function $a_1(-0.5)$, we compare the function's argument with the threshold $-0.5 \geq \theta_1 = -1$ from which follows that $y_1 = a_1(-0.5) = 1$. In analogy, we find for Eq. 17.42 that $y_2 = a_2(0.57) \geq \theta_2 = -1$. From this follows that $y_2 = 1$. Finally, we compute the overall output from Eq. 17.41 via $a_3(-1.8 \cdot 1 + (-1.8) \cdot 1)$ and the comparison with θ_3 resulting in $y = 0$.

Is this a network with 2 or 3 neurons?



The network in Example 17.3 seems to show altogether three neurons but we are only talking about two—was there a mistake?

Well, even though in the figure, y is computed as the output of a perceptron with the same mathematical form as the two first units, it is composed of a distinct function. This will become more clear, e.g., in case of a multi-class classification or regression problem where this unit will take a different mathematical form. This is why this network is considered a “2-neuron network.” We will also discuss this when we analyze the decision boundary of such a network.

17.4.2 The Boolean XOR Decision Problem

Is this simple network able to perform better on the Boolean XOR problem than a single neuron? Using the same network structure as in Example 17.3, we perform computations with all possible permutations of 0s and 1s as input, as shown in Table 17.4 (note that there we used the parameters given in the caption of the table). Then, it is easy to show that the Boolean XOR operator can indeed be represented by this network.

Note that the output of the two neurons is -1 or $+1$, whereas the final output y will be either 0 or 1 as we wanted to represent the Boolean states `False` and `True` (another hint that the output unit is different from the two neurons).

Why are two neurons able to solve the XOR problem and in which aspects is their combination different from a single neuron? To understand this, we visualize the function a_3 depending on the input x_1 and x_2 as shown in Fig. 17.11 (recall that in Eq. 17.40 the variables y_1 and y_2 depend on x_1 and x_2 and therefore y implicitly depends on them as well).

The “decision-making” of the two individual TLUs shows the expected linear decision boundary separating regions of $y_1(x_1, x_2) = -1$ from those with $y_1(x_1, x_2) = +1$ (y_2 in analogy). These two regions are also called *half-spaces* as they start at the decision boundary and from there continue infinitely. The output node essentially adds weighted versions of y_1 and y_2 and then compares it with a threshold value. In our case, the values of the weights w_5 and w_6 are not too different so that effectively y_1 and y_2 are averaged, as can be clearly seen in the bottom left plot of Fig. 17.11. The values between which the output node should choose are expected to represent Boolean values. Therefore, we set it such that the values can only be 0 or 1.

In conclusion, we observe that the composition of two perceptrons effectively combines two decision boundaries, which allows to create a more complex resulting decision boundary that is not restricted to linearly separable problems and—finally—is able to solve the Boolean XOR decision problem. Do we also have to show that the other Boolean decision problems (AND NOT and OR) can still be

Table 17.4 Representation of the logical XOR by two connected perceptrons. Parameters used are $w_1 = 0.7$, $w_2 = -1.1$, $b_1 = 1$, $w_3 = -w_4 = 2$, $b_2 = 0$, $\theta_1 = 0.4$, $\theta_2 = 0.8$, $w_5 = -w_6 = 0.5$, $b_3 = 0$, $\theta_3 = 0.1$. The two output columns show the model output as well as the true output $x_1 \dot{\vee} x_2$

Input				Data Processing							Output
x_1	x_2	a_1	$a_1 \geq \theta_1 ?$	y_1	a_2	$a_2 \geq \theta_2 ?$	y_2	a_3	$a_3 \geq \theta_3 ?$	$y = \varphi(a_3)$	$x_1 \dot{\vee} x_2$
0	0	1.0	True	1	0.0	False	-1	1.0	True	0	0
1	0	1.7	True	1	2.0	True	1	0.0	False	1	1
0	1	-0.1	False	-1	-2.0	False	-1	0.0	False	1	1
1	1	0.6	True	1	0.0	False	-1	1.0	True	0	0

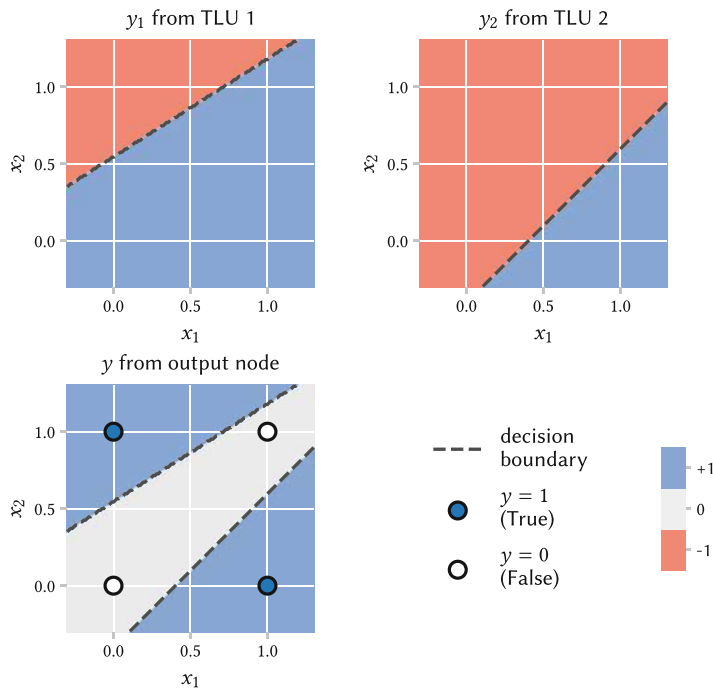


Fig. 17.11 Results from an assembly of two perceptrons with parameters according to Example 17.3. The weights, bias, and threshold values are chosen as in Table 17.4. The top left and top right panels are the intermediate outputs of neuron 1 and 2, the bottom panel shows the results of the output node. y_1 and y_2 are the output of perceptrons and are either -1 or $+1$. As the output y should represent Boolean values, it was here defined to be either 0 (`False`) or 1 (`True`). In the bottom left panel, the composition of the effect of the two perceptrons becomes directly visible in the resultant decision boundaries

represented? The proof that this still holds is simple as we can set one of the weights of the output unit to zero such that effectively only one unit is contained.

17.4.3 Python Implementation of the Two-Neuron-Assembly

In Python Listing 17.3, an implementation of the assembly of two neurons is shown, which is intuitively understandable and matches the mathematical formulation. There, we don't even reuse functions from the previous sections.

Why not directly implement this for an arbitrary number of inputs or neurons? Because such code would not be very flexible, e.g., if the number of neurons or inputs changes, a substantial part of it needs to be rewritten. The purpose of this code example is twofold: First of all, it shows the information flow very clearly and thereby paves the way for more complex and general networks introduced in the following sections. The second important aspect is that we can use these lines of

```

1 import numpy as np
2
3 b1, w1, w2, theta1 = 0.5, -2.0, 2.0, -1.0 # TLU 1
4 b2, w3, w4, theta2 = 0.0, 1.5, -2.1, -1.0 # TLU 2
5 b3, w5, w6, theta3 = 0.0, -1.8, -2.2, -2.5 # output node
6
7 # the 4 points for the XOR problem: (0, 0), (1, 0), (0, 1), (1, 1)
8 x1 = np.array([0, 1, 0, 1])
9 x2 = np.array([0, 0, 1, 1])
10
11 # TLU 1
12 a1 = w1 * x1 + w2 * x2 + b1 # weighted sum
13 y1 = np.where(a1 >= theta1, +1, -1) # evaluate activation functions
14
15 # TLU 2
16 a2 = w3 * x1 + w4 * x2 + b2 # weighted sum
17 y2 = np.where(a2 >= theta2, +1, -1) # evaluate activation functions
18
19 # Output node
20 a3 = w5 * y1 + w6 * y2 + b3
21 y = np.where(a3 >= theta3, False, True)
22
23
24 print(f'x1 = {x1}, x2 = {x2}')
25 print(f'y = {y}')
26 # [output]: x1 = [0 1 0 1], x2 = [0 0 1 1]
27 # [output]: y = [ True False False  True]

```

Python Listing 17.3 An implementation of the network of two neurons. Even though it is a very simplistic implementation, it is able to handle a number of input records at the same time as can be seen by the fact that `x1` and `x2` are both numpy `array`s

code to produce “reference solutions” to be used as validation of more complicated networks.

Parameters used are those for solving the XOR problem from above. With the given comments, the code should be mainly self explanatory. We note, that `a1` in line 9 is again an `array` where each entry corresponds to one of the four data record of the input `x1` and `x2`. `y1` uses numpy’s `where` function that gives `+1` for each element that fulfills the condition (e.g. `a1 >= theta1`) and `-1` otherwise. The function `where` is able to do this for all elements of an array.

Words of advice



In case that the reader skipped the sections on vectorization in the chapters on regression (Sect. 12.4) we suggest to try to understand the “vector operations” in Python Listing 17.3 (i.e., the way how the four data records in `x1` and `x2` are handled). This will be needed for many of the subsequent implementations. Printing the output of each line and adding more entries to `x1` and `x2` (=increasing the number of data records) might also be helpful.

17.4.4 A Fully Connected Network with Input, Output and a Single Hidden Layer

Already a few years after ROSENBLATT published the perceptron [8] he also published the concept of a *multilayer perceptron* (MLP), which contained an input layer, a hidden layer, and an output layer [9] (“layers” denote arrays of similar neurons or nodes, stacked on top of each other, as shown in Fig. 17.12). The hidden layer was initialized with random values for the weights. However, there was no mechanism such that the hidden layer was able to adjust the weights. MLPs has been used in many variations and could also be studied mathematically. They are a predecessor of DL, and therefore, we will introduce them now in somewhat more detail.

Starting from single units, we can assemble more complex structures in the same spirit as when we assembled two perceptrons above. As the assemblies are now getting more complex, also the notations become more and more involved. As this can quickly become overwhelming (who likes dealing with variables with four or five indices?!), we first take an intermediate step considering a network with only *three* different types of “layers.” Figure 17.12 shows an example of such a network.

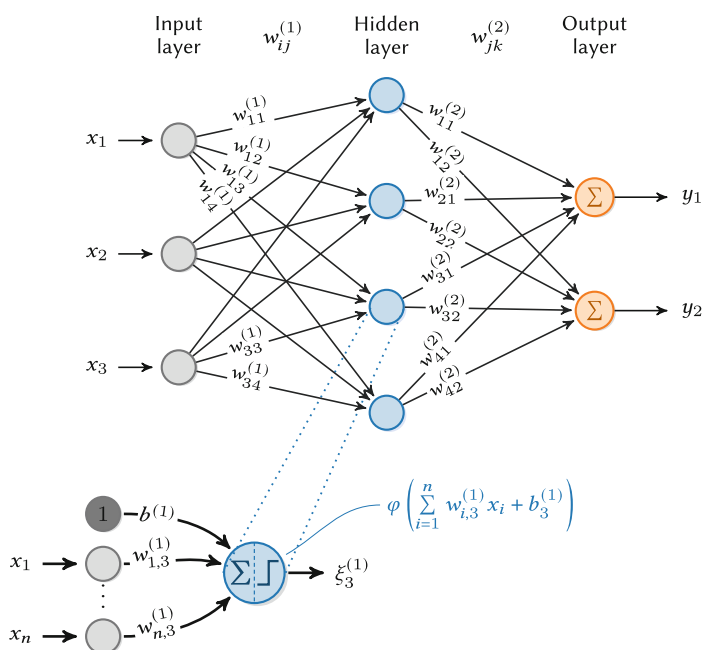


Fig. 17.12 Example of a neural network with one input layer (containing three input nodes), a hidden layer (with four perceptrons), and an output layer (with two output nodes). On the right is the symbolic representation that is used in the following and that is in detail explained in the text. Note that the “bias input node,” usually indicated by the “1,” is not shown but implicitly assumed to be always present. Two distinct sets of weights are present, the $w_{ij}^{(1)}$ and the $w_{jk}^{(2)}$

There are two layers that are visible or accessible from the outside of the model: the input layer containing nodes that store the input data x_i , and the output layer collecting data and returning them in form of the values y_j . In between is a hidden layer that is not visible from outside the model. All layers can have an arbitrary number of nodes or neurons. Note that there is a fundamental difference between the nodes of the input and the output layer: the output nodes perform computations, while the nodes of the input layer only store data.

Words of advice



Naming conventions are often not standardized and rather based on “common agreements” or habits. For example, a number of different names exist for the network shown in Fig. 17.12: sometimes it is called a *two-layer* network as there are *two* layers that do actual computations the hidden and the output layer (or because there are two “layers of weights”), but it really *consists* of three layers of nodes (the input, hidden, and output layer). Sometimes the network is also called *single layer with hidden units* or *network with one hidden layer*.

Fully Connected Layers

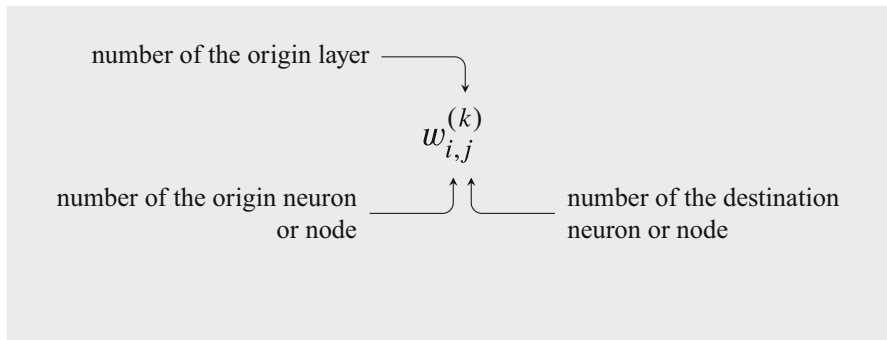
All nodes of the two neighboring layers are interconnected as shown in Fig. 17.12: each node on the origin layer (e.g. the input layer) is connected with all nodes of the destination layer (e.g. the hidden layer). Layers with this kind of connection are called *fully connected layers*. The two layers can have arbitrary numbers of nodes. For example, if the first layer has n nodes and the second layer has m nodes, then there are altogether $n \times m$ connections between the two layers.

Notations for Weights

Each of these connections is associated with a weight, where the weights between the first layer (the input layer) and the hidden layer are indicated by a superscript ⁽¹⁾, e.g., $w_{ij}^{(1)}$, and those between hidden and output layer are denoted as $w_{ij}^{(2)}$. To uniquely indicate the start and the end node of the sets of weights, we will use the following index notation:

Definition 17.1 Notation for Weights Each weight of the fully connected layers is uniquely characterized by one superscript and two subscripts:

(continued)



For example, the connection from input layer (layer no. 1) to hidden layer (layer no. 2) has $k = 1$ and the connection from the hidden to the output layer is $k = 2$. The subscripts in $w_{1,2}^{(2)}$ indicate that the first node/feature of the starting layer is connected to the second node or neuron of the receiving layer (starting and receiving are meant in terms of the information flow); a starting node with the number 0 indicates that this “weight” is in fact the bias, e.g., $w_{0,1}^{(1)}$ denotes the bias of the first neuron in the hidden layer.

Words of advice



The above introduced “index notation” should not be confused with a short notation for partial derivatives where, e.g., $f_{i,x} \equiv \partial f_i / \partial x$ indicates the partial derivative w.r.t. x of the i -th component of (the vector-valued) f .

Mathematical Formulation of the Network

We will now derive the governing equation for the first output node for the network shown in Fig. 17.12. For this we start from the output node and make our way through the hidden layer, back to the input layer. For simplicity, we only assume a single data record; the more general case of multiple data records that can be handled simultaneously will be formulated below in Sect. 17.4.5. We start our derivations from the first output y_1 . The corresponding output nodes simply sum up all incoming values together with a bias value $b_1^{(2)}$ (again, the bias is not shown in Fig. 17.12):

$$y_1 = b_1^{(2)} + w_{11}^{(2)} \xi_1 + w_{21}^{(2)} \xi_2 + w_{31}^{(2)} \xi_3 + w_{41}^{(2)} \xi_4 . \quad (17.43)$$

There, ξ_i is the output of the i -th neuron of the hidden layer and therefore, strictly speaking, also should have the superscript $^{(1)}$. If the bias is again treated as a weight, $w_{01}^{(2)}$, then Eq. 17.43 can be written as a scalar product of two vectors,

$$y_1 = \boldsymbol{\xi} \cdot \mathbf{w}_1^{(2)} \quad \text{with} \quad \boldsymbol{\xi} = [1, \xi_1, \dots, \xi_4]$$

$$\text{and } \mathbf{w}_1^{(2)} = [w_{01}^{(2)}, \dots, w_{41}^{(2)}]^\top. \quad (17.44)$$

The scalar product $\xi \cdot \mathbf{w}_1^{(2)}$ is shorter and looks more elegant than Eq. 17.43 but is also less explicit—in particular keep the “special” definition of ξ from Eq. 17.44 in mind and note that $\mathbf{w}_1^{(2)}$ is the vector that contains the weights for the output node number 2. The second output is computed in analogy, exchanging $\mathbf{w}_1^{(2)}$ with $\mathbf{w}_2^{(2)}$.

ξ_i is the output from neuron number i and can be computed by just following the same rule as we did already for the single Rosenblatt perceptron:

$$\begin{aligned} \xi_1 &= \varphi_1 \left(\mathbf{x} \cdot \mathbf{w}_1^{(1)} \right) \quad \text{with } \mathbf{x} = [1, x_1, x_2, x_3] \\ \text{and } \mathbf{w}_1^{(1)} &= [w_{01}^{(1)}, \dots, w_{41}^{(1)}]^\top. \end{aligned} \quad (17.45)$$

As before, we treat the bias as 0-th weight and therefore again extend the vectors $\mathbf{x}$ and $\mathbf{w}_1^{(1)}$ with 1 and $w_{01}^{(1)}$, respectively. The outputs from the other neurons are then computed in analogy, e.g.,

$$\xi_2 = \varphi_2 \left(\mathbf{x} \cdot \mathbf{w}_2^{(1)} \right). \quad (17.46)$$

17.4.5 “Vectorized” Operations: The Equations for the Network with a Hidden Layer

This can now be summarized for the network with three features and inputs, four units contained in the hidden layer, and two output nodes, starting from the final output of the network. Additionally, for the computational efficiency, it is beneficial to “vectorize” these steps. Vectorization implies that iterating over elements of a vector or matrix by using `for` loops can be avoided. Instead, by performing matrix–matrix or matrix–vector products, all calculations can be done virtually simultaneously because Python’s numpy library or MATLAB® is able to strongly optimize these operations.⁵ As compared to using regular `for` loops over individual elements of arrays, this may result in a speed up of the computational time by factors of 10 and even up to 1000 or more! Last but not least, for the current mathematical problem, such a notation additionally looks compact and thereby helps to understand the structure of the equations.

⁵ This is typically done by using pre-compiled and possibly strongly optimized code, which itself is typically written in a low-level language such as C or Fortran. *Vectorization* does not imply any parallel execution of code even though an already vectorized code often can easily be run on multi-cores/multi-processors as well.

$$\mathbf{y} = [y_1 \ y_2] = [1 \ \xi_1 \ \xi_2 \ \xi_3 \ \xi_4] \cdot \underbrace{\begin{bmatrix} w_{01}^{(2)} & w_{02}^{(2)} \\ w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} \\ w_{41}^{(2)} & w_{42}^{(2)} \end{bmatrix}}_{\mathbf{W}^{(2)}}. \quad (17.47)$$

Next, the intermediate values ξ_i can be computed by applying a different activation function to each of them:

$$\boldsymbol{\xi} = [\xi_1 \ \xi_2 \ \xi_3 \ \xi_4] = \left[\varphi_1 \left(\mathbf{x} \cdot \mathbf{w}_1^{(1)} \right) \ \dots \ \varphi_4 \left(\mathbf{x} \cdot \mathbf{w}_4^{(1)} \right) \right]. \quad (17.48)$$

Defining $\boldsymbol{\varphi}$ as the vector-valued function that consists of the φ_i , this can be written in short as

$$\boldsymbol{\xi} = \boldsymbol{\varphi} \left(\mathbf{x} \cdot \begin{bmatrix} w_{01}^{(1)} & w_{02}^{(1)} & w_{03}^{(1)} & w_{04}^{(1)} \\ w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} & w_{14}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} & w_{24}^{(1)} \end{bmatrix} \right) = \boldsymbol{\varphi} \left(\mathbf{x} \cdot \mathbf{W}^{(1)} \right), \quad (17.49)$$

where we additionally introduced the matrix of all weights between the input layer and the hidden layer as $\mathbf{W}^{(1)}$. These are now formulations that are straightforward to be generalized to a network with one hidden layer, an arbitrary number of features, arbitrary number of units contained in the hidden layer, and output nodes. Furthermore, without much of additional work, it is also possible to consider not just a single data record but *all* data records of the data matrix at once and in a vectorized manner.

We can now easily turn the vectorized equations into a Python implementation as shown in Python Listing 17.4.

17.4.6 Capabilities of the Networks for Classification and Regression

One of the most important components that determines whether the network performs a regression or a classification task is the output node. So far we encountered two different types of output nodes: the first one contained a threshold function—indicator for the fact that this is about classification. With a `True` or `False` (i.e., 1 or 0) output, we can perform a *binary classification* for deciding if something belongs to class A or to class B. The first more complex network, Fig. 17.12, contained two output nodes that were only performing (weighted) summations resulting in real numbers y_1 and y_2 and therefore is a regression model. This particular example takes three features as input, e.g., the spatial x , y , and z coordinates of a point and results in two real valued outputs, e.g., the

```

30 import numpy as np
31
32 """ == These are the used parameters and variables: =====
33 X: input data,                shape: (n_samples,      n_features)
34 W1: weights of hidden layer,  shape: (n_features + 1, n_units   )
35 W2: weights of output node,   shape: (n_units + 1,   1       )
36 theta1: threshold hidden layer, shape: (n_units)  -> a 1d array
37 theta1: threshold output node, shape: (1)         -> a 1d array
38 """
39
40 n_samples = X.shape[0]                # number of samples/data records
41
42 # prepend a "1"-column for bias of hidden layer neurons
43 X1 = np.concatenate((np.ones((n_samples, 1))), X), axis=1)
44
45 # copy the row vector theta 1 n_samples-times
46 THETA1 = np.repeat(theta1, n_samples, axis=0)
47 #
48 # evaluate the neurons of the hidden layer
49 T1 = np.where(X1 @ W1 >= THETA1, 1, -1)
50
51 # prepend a "1"-column for bias of output nodes
52 T1 = np.concatenate((np.ones((n_samples, 1))), T1), axis=1)
53
54 # compute results of output nodes
55 y = np.where(T1 @ W2 >= theta2, True, False)

```

Python Listing 17.4 General implementation of the vectorized equations for a network with one hidden layer. Note that `np.where` also works with the row vector `theta1`: it automatically applies it to each row of `X1 @ W1` so that creating `THETA1` is not required

temperature and a chemical concentration. Generalizing these approaches to, e.g., multiple classes and increasing their performance is a subject that will be covered at a later point after the training of these networks was introduced.

17.5 Summary and Historical Remarks

We saw how increasingly complex networks developed, starting from the example of the biological neuron as a “template.” All of the above developed networks of increasing complexity are able to represent increasingly complicated functions and, altogether, looked very promising in the 1960s. Problems intensively studied back then included the structure of “error surfaces” for Boolean operators of various complexities and how to represent assemblies of Boolean operators with possibly many inputs (see e.g. [6, 10] and references therein).

The greatest shortcoming, however, turned out to be a suitable “training algorithm” for more complex networks: transferring the strategy of the [ADALINE](#) training with the “delta rule” (or the Widrow & Hoff learning rule) from a single unit to more complex networks was not straightforward as the delta rule is based on the difference of the actual and desired output. The latter is available only for the unit that yields the output. Therefore, it is not clear how the weights and bias values of the other neurons in any of the hidden layers should be adjusted during the training.

This is why after a sustained period of excitement, [ANNs](#) was seen as a research “dead end” and, what is commonly known as the “AI winter” began and lasted until the end of the 1970s.

17.6 Exercises

17.1 Given are two Boolean variables x and y . Show how the Boolean operations AND, OR, and NOT can be expressed using “regular” number arithmetic, i.e., by interpreting `True` as 1 and `False` as 0.

17.2 Show that the perceptron is not able to represent the Boolean XOR operation.

17.3 Visualize a perceptron with two inputs that would be able to represent the logical AND operation. Also show the decision boundary in analogy to Fig. [17.4](#).

17.4 Assuming that we have Boolean functions with two input variables. Show that then there exist 16 different Boolean functions. Show geometrically that only two of them are not linearly separable

17.5 Assuming that we have Boolean functions with three input variables. How many different Boolean functions exist then and how many of them are linearly separable?

17.6 Visualize a perceptron with two inputs that would be able to represent the logical AND operation. Also show the decision boundary in analogy to Fig. [17.4](#).

17.7 Use the following function to create a dataset that consists of two normal distributions so that the data set has 2 features (the x and y coordinate) and the label 0 or 1.

```

1  def create_bivariate_normal_distribution(n_points=500):
2      X = np.empty((0, 2))
3      y = np.empty(0)
4
5      mean = [6, -2]
6      cov = [[1, -0.6], [-0.6, 3]]
7      x1, x2 = np.random.multivariate_normal(
8                  mean, cov, n_points
9              ).T
10     X = np.append(X, np.stack((x1, x2), axis=1),
11                  axis=0)
12     y = np.append(y, np.zeros_like(x1))
13
14     mean = [-1, 1]
15     cov = [[2, 0.6], [0.6, 12]]
16     x1, x2 = np.random.multivariate_normal(
17                 mean, cov, n_points
18             ).T

```

```

19 X = np.append(X, np.stack((x1, x2), axis=1),
20                  axis=0)
21 y = np.append(y, np.ones_like(x1))
22
23 return X, y

```

Use the Python Listing 17.2 to perform a classification. Create a plotting function that shows the decision boundary. Additionally, adjust the code such that the error is plotted as a function of the number of epoch.

17.8 The ADALINE model can also be used for regression problems. Which are the parts of the ADALINE model that need to be adapted? Apply your solution for predicting the inclination of the linear elastic regime of the MDS-1 (Tensile Test) dataset.

17.9 Assume that all weights and biases in a network of perceptrons are scaled by a constant value, $\lambda > 0$. How does this influence the network?

17.10 In Sect. 17.4.2 it was shown that for a particular set of weights, biases and thresholds a network of two TLUs is able to represent the Boolean operator XOR. The solution resulted in three regions for the output value y : two regions where $y = 1$ and one regions where $y = 0$.

- (i) Show by an example that there exists another class of solutions that consist of two regions where $y = 0$ and one regions where $y = 1$.
- (ii) Explain a systematic approach for finding these weights starting from the solution in Sect. 17.4.2.

References

1. G. Berlucchi and H. A. Buchtel. Neuronal plasticity: historical roots and evolution of meaning. *Exp Brain Res*, 192:307–319, 2009. DOI <https://doi.org/10.1007/s00221-008-1611-6>.
2. B. Blaus. Multipolar neuron. URL https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png.
3. W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.
4. M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.
5. G. Nagy. Neural networks-then and now. *IEEE Transactions on Neural Networks*, 2(2):316–318, 1991. DOI <https://doi.org/10.1109/72.80343>.
6. R. Rojas. *Neural Networks—A Systematic Introduction*. Springer, Berlin, Heidelberg, 1st edition, 1996. ISBN 978-3-540-60505-8. DOI <https://doi.org/10.1007/978-3-642-61068-4>.
7. F. Rosenblatt. *The Perceptron—A Perceiving and Recognizing Automaton Project Para. Report: 85-460-1*, Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, Buffalo, New York, Ithaca, New York, January 1957.
8. F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. DOI <https://doi.org/10.1037/h0042519>.

9. F. Rosenblatt. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms, 1962.
10. Q. Sheng. *Threshold Logic*. Academic Press Inc, 1969. ISBN 978-0126398502.
11. W. Waldeyer. Ueber einige neuere Forschungen im Gebiete der Anatomie des Centralnervensystems). *DMW—Deutsche Medizinische Wochenschrift*, 17(44):1213–1218, Nov. 1891. DOI <https://doi.org/10.1055/s-0029-1206824>.
12. B. Widrow. An Adaptive “Adaline” Neuron Using Chemical “MEMISTORS”, Technical Report No. 15553-2, Stanford University, USA, 1960.

More data beats clever algorithms, but better data beats more data.

Peter Norvig (born 1956)

American Computer Scientist

18.1 Overview of the Historical Developments

Even though research activities were strongly reduced during “first AI winter”, which lasted from the early 1970s until 1980, there still was some progress. One of the important developments was the introduction of nonlinear step functions (such as the sigmoidal function). Furthermore, in 1974, PAUL JOHN WERBOS described a new type of training process where the errors were *backpropagated* through the layers of the network¹ [10]. While WERBOS was the first to describe backpropagation, this approach was also (re)discovered in other research communities. It took until approximately 1985 when DAVID E. RUMMELART, GEOFFREY E. HINTON, and RONALD J. WILLIAMS introduced the algorithm for backpropagation of errors in the context of *multilayer perceptrons* (MLPs) in [8]. This made many of the developments possible, which lead to today’s deep learning.

The following sections start by introducing the most important building blocks for *deep learning* (DL): new types of activation functions and the backpropagation algorithm.

¹ The PhD thesis [10] is the original publication; additionally, the thesis was later published in the book [11].

18.2 Activation Functions

In the previous chapter, we only encountered one type of activation function, the binary step function (or Heaviside function). However, it turns out that for back-propagation, non-zero gradients are required. Also, having a certain degree of nonlinearity in the activation function generally helps with the training and increases the predictive power of a network. In fact, the activation function is the first place where a *non-linearity* can enter the network. Thus, there are many options for mathematically formulating activation functions. One of their common properties is that they all are elementary mathematical functions. They are always monotonic functions, i.e., their function values are always increasing. Additionally, some of them have the property to saturate at large absolute values: they are *squashing functions* that map an arbitrary value range to a bounded output. Figure 18.1 shows an overview of some activation functions and the respective mathematical formulations; further discussions can be found, e.g., in [12] and in the references therein.

Activation Functions



Usually, all hidden layers use the same activation function. The output layer (as we already saw in the previous chapter) may use a different type of activation function. This depends on the kind of prediction (classification, regression) that is to be made. For example, for regression problems, the last layer is a layer with linear activation.

18.2.1 Step Function

The *step function* is the simplest approach, which, however, does not work for multi-class classification problems because it can only give two different outputs, 0 and 1. Additional, the gradient is zero, which poses a problem for back-propagation.

18.2.2 Linear Function

Using a *linear activation function*, which is also called *identity function*, the performance of the neural network will not or only slowly improve due to the constant gradient during the back-propagation: a constant gradient implies that it is insensitive to the value of the input. If all hidden layers have linear activation functions, then each layer is a linear function of the previous one and, effectively, all layers can be replaced by a single layer with linear activation functions.

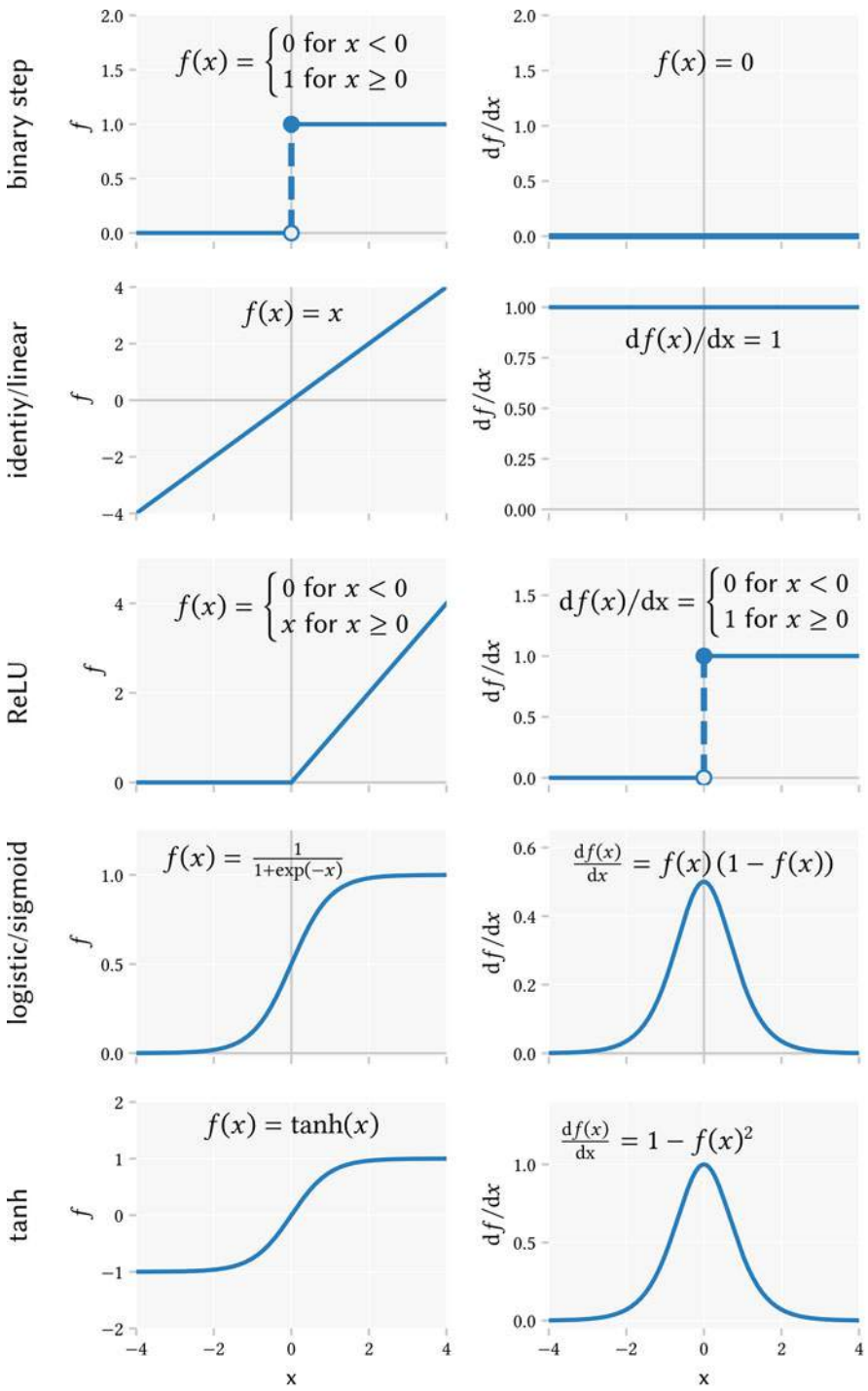


Fig. 18.1 Activation functions (left column) and their derivatives (right column)

18.2.3 Sigmoidal Function

The *logistic* or *sigmoid* activation function is a nonlinear function. The function is non-symmetric with $f(x) > 0$ around $x = 0$. As a consequence, the outputs of all neurons have the same sign, which makes training less stable. Due to its range of the function values of $[0, 1]$, it is used in the context of predicting probabilities. The function has only small gradients further away from the center; the fact that $df(x)/dx = f(x) \cdot (1 - f(x))$ implies that for every iteration the values are getting smaller (below we will discuss this in the context of the “vanishing gradient problem”).

18.2.4 The Family of ReLU Functions

One of the often used type of activation functions today is the *rectified linear unit* (ReLU) function: The ReLU activation function looks extremely simple: it consists of a horizontal line and an 45° -inclined line. It is not differentiable at $x = 0$, which might lead to problems during gradient-based optimization. A property of the ReLU function is that it is scale-invariant. However, one of the side effects of this mathematical formulation is that it does not activate all neurons at the same time; due to the zero gradient, it even might lead to “dead neurons” that *never* get activated. This is also the reason, why we typically prefer to initialize the neurons directly connected to ReLU activation functions with a slightly positive bias by, e.g., adding a small “offset” to the initial random values.

A variant of ReLU that aims to remedy the problem of dead neurons is the *leaky ReLU*. This variant avoids non-zero function values and derivatives by adding a small slope to the region of $x \leq 0$. The slope is the “leakage” parameter:

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases} \quad (18.1)$$

$$\frac{\partial f(x)}{\partial x} = \begin{cases} 1 & \text{if } x > 0, \\ 0.01 & \text{otherwise.} \end{cases} \quad (18.2)$$

The first equation can also be implemented as $f(x) = \max(0.01x, x)$.

Furthermore, the *parameterized ReLU* treats the leakage parameter as a parameter that needs to be identified during the training. These are just a few examples, there are many more formulations, some of which are specialized to particular network architectures.

18.2.5 Hyperbolic Tangent Function

The second, most commonly used type of activation function, is the hyperbolic tangent (tanh) function. It is very similar to the sigmoid function, but it is symmetric around 0. It also has a steeper gradient, which can be seen by the value of the derivative at $x = 0$ in Fig. 18.1. Usually, it performs better than the sigmoid function. As opposed to ReLU, it is not scale-invariant.

18.3 Back-Propagation: Introduction and Example

Already a few years after ROSENBLATT published the *perceptron*, [6], he also published the concept of a *multilayer perceptron* (MLP), which contained an input layer, a hidden layer, and an output layer [7]. The hidden layer was initialized with random values for the weights. However, there was no mechanism such that the hidden layer was able to adjust the weights. Because a DL network has the ability to adjust the weights in the hidden layer, the multilayer perceptron would not yet be considered a proper DL network.

Even though the gradient descent method in the context of training was indirectly established through the delta rule, at this point the concepts of a cost function in conjunction with gradient descent minimization did not exist for *artificial neural networks* (ANNs). The great achievement of RUMMELHARD et al. [8] was to generalize the gradient descent framework to the context of training multilayer neural networks with nonlinear activation functions in a way such that *all* weights get automatically adjusted.

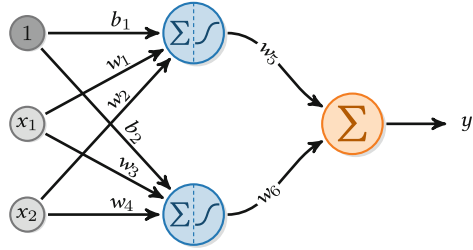
18.3.1 The General Concept

The *back-propagation* algorithm (which is sometimes also called in short “back-prop” algorithm) consists of two processes: the *forward pass* (or propagation phase) where the inputs are propagated through the network and the output predictions are computed, and the *backward pass* during which the gradient of the loss function of the output layer is computed *with respect to all weights*; this gradient is then recursively propagated through the network from the output layer all the way to the input layer. The chain rule for differentiation is key for finding how each individual weight of all hidden layer(s) has to be adjusted (the so-called *weight update*) in order to minimize the loss function.

18.3.2 A First Example

We will now investigate a very simple system, similar to that from Sect. 17.4.1, and introduce the backpropagation formalism. The network has two input nodes, two

Fig. 18.2 Network with one hidden layer and a sigmoidal activation function. The output is just aggregated, and there is no activation function



nodes in the hidden layer, and a single output. The hidden nodes contain sigmoidal activation functions, and the whole network is shown in Fig. 18.2.

Furthermore, we only consider a single data record consisting of two features and a single target variable, $\mathcal{P} = (x_1, x_2; y)$.

Activation Function

The activation functions of the hidden layer both have the following mathematical form:

$$\varphi(z) = \frac{1}{1 + \exp(-z)} \quad (18.3)$$

Loss and Cost Function

Additionally, we need to quantify the “badness” of the performance of the network. This is done using a quadratic *loss function*, in analogy to what has been introduced in Sect. 12.3.3 and in particular in Definition 12.6. For each training example, it is

$$\mathcal{L}(\mathbf{w}; (\mathbf{x}, y)) = \frac{1}{2} \left(y - y^{\text{pred}}(\mathbf{w}; x_1, x_2) \right)^2. \quad (18.4)$$

Note that the factor of $1/2$ is there to simplify the equation during computing the derivative; this does not have any effect on finding the minimum of the cost function below. From this equation, we see that $\mathcal{L}$ depends on the current values of all weights that are varied during the training, as well as on a given record from the training dataset.

The loss function gives a single value for each data record. The *cost function* on the other hand is the average loss for all records of the dataset. Similar to Definition 12.5 we can now define the cost function $\mathcal{J}$. Writing this directly for only a single data record hides some details; thus, we write it first in all generality as the average cost of M data records:

$$\mathcal{J} = \frac{1}{2M} \sum_{i=1}^M \left(y_i - y^{\text{pred}}_i(\mathbf{w}; x_{i1}, x_{i2}) \right)^2, \quad (18.5)$$

which, for a dataset that consists of only one record, would simplify to

$$\mathcal{J} = \frac{1}{2} \left(y - y^{\text{pred}}(\mathbf{w}; x_1, x_2) \right)^2 . \quad (18.6)$$

The first and second features for several data records are given by the column vectors X_1 and X_2 (compare the definition of the feature matrix, Sect. 3.3.3), the vector of all labels is Y . The resulting cost function is then a function that depends on six weights and two bias values as variables:

$$\mathcal{J} = \mathcal{J}(w_1, \dots, w_6, b_1, b_2; X_1, X_2; Y) . \quad (18.7)$$

Words of advice



In the field of deep learning, the difference between cost and loss is very often not recognized, and ML practitioners only talk about the “loss functions.” To be consistent with the formulations from earlier chapters, we will continue to differentiate between cost and loss.

Forward Pass

The feed-forward step consists of evaluating the network prediction y^{pred} for a given input, in analogy to Sect. 17.4.1, together with computing the cost. The quantities can be best computed starting from the output layer and then moving toward the input layer. The cost function can then be written as

$$\mathcal{J} = \frac{1}{2} (y - a_3)^2 , \quad (18.8)$$

where we replaced y^{pred} with the result from the agglomeration function a_3

$$a_3 = b_3 + y_1 w_5 + y_2 w_6 = y^{\text{pred}} , \quad (18.9)$$

This function performs the weighted summation, and y_1 and y_2 are the outputs of the two units of the hidden layer. They can be computed as

$$y_1 = \varphi(a_1) \quad (18.10)$$

$$y_2 = \varphi(a_2) . \quad (18.11)$$

where φ is the sigmoidal activation function. The weighted summations are:

$$a_1 = b_1 + x_1 w_1 + x_2 w_2 \quad (18.12)$$

$$a_2 = b_2 + x_1 w_3 + x_2 w_4 . \quad (18.13)$$

For brevity, we refrained from stating the function arguments. For example, we automatically implied the short notation $a_1 \equiv a_1(w_1, w_2, b_1; x_1, x_2)$.

Backward Pass

Training implies to find the weights that minimize the cost function,

$$\text{Find } w_1, \dots, w_6, b_1, b_2 \text{ such that } \mathcal{J} \rightarrow \text{MIN} . \quad (18.14)$$

Thus, to solve this minimization problem with the *gradient descent* (GD) method, we will need to compute the derivatives $\partial \mathcal{J} / \partial w_1 \dots \partial \mathcal{J} / \partial w_6$, as well as $\partial \mathcal{J} / \partial b_1$ and $\partial \mathcal{J} / \partial b_2$. Again, we start from the output of the network and then move through the network toward the input layer.

We start to compute the derivative w.r.t. one of the weights of the output node using the *chain rule* and Eq. 18.8:

$$\frac{\partial \mathcal{J}}{\partial w_5} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial w_5} = -(y - a_3) \cdot y_1 \quad (18.15)$$

Hint for Writing the Chained Products of Derivatives



Start writing the chain of derivatives from the right to the left. For example, if the partial derivative is w.r.t. w_1 then search for the term that contains w_1 : it is a_1 . Thus the rightmost term is $\partial a_1 / \partial w_1$. Next, watch out for the equation that contains a_1 : it is y_1 , giving you the second derivative term $\partial y_1 / \partial a_1$. Then, search for the equation that contains y_1 : it is a_3 , and we found a third term. It is $\partial a_3 / \partial y_1$. Last but not least, the function that contains a_3 is $\mathcal{J}$, hence, $\partial \mathcal{J} / \partial a_3$ and we're done.

The result is:

$$\frac{\partial \mathcal{J}}{\partial w_1} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$

The derivation of $\partial \mathcal{J} / \partial w_6$ follows in analogy:

$$\frac{\partial \mathcal{J}}{\partial w_6} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial w_6} = -(y - a_3) \cdot y_2 . \quad (18.16)$$

Next, the expressions for $\partial \mathcal{J} / \partial b_2$, $\partial \mathcal{J} / \partial w_3$, and $\partial \mathcal{J} / \partial w_4$ are derived:

$$\frac{\partial \mathcal{J}}{\partial b_2} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_2} \cdot \frac{\partial y_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial b_2} = -(y - a_3) \cdot w_6 \cdot y_2(1 - y_2) \quad (18.17)$$

$$\frac{\partial \mathcal{J}}{\partial w_3} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_2} \cdot \frac{\partial y_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_3} = -(y - a_3) \cdot w_6 \cdot y_2(1 - y_2) \cdot x_1 \quad (18.18)$$

$$\frac{\partial \mathcal{J}}{\partial w_4} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_2} \cdot \frac{\partial y_2}{\partial a_2} \cdot \frac{\partial a_2}{\partial w_4} = -(y - a_3) \cdot w_6 \cdot y_2(1 - y_2) \cdot x_2 \quad (18.19)$$

where we used that for a sigmoidal function f , its derivative is $f'(x) = f(x)(1 - f(x))$, which helped to write

$$\frac{\partial y_2}{\partial a_2} = \frac{\partial \varphi(a_2)}{\partial a_2} = \varphi(a_2)(1 - \varphi(a_2)) = y_2(1 - y_2) . \quad (18.20)$$

The derivation for $\partial \mathcal{J} / \partial b_1$, $\partial \mathcal{J} / \partial w_1$, and $\partial \mathcal{J} / \partial w_2$ follows along the same line and are summarized as

$$\frac{\partial \mathcal{J}}{\partial b_1} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial b_1} = -(y - a_3) \cdot w_5 \cdot y_1(1 - y_1) \quad (18.21)$$

$$\frac{\partial \mathcal{J}}{\partial w_1} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1} = -(y - a_3) \cdot w_5 \cdot y_1(1 - y_1) \cdot x_1 \quad (18.22)$$

$$\frac{\partial \mathcal{J}}{\partial w_2} = \frac{\partial \mathcal{J}}{\partial a_3} \cdot \frac{\partial a_3}{\partial y_1} \cdot \frac{\partial y_1}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_2} = -(y - a_3) \cdot w_5 \cdot y_1(1 - y_1) \cdot x_2 . \quad (18.23)$$

These are all possible “sensitivities” of the cost function w.r.t. the network parameters, and the backward pass is finished.

Update of a Weight in a Gradient Descent Approach

The above results can now be used in a [GD](#) method, where, e.g., the weight w_4 is updated according to

$$w_4^{\text{new}} = w_4^{\text{old}} - \eta \frac{\partial \mathcal{J}}{\partial w_4} , \quad (18.24)$$

where η is the learning rate. Note that if the algorithm is performed with multiple data records, then each record would result in a different weight increment. The weight update has to be done using the mean value of all weight increments. For weight w_j , this is then written as

$$w_j^{\text{new}} = w_j^{\text{old}} - \eta \frac{1}{n} \sum_{i=1}^n \frac{\partial \mathcal{J}(\mathbf{x}_i)}{\partial w_j} , \quad (18.25)$$

where n denotes the number of records. With this, all components for a complete training process are now available. The following Python implementation almost exactly uses the mathematical formulations.

18.3.3 A Complete Python Implementation

In the previous subsection, we derived the formulation for a forward and backward pass under the assumption that there is only one example in the dataset. What would change if, instead, we have n data records? The partial derivative operations will be “simultaneously” applied to all data records, but there are no vector–matrix products involved. Only for computing the cost function, we have to return to the formulation in Eq. 18.5. Therefore, we can directly implement the case of multiple data records using vectorization over data records. However, we will not generalize the formulation to arbitrary numbers of, e.g., input nodes or hidden layers but rather hard-code the network architecture.

Python Listing 18.1 shows a Python implementation of the formulations derived in Sect. 18.3.2, which will be briefly explained in what follows. We assume that the training data are provided in form of two feature vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ together with the target vector $\mathbf{y}$, all of which are numpy arrays. The goal was to keep the implementation as readable as possible. For this we keep all weights and bias value in the sets $\mathbf{w}$ and $\mathbf{b}$ when functions are called. This requires that inside the function we again extract (or “unpack”) the individual weight and bias values from these two variables—and similar, when, e.g., the $dJdw_1$, $dJdw_2$ are returned. Using object oriented programming and classes could help to avoid this kind of “book-keeping overhead.”

- In line 3 we define the sigmoidal function. Due to the fact that we use numpy’s exponential function, it is possible to compute it for all elements of an array.
- Starting with line 6, we define a function for performing the forward pass. After extracting the individual weight and bias variables from $\mathbf{w}$ and $\mathbf{b}$, we can directly generate Python code from the mathematical equations. All operations work for $\mathbf{x}_1$ and $\mathbf{x}_2$ being scalars as well as numpy arrays. The individual weights and biases are always scalar values.
- Predicting a label for given weights, biases, and feature values in Line 16 consists just in a forward pass. The “_ , _ , _ , _” in Line 15 are placeholders’ return values that are not required (but any other variable name would do as well).
- Doing the backward pass in Line 20 does not require anything new. The implementation is again identical to the mathematical formulation and works both for scalar and array data for $\mathbf{x}_1$ and $\mathbf{x}_2$ as all operations are performed element-wise.
- For updating the weights in Line 37 we again start by unpacking the weights, biases, and derivatives. Then, Eq. 18.25 is implemented and the updated weights and biases are returned

```

1 import numpy as np
2
3 def sigmoidal_function(x):
4     return 1 / (1 + np.exp(-x))
5
6 def forward_pass(W, B, X1, X2):
7     (w1, w2, w3, w4, w5, w6), (b1, b2, b3) = W, B
8
9     a1 = b1 + X1 * w1 + X2 * w2
10    a2 = b2 + X1 * w3 + X2 * w4
11    y1 = sigmoidal_function(a1)
12    y2 = sigmoidal_function(a2)
13    a3 = y1 * w5 + y2 * w6 + b3
14    return a1, a2, y1, y2, a3
15
16 def predict(W, B, X1, X2):
17     _, _, _, _, Y_pred = forward_pass(W, B, X1, X2)
18     return Y_pred
19
20 def backward_pass(W, B, X1, X2, Y):
21     w1, w2, w3, w4, w5, w6 = W
22     a1, a2, y1, y2, a3 = forward_pass(W, B, X1, X2)
23
24     dJdb1 = -(Y - a3) * w5 * y1 * (1 - y1)
25     dJdw1 = dJdb1 * X1
26     dJdw2 = dJdb1 * X2
27     dJdb2 = -(Y - a3) * w6 * y2 * (1 - y2)
28     dJdw3 = dJdb2 * X1
29     dJdw4 = dJdb2 * X2
30     dJdb3 = -(Y - a3)
31     dJdw5 = -(Y - a3) * y1
32     dJdw6 = -(Y - a3) * y2
33
34     dJdW, dJdB = (dJdw1, dJdw2, dJdw3, dJdw4, dJdw5, dJdw6), (dJdb1, dJdb2, dJdb3)
35     return dJdW, dJdB
36
37 def update_weights(W, B, dJdW, dJdB, learning_rate):
38     (w1, w2, w3, w4, w5, w6), (b1, b2, b3) = W, B
39     (dJdw1, dJdw2, dJdw3, dJdw4, dJdw5, dJdw6), (dJdb1, dJdb2, dJdb3) = dJdW, dJdB
40
41     w1 -= learning_rate * np.mean(dJdw1)
42     w2 -= learning_rate * np.mean(dJdw2)
43     w3 -= learning_rate * np.mean(dJdw3)
44     w4 -= learning_rate * np.mean(dJdw4)
45     w5 -= learning_rate * np.mean(dJdw5)
46     w6 -= learning_rate * np.mean(dJdw6)
47     b1 -= learning_rate * np.mean(dJdb1)
48     b2 -= learning_rate * np.mean(dJdb2)
49     b3 -= learning_rate * np.mean(dJdb3)
50
51     W_new, B_new = (w1, w2, w3, w4, w5, w6), (b1, b2, b3)
52     return W_new, B_new

```

Python Listing 18.1 Implementation of a network with two input nodes, two nodes in a hidden, layer and a single output node. See the text for more information. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact

With this we can now attempt to use the implementation for a concrete problem: classification using only the first two features of the “Iris dataset.” We import the dataset using the scikit-learn module (or the online resources of this book, or any other resources—as long as `x` contains the four features in columns and `y` is the single target variable) and then extract only those records that belong to the first two classes.

```
In [1]: import numpy as np
        from sklearn.datasets import load_iris

        iris = load_iris()
        X = iris["data"][:, :2]
        Y = iris["target"]

        # only use the first two labels 0 and 1
        X = X[Y < 2]
        Y = Y[Y < 2]
```

The data are split into a training and a testing dataset, using the function `train_test_split` that has already been introduced in Sect. 16.2.2.

```
In [2]: # split into training and testing data
        X_train, Y_train, X_test, Y_test = train_test_split(X, Y)

        # and for convenience define individual features X1 and X2
        X1_train = X_train[:, 0]
        X2_train = X_train[:, 1]
        X1_test = X_test[:, 0]
        X2_test = X_test[:, 1]
```

Next, we initialize the six weights with random values and set the bias values to zero. The number of epochs (i.e., the number of complete forward/backward passes) together with the learning rate are set

```
In [3]: W = 1e-3 * (rng.random(size=6) - 0.5)
        B = np.zeros(3)
        epochs = 600
        learning_rate = 0.2
```

with which now the gradient descent update iterations can be performed:

```
In [4]: mean_losses = []
        for epoch in range(epochs):
            Y_pred = predict(W, B, X1_train, X2_train)
            mean_losses.append(mean_loss(Y_train, Y_pred))

            dJdW, dJdB = backward_pass(W, B, X1_train, X2_train, Y_train)
            W, B = update_weights(W, B, dJdW, dJdB, learning_rate)

        print('MSE difference:', mean_losses[0], mean_losses[-1])
```

```
Out [4]: MSE difference: 0.26667596178013414 0.019281451533922658
```

Comparing the initial training MSE to the final training MSE suggests that at least the numerical scheme is stable and no undefined values or other errors occurred. Since the training dataset is two-dimensional, it is a good approach to visualize the data. Here, we show two different plot types. The first plot just shows the training *mean squared error* (MSE) as a function of the number of epochs:

```
In [5]: fig, ax = plt.subplots()
ax.plot(mean_losses)
ax.set(xlabel='number of epochs', ylabel='mean training loss (MSE)')
```

The second plot is a more complex plot. It is a heat map using `imshow`, which shows the training MSE evaluated at a grid of equidistant points of the feature space. This was superimposed with a scatter plot of the position of the training and testing data. Additionally, the decision boundary was shown where all target values smaller than 0.5 indicated class #1 and all values larger than 0.5 indicate class #2.

```
In [6]: fig, ax = plt.subplots()
ax.set(xlabel='X1', ylabel='X2');

# heatmap with contourplot (showing the decision boundary)
XX1, XX2 = np.meshgrid(np.linspace(4, 7.25), np.linspace(1.5, 4.6))
Y_pred = predict(W, B, XX1, XX2)
im = ax.imshow(Y_pred, origin='lower', vmin=-1, vmax=2, cmap='RdBu_r',
               extent=(XX1.min(), XX1.max(), XX2.min(), XX2.max()))
plt.colorbar(im)
ax.contour(XX1, XX2, Y_pred, levels=[0.5])

# scatter plot
ax.scatter(X1_train, X2_train, c=Y_train, cmap='RdBu_r', label='training')
ax.scatter(X1_test, X2_test, c=Y_test, cmap='RdBu_r', marker='.',
          label='testing')
ax.legend(ncols=2, loc='lower center')
```

The resulting plots are similar to those shown in Fig. 18.3 (plotting the loss curve for the testing follows the same path and is left as an exercise). Additionally, we also showed the evolution of the mean testing loss, which is left as an exercise for the reader.

The learning curve shows that after approximately 400 epochs, a stationary state is reached, both for training and testing. The testing MSE is slightly below the training MSE, but using different random splits does not result in significant

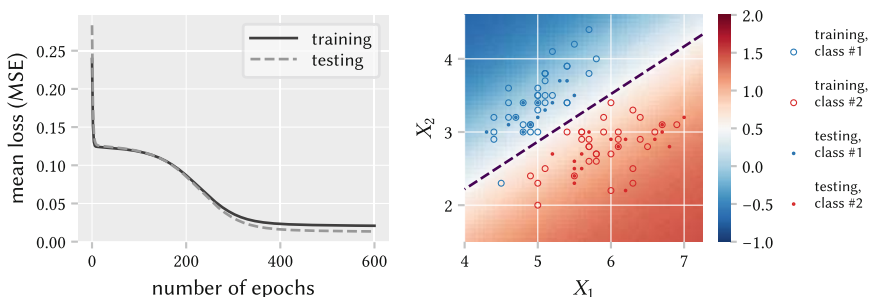


Fig. 18.3 Results from the network with one hidden layer for the first two features of the Iris dataset. The left panel shows the learning curves, the right panel shows the datasets together with the predicted value Y^{pred} and the decision boundary

difference. Here, a cross-validation could be useful as well. Visually inspecting the dataset reveals that the data points should be linearly separable; the decision boundary that was found by the network does this perfectly.

To quantify how well our model performs, we evaluate the metric of true positives, true negatives, false negatives, and false positives, as introduced in Sect. 11.8.1:

```
In [7]: TP = np.sum((Y_pred == 1) & (Y_test == 1))
        TN = np.sum((Y_pred == 0) & (Y_test == 0))
        FP = np.sum((Y_pred == 1) & (Y_test == 0))
        FN = np.sum((Y_pred == 0) & (Y_test == 1))

        print(f'TP: {TP}, FP: {FP}, TN: {TN}, FN: {FN}')
```

```
Out [7]: TP: 17, FP: 0, TN: 13, FN: 0
```

The value of 0.5 is the threshold above which the output is considered to be `True`. The result confirms what we already observed in the right panel of Fig. 18.3: all examples have been correctly predicted. Even though this comes as no surprise since the two classes are linearly separable, it is nonetheless a good validation of our network.

18.4 General Formulation of Backpropagation

In the last section, it became obvious that determining all chained partial derivatives of all layers is very tedious. It is also inflexible as for every new *network architecture* (i.e., the particular combination of different layers and nodes), the derivations have to be performed anew. An alternative way is to treat each layer as a basic building block that takes an input, processes the data, and returns an output. For example, for layer number k the input is $\mathbf{x}^{(k)}$ and the output is $\mathbf{y}^{(k)}$. For simplicity of the formulations, both vectors represent only a single data record and are thus row vectors. Also, we treat the activation functions as an individual layer, the *activation layer* (AL) while the layer where the summation is done is called *fully connected layer* (FCL).

This is also the way how current DL frameworks are designed. Therefore, we will go through these derivations in some more detail.

18.4.1 Forward Pass

Fully Connected Layer

We will now derive the formulations for the forward pass and start with the fully connected layer, which is layer number k . A sketch of the considered situation is shown in Fig. 18.4.

The forward pass works as expected: it takes as input the output from the previous layer (step 1. in Fig. 18.4), computes the values of the target variable (step 2. in

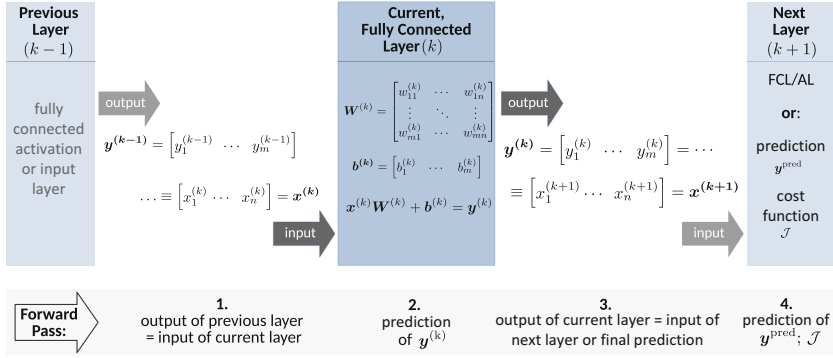


Fig. 18.4 Notations and visualization of the forward pass for a fully connected layer. Note that the sizes n and m of the input and output vectors can change from layer to layer even though we do not explicitly indicate this by subscripts such as (k)

Fig. 18.4), and provides this as the output to the next layer (step 3. in Fig. 18.4). If the next layer is the output layer, then this is the prediction of the network, and we additionally can evaluate the cost function (step 4. in Fig. 18.4).

What is new in this section is the notation of using superscripts to indicate the layer. Thereby, we only have to define each layer type ones and can then use them as basic building blocks for general networks. For example, the output of layer number $k-1$ is given by

$$y^{(k-1)} = [y_1^{(k-1)} \dots y_m^{(k-1)}] \equiv [x_1^{(k)} \dots x_n^{(k)}] = x^{(k)}, \quad (18.26)$$

which serves as the input of layer number k . The number of input nodes of each layer is denoted by n , while the number of output nodes of each layer is m . To be fully consistent, also the m and n would need a superscript to indicate the number of the layer; however, given that for m and n usually the context is clear and for the sake of readability, we will remember that, e.g., the number of input nodes n of a layer k can be different from the number of input nodes n of any other layer, and that writing n in fact would imply $n^{(k)}$ for the k -th layer.

The agglomeration in the layer is then given by

$$x^{(k)} W^{(k)} + b^{(k)} = y^{(k)} \quad (18.27)$$

where the matrix of weights and the vector of bias values of layer k are given by

$$W^{(k)} = \begin{bmatrix} w_{11}^{(k)} & \dots & w_{1m}^{(k)} \\ \vdots & \ddots & \vdots \\ w_{n1}^{(k)} & \dots & w_{nm}^{(k)} \end{bmatrix} \quad \text{and} \quad b^{(k)} = [b_1^{(k)} \dots b_m^{(k)}]. \quad (18.28)$$

The r -th component of $\mathbf{y}^{(k)}$ can then be written as

$$y_r^{(k)} = \begin{bmatrix} x_1^{(k)} & \dots & x_n^{(k)} \end{bmatrix} \begin{bmatrix} w_{1r}^{(k)} \\ \vdots \\ w_{nr}^{(k)} \end{bmatrix} + b_r^{(k)} = x_1^{(k)} w_{1r}^{(k)} + \dots + x_n^{(k)} w_{nr}^{(k)} + b_r^{(k)} . \quad (18.29)$$

Activation Layer

The formulation for the activation layer is rather straightforward, and all that needs to be done, is, to apply the activation function to each input of the layer. For example, for a layer number $k + 1$ it is

$$\mathbf{y}^{(i+1)} = \left[\varphi(x_1^{(i+1)}), \dots, \varphi(x_n^{(i+1)}) \right] = \varphi \left(\mathbf{x}^{(i+1)} \right) , \quad (18.30)$$

where the function φ is applied to each element of $\mathbf{x}$ individually. $\mathbf{x}$ is the input of the activation layer, which is typically given by the output of a fully connected layer. In an activation layer, $\mathbf{y}$ has always the same dimensions as $\mathbf{x}$.

Cost Function

The forward propagation step is then repeated for each layer, until we arrive at the output layer for which we obtain the predicted output $\mathbf{y}$. With this, the cost function can be evaluated; as we only have a single data record, it is simply

$$\mathcal{J}(\mathbf{y}; \mathbf{y}^{\text{train}}) = \frac{1}{2m} \sum_{i=1}^m \left(y^{\text{train}}_i - y^{\text{pred}}_i \right)^2 \quad (18.31)$$

18.4.2 Backward Pass

The backward pass is more complicated than the forward pass and requires the computation of a number of derivatives. The strategy is to establish a backward pass for one layer, which can then be generalized to more complex networks by connecting such layers.

Our point of departure for the derivation is the output layer where we already computed the cost function. The goal is now to use the established framework of a gradient descent method, which is supposed to find values of *all* weights and *all* bias values of the N_{layers} layers in the network such that the cost function is minimized,

$$\text{Find } \left\{ \mathbf{w}^{(k)}, \mathbf{b}^{(k)} \right\}_{k=1 \dots N_{\text{layers}}} \text{ such that } \mathcal{J} \rightarrow \text{MIN} . \quad (18.32)$$

The numerical approach is the same as before: we calculate the sensitivity of the cost function w.r.t. all weights and bias values,

$$\frac{d\mathcal{J}}{dw_{ij}^{(k)}} \quad \text{and} \quad \frac{d\mathcal{J}}{db_j^{(k)}} , \quad (18.33)$$

and use this to define an update increment for all weights and bias values. While computing these derivatives manually is possible in case that there is only one fully connected layer (as we saw in the previous chapter), for networks with several layers, this is more difficult as one layer depends on the output of the previous layer, and we thus have an implicit dependency of the cost function on the weights. This is where the *chain rule* for a function that implicitly depends on a third variable through two variables will come into play:

Definition 18.1. Chain Rule If f is a function of two variables, $f(y_1, y_2)$, and each of the y_1 and y_2 are again functions of a variable x , then it is

$$\frac{df}{dx} = \frac{\partial f}{\partial y_1} \frac{\partial y_1}{\partial x} + \frac{\partial f}{\partial y_2} \frac{\partial y_2}{\partial x} . \quad (18.34)$$

In our case, the function under consideration is the cost function $\mathcal{J}$, and we are looking for the derivatives w.r.t. the weights and biases, see Eq. 18.33. However, we are only able to obtain the derivatives w.r.t. to the final output $y^{(k)}$, because this is the only variable that shows up in Eq. 18.31. What other information could be used? We already know that $\mathbf{y}$ implicitly depends also on the weights and bias values. Thus, we can write the first term from Eq. 18.33 using the chain rule as

$$\begin{aligned} \frac{d\mathcal{J}}{dw_{ij}^{(k)}} &= \frac{\partial \mathcal{J}}{\partial y_1^{(k)}} \frac{\partial y_1^{(k)}}{\partial w_{ij}^{(k)}} + \cdots + \frac{\partial \mathcal{J}}{\partial y_m^{(k)}} \frac{\partial y_m^{(k)}}{\partial w_{ij}^{(k)}} \\ &= \sum_{r=1}^m \frac{\partial \mathcal{J}}{\partial y_r^{(k)}} \frac{\partial y_r^{(k)}}{\partial w_{ij}^{(k)}} . \end{aligned} \quad (18.35)$$

Each of the summands in Eq. 18.35 consists of two factors. The first terms *only* can be easily computed for the final layer using the definition of the cost function. For example, for the r -th component of $\mathbf{y}$ we get:

$$\frac{\partial \mathcal{J}}{\partial y_r^{(k)}} = -\frac{1}{m} \left(y^{\text{train}}_r - y_r \right) , \quad (18.36)$$

where m is the number of output nodes which is identical to the number of elements of $\mathbf{y}$. For all other layers, we first have to propagate the error from the output layer to the respective layer of interest. For doing so, the chain rule comes in handy again,

and we can write:

$$\frac{d\mathcal{J}}{dx_i^{(k)}} = \sum_{r=1}^m \frac{\partial \mathcal{J}}{\partial y_r^{(k)}} \frac{\partial y_r^{(k)}}{\partial x_i^{(k)}} \quad \text{or} \quad \frac{d\mathcal{J}}{d\mathbf{x}^{(k)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{x}^{(k)}}. \quad (18.37)$$

Next, we realize that the output of layer $(k - 1)$ serves as the input of layer k . With this we are now able to rewrite Eq. 18.37 as

$$\frac{d\mathcal{J}}{d\mathbf{y}^{(k-1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{x}^{(k)}}, \quad (18.38)$$

which also very clearly shows where “back-propagation” got its name from: we start from the end of the network and iteratively compute the derivative of the cost function w.r.t. the weights for layer $k - 1$ using the already known value from the previous layer k . Thereby, the change of the overall error is moving backward through all layers.

For the following derivations, we again differentiate between a fully connected layer and an activation layer.

Fully Connected Layer

First, we continue with the previous expression, Eq. 18.38. For a fully connected layer, we can specify $\partial \mathbf{y} / \partial \mathbf{x}$

$$\frac{d\mathcal{J}}{d\mathbf{y}^{(k-1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \mathbf{W}^{(k)}. \quad (18.39)$$

Next, the second term in Eq. 18.35 still requires to be determined. We start from explicitly rewriting them in terms of the components of Eq. 18.29:

$$\begin{aligned} \frac{\partial y_1^{(k)}}{\partial w_{ij}^{(k)}} &= \frac{\partial}{\partial w_{ij}^{(k)}} \left(x_1^{(k)} w_{11}^{(k)} + x_2^{(k)} w_{21}^{(k)} + \dots + x_n^{(k)} w_{n1}^{(k)} + b_1^{(k)} \right) \\ \frac{\partial y_2^{(k)}}{\partial w_{ij}^{(k)}} &= \frac{\partial}{\partial w_{ij}^{(k)}} \left(x_1^{(k)} w_{12}^{(k)} + x_2^{(k)} w_{22}^{(k)} + \dots + x_n^{(k)} w_{n2}^{(k)} + b_2^{(k)} \right) \\ &\vdots \\ \frac{\partial y_m^{(k)}}{\partial w_{ij}^{(k)}} &= \frac{\partial}{\partial w_{ij}^{(k)}} \left(x_1^{(k)} w_{1m}^{(k)} + x_2^{(k)} w_{2m}^{(k)} + \dots + x_n^{(k)} w_{nm}^{(k)} + b_m^{(k)} \right). \end{aligned} \quad (18.40)$$

All derivatives are zero except for the only term containing a w_{ij} :

$$\frac{\partial y_j^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial}{\partial w_{ij}^{(k)}} \left(x_i^{(k)} w_{ij}^{(k)} \right) = x_i^{(k)} \quad (18.41)$$

which is the expression that contains the j -th component of $\mathbf{y}$. Hence, we can simplify Eq. 18.35 to

$$\frac{d\mathcal{J}}{d\mathbf{w}_{ij}^{(k)}} = \frac{\partial \mathcal{J}}{\partial y_j^{(k)}} x_i^{(k)}, \quad (18.42)$$

which can also be written as the outer product of two vectors

$$\frac{d\mathcal{J}}{d\mathbf{W}^{(k)}} = \mathbf{x}^{(k)} \otimes \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \equiv \left(\mathbf{x}^{(k)} \right)^\top \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}}, \quad (18.43)$$

where the last term rewrites the outer product symbol $\otimes$ as the product of a column vector with a row vector, resulting in a matrix. The derivation for the derivative w.r.t. the bias follows nearly the identical path, and we end up with

$$\frac{\partial y_j^{(k)}}{\partial b_j^{(k)}} = 1, \quad (18.44)$$

such that the sensitivity of the cost function w.r.t. the bias is given by

$$\frac{d\mathcal{J}}{d\mathbf{b}^{(k)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}}. \quad (18.45)$$

We can now summarize all details for a back-propagation step for a fully connected layer in Algorithm 18.1:

Algorithm 18.1: Backpropagation Step of a Fully Connected Layer



The following are the required steps for backpropagation:

1. If the layer $k + 1$ is the final, output layer, compute

$$\frac{\partial \mathcal{J}}{\partial \mathbf{y}} = -\frac{1}{m} (\mathbf{y}^{\text{train}} - \mathbf{y}); \quad (18.46)$$

otherwise just use the value $\partial \mathcal{J} / \partial \mathbf{y}^{(k+1)}$ from the previous layer in step 2.

2. Propagate the error from layer $k + 1$ to layer k :

$$\frac{d\mathcal{J}}{d\mathbf{y}^{(k)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k+1)}} \mathbf{W}^{(k)}. \quad (18.47)$$

3. Compute the sensitivity of the cost w.r.t. the weights and bias values of the current layer

$$\frac{d\mathcal{J}}{d\mathbf{w}^{(k)}} = \mathbf{x}^{(k)} \otimes \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \quad \text{and} \quad \frac{d\mathcal{J}}{d\mathbf{b}^{(k)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \quad (18.48)$$

4. Update the weights and biases using the GD methods

$$\mathbf{w}^{(k)} \leftarrow \mathbf{w}^{(k)} - \frac{\eta}{N} \frac{d\mathcal{J}}{d\mathbf{w}^{(k)}} \quad \text{and} \quad \mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k)} - \frac{\eta}{N} \frac{d\mathcal{J}}{d\mathbf{b}^{(k)}} \quad (18.49)$$

with the learning rate η and N the number of data records.

5. Continue to the next layer, $(k) \Leftrightarrow (k - 1)$.

Activation Layer

The back-propagation of the errors for one activation layer is significantly easier. We again start from Eq. 18.38 for which we need to specify $\partial \mathbf{y} / \partial \mathbf{x}$. Recall the definition of the forward pass of the activation layer in Eq. 18.30, which states that the output is obtained from applying the activation function on each of the entries of $\mathbf{x}$ individually. The derivative then reads

$$\frac{d\mathcal{J}}{d\mathbf{y}^{(k-1)}} = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}} \frac{\partial \varphi(\mathbf{x}^{(k)})}{\partial \mathbf{x}^{(k)}} \quad (18.50)$$

$$= \left[\frac{\partial \mathcal{J}}{\partial y_1^{(k)}} \frac{\partial \varphi(x_1^{(k)})}{\partial x_1^{(k)}}, \dots, \frac{\partial \mathcal{J}}{\partial y_n^{(k)}} \frac{\partial \varphi(x_n^{(k)})}{\partial x_n^{(k)}} \right] = \frac{\partial \mathcal{J}}{\partial \mathbf{y}^{(k)}}. \quad (18.51)$$

Thus, the sensitivity of the error w.r.t. the output is not changed by the activation layer. Note that the product in Eq. 18.50 is the element-wise (or HADAMARD) product as both terms result in row vectors. Nothing else needs to be done here as there are no trainable weights or biases.

18.5 Python Implementation and Example for the Fully Connected Network

The algorithm and the data structure for this network are already fairly complex. As a consequence, the Python code is also getting more lengthy and requires vectorization of the numpy operations. Therefore, we are approaching a point at which showing code and commenting it are no longer feasible in this text. Nonetheless, we will do this for the fully connected network, and, based on this, explain the design strategy and the implementation details. For the Python

implementation of all subsequent network architectures and the corresponding DL experiments, we refer the reader to the accompanying website.

18.5.1 Python Implementation

The Python implementation can be best done in an object-oriented approach using classes and inheritance,² because each activation layer of a network has always the same structure and needs to perform the same computations as derived above. Additionally, different layers must be able to keep track of their “state,” governed by the input values and, in the case of a fully connected layer, also of the values of weights and biases. Thus, each fully connected layer is an instantiation of the same class. The same also holds for the activation layer. Since we want to ensure that all layers should have the same “structure,” in terms of input and output of the functions (or rather methods, as it is called in the context of a class), we create an *abstract base class* from which both the activation layer and the fully connected layer are derived:

```
In [1]: from abc import ABC # For implementing an abstract base class (ABC)
import numpy as np

class NNLayer(ABC):
    def __init__(self):
        self.x = np.empty()
        self.y = np.empty()

    def feed_forward(self, x):
        raise NotImplementedError("Any layer needs to implement the \
            feed_forward method.")

    def backward_propagation(self, dJdy, learning_rate):
        raise NotImplementedError("Any layer needs to implement the \
            backward_propagation method.")
```

Thereby, we ensure that any derived class implements the feed-forward method and the backward propagation method.

We now begin with the concrete implementation of the fully connected layer, which we will explain in two steps. `FullyConnectedLayer` derived from the abstract base class `NNLayer` and therefore inherits all attributes and methods. The `__init__` method initializes the random number generator using the integer value `seed` as a seed. It also initializes the weights and biases with random values. The “state” of the class (i.e., the member variables `x` and `y`) is set during the forward pass. Note, that all numpy operations are vectorized and need to have correct dimensions; for brevity, we do not ensure this.

² We are fully aware that not everyone is familiar with object-oriented programming, and, so far, we entirely avoided it. However, the object-oriented code is, in our opinion, much more readable than a code written with functional programming. For those who are looking for an introduction, [3] is one out of many good introductions.

```
In [2]: class FullyConnectedLayer(NNLayer):
def __init__(self, n_inputs, n_outputs, seed=None):
    rng = np.random.default_rng(seed)
    self.weights = rng.random(size=(n_inputs, n_outputs)) - 0.5
    self.biases = rng.random(size=(1, n_outputs)) - 0.5

def feed_forward(self, x):
    self.x = np.atleast_2d(x)
    self.y = np.dot(self.x, self.weights) + self.biases
    return self.y

def backward_propagation(self, dJdy, learning_rate): ...
```

The implementation of the backward propagation is shown in the next cell. The class method implements the sensitivity from step 3 as well as the update step 4 from the above algorithm Algorithm 18.1. If we would not implement these methods, the corresponding methods of the base class (cf. In [1]:) would be called. This causes an exception to be raised and the program stops.

```
In [2]: class FullyConnectedLayer(NNLayer):
def __init__(self, n_inputs, n_outputs, seed=None): ...
def feed_forward(self, x): ...

def backward_propagation(self, dJdy, learning_rate):
    dJdW = np.dot(self.x.T, dJdy)
    dJdb = np.dot(np.ones(self.x.shape[0]), dJdy)
    self.weights -= learning_rate * dJdW
    self.biases -= learning_rate * dJdb

    dJdy_prev = np.dot(dJdy, self.weights.T)
    return dJdy_prev
```

As a preparation for the activation layer, we now define a activation function and its derivative.

```
In [3]: def tanh_function(x):
return np.tanh(x)

def tanh_derivative(x):
return 1 - np.tanh(x) ** 2
```

We are now able to implement the activation layer, which is again derived from the abstract base class `NNLayer`. The specific activation function is hard-coded in the `__init__` method, because all activation layers should use the same activation function. The feed-forward and the backward propagation methods follow the mathematical form.

```
In [4]: class ActivationLayer(NNLayer):
        def __init__(self):
            self.phi = tanh_function
            self.dphidx = tanh_derivative

        def feed_forward(self, x):
            self.x = x
            self.y = self.phi(self.x)
            return self.y

        def backward_propagation(self, dJdy, learning_rate):
            return self.dphidx(self.x) * dJdy
```

For later use during the training, we now implement the cost function and its derivative based on the [MSE](#).³

```
In [5]: def MSE(y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)

        def dMSE_dy(y_true, y_pred):
            return 2 / y_pred.size * (y_pred - y_true)
```

We are now ready to implement the `Network` class, which contains functionality for assembling the network, for training, and for predicting. Note that the following three code snippets always show the same class but hide the methods that are not being explained, indicated by the three dots. We start by storing a reference to the cost function and its derivative, which are to be defined during creation of the network. All layers are simply stored in a list by the method `add_layer`.

```
In [6]: class Network:
        def __init__(self, cost_function, derivative_of_cost):
            self.layers = []
            self.cost_function = cost_function
            self.derivative_of_cost = derivative_of_cost

        def add_layer(self, layer):
            self.layers.append(layer)

        def train(self, X_train, Y_train, epochs, learning_rate): ...

        def predict(self, X): ...
```

The method that does the actual training needs some more explanations. The training method takes the training dataset, the number of training epochs (i.e., the number of iterations), and the learning rate. During one epoch, it iterates through the whole training dataset and for each pair `x_train, y_train`, it performs the forward propagation through *all* layers. The line `y_pred = x_train` might need a short explanation: in the following `for` loop, the forward propagation is done by using the output of the previous as the input of the current layer. This procedure

³ Sometimes, a factor of 2 be found in the [MSE](#). However, this factor does not affect the training behavior if the learning rate is scaled accordingly.

is sequentially done using the variable `y_pred`, which is why `y_pred` needs to be properly initialized with the output of the “previous” layer, the inputs `x_train`. At the end of this sequence, we obtain the cost function for the prediction of the output layer.

Next, the errors are back-propagated through all layers, which adjusts all weights and biases (see the back-propagation method in the above code cell “In [2]:”). At the end of each epoch, the value of the cost function is stored in the list `all_costs`, which will be returned at the end of the training. It then can be used, e.g., for plotting the training error.

```
In [6]: class Network:
        def __init__(self, cost_function, derivative_of_cost): ...
        def add_layer(self, layer): ...

        def train(self, X_train, Y_train, epochs, learning_rate):
            all_costs = []
            for i in range(epochs):
                cost = 0
                for x_train, y_train in zip(X_train, Y_train):
                    # compute prediction through forward propagations
                    y_pred = x_train
                    for layer in self.layers:
                        y_pred = layer.feed_forward(y_pred)

                    cost += self.cost_function(y_train, y_pred)

                    # perform backprop of errors
                    error = self.derivative_of_cost(y_train, y_pred)
                    for layer in reversed(self.layers):
                        error = layer.backward_propagation(error,
                                                            learning_rate)

                all_costs.append(cost / X_train.shape[0])
            return all_costs

        def predict(self, X): ...
```

In this part of the code, the variable `all_costs` is a list of the cost values for each epoch that can then be used for plotting. Since the training sometimes takes longer, it can be useful to get some feedback about the progress. The Python package `tqdm` is very useful for this purpose. For example, in a jupyter notebook, after importing `tqdm` with `from tqdm.notebook import trange`, you can replace `range` with `trange` and then will see a continuously updated progress bar.

The last method of the `Network` class is used for making predictions. It takes a feature matrix which, as usual, has records in rows, and each column is a feature. We iterate over all records and for each of them compute the output by a complete forward pass. For consistency, the resulting prediction is returned as a numpy array and not as a list.

```
In [6]: class Network:
        def __init__(self, cost_function, derivative_of_cost): ...
        def add_layer(self, layer): ...
        def train(self, X_train, Y_train, epochs, learning_rate): ...

        def predict(self, X):
            Y = []
            for x in X:
                y = x
                for layer in self.layers:
                    y = layer.feed_forward(y)
                Y.append(y)
            return np.array(Y)
```

With this, our implementation is finished, and we can perform a short test.

18.5.2 Application: Identification of Handwritten Digits

We use the dataset of handwritten digits, DS-2-small (UCI Handwritten Digits), and seek to find the labels 0...9 for each image in an supervised learning approach. We start by importing the data. Note that the dataset already contains the “flattened” images, i.e., each $N \times N$ image array was converted into a long one-dimensional array of $N \times N$ elements. We then perform *one-hot encoding* (OHE) (cf. Sect. 16.1.1) such that we can have an output layer with 10 units. The dataset is then split into a training and a testing portion with the function introduced in Sect. 16.2.2.

```
In [7]: from sklearn.datasets import load_digits
        digits = load_digits()
        X = digits.data / 16. # Scale the feature value range to [0, 1]

        # Target variable: One-hot-encoding: map each number of the interval [0,9]
        # 9 zeros and a one, such that, e.g., 2 is mapped to [0, 0, 1, 0, 0, 0, 0
        Y = digits.target
        Y = np.eye(10, dtype=int)[Y]

        X_train, Y_train, X_test, Y_test = \
            train_test_split(X, Y, fraction=0.7, seed=None)
```

We can now assemble the network. We chose an architecture consisting of as many input nodes as there are features, which is why in the code we use `X_train.shape[-1]`. These are 64 nodes, since the images contain $8 \times 8 = 64$ pixel. The hidden layer has 80 units. The next fully connected layer reduces the number of nodes to 40, the next but one reduced them to 10, which are the number of output nodes, consistent with the one-hot encoding. Finally, we train the model for 200 epochs and with a learning rate of 0.1. These are values that were experimentally found. A good strategy is to start with a few epochs and a learning rate in between 0.001 and 0.1. If the resulting MSE (shown as output of the last cell) does not blow up, then one could increase the number of epochs or increase the learning rate.

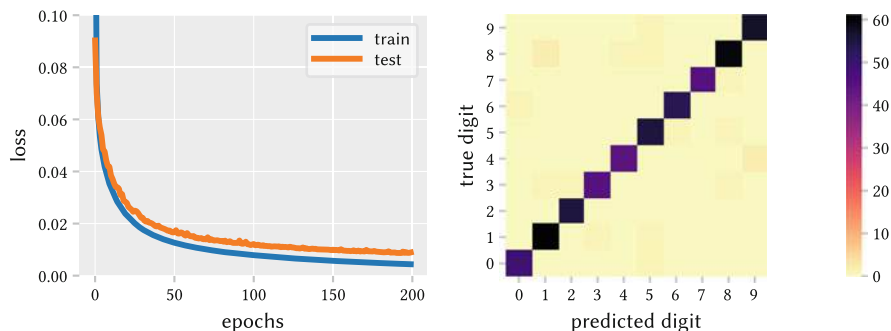


Fig. 18.5 Classification of the handwritten digits dataset using the implementation of the fully connected network. The left panel shows the training and testing error, the right panel shows the confusion matrix

```
In [8]: nn = Network(MSE, dMSE_dy)
nn.add_layer(FullyConnectedLayer(X_train.shape[-1], 80))
nn.add_layer(ActivationLayer())
nn.add_layer(FullyConnectedLayer(80, 40))
nn.add_layer(ActivationLayer())
nn.add_layer(FullyConnectedLayer(40, 10))
nn.add_layer(ActivationLayer())

train_mse = nn.train(X_train, Y_train, epochs=200, learning_rate=0.01)
train_mse[0], train_mse[-1]
```

```
Out [8]: (0.35482118936344176, 0.0018668214460151819)
```

There are (at least) two very useful methods to investigate the performance of the training. The first is to simply plot the training and testing error, as shown in Fig. 18.5.

It is important to ascertain that not only the training error decreases but also that the testing error decreases. With the above implementation, only the training error can directly be plotted. The testing MSE is obtained by using the weights and biases of each epoch for making predictions on the testing dataset. This is left as an exercise.

Creating the confusion matrix comes down to counting those digits that got correctly predicted. In the next Python cell, `yp` is an array of 10 entries (one for each digit, i.e., output node). Finding the one that has the largest value gives us the respective digit. This is done, using the `np.argmax` function.

```
In [9]: confusion_matrix = np.zeros((10, 10), dtype=int)
for yt, yp in zip(Y_test, Y_pred):
    pred_digit = np.argmax(yp)
    true_digit = np.argmax(yt)
    confusion_matrix[true_digit, pred_digit] += 1
```

We can now plot the confusion matrix, as shown in the next Python cell, where we used `imshow` to plot the two-dimensional 10×10 array.

```
In [10]: fig, ax = plt.subplots()
         im = ax.imshow(confusion_matrix, cmap='magma_r', origin='lower')
         plt.colorbar(im)
         ax.set(xlabel='predicted digit', ylabel='true digit',
               xticks=range(10), yticks=range(10));
```

We observe that both the train as well as the test loss decrease quite smoothly. The test loss is only slightly higher than the train loss. Both losses continue to very slowly decrease. The classification accuracy (for this particular random split into training and testing data) is after 200 epochs already at 96.5%, and this doesn't increase significantly even if we train for another 1000 epochs. The confusion matrix also shows a very good pattern, and only few images are misclassified. We observe that the two most misclassified digits are the 4 (which is identified as a 9) and the 8 (which is identified as a 1 but also as a 4 or 5). Quite surprising, on a first look, this agree only partially with the results from the principal component analysis in Fig. 15.9. However, the dataset investigated there was the full MNIST dataset DS-2 (MNIST) and not the DS-2-small (UCI Handwritten Digits) used here. Furthermore, there are only very few images misclassified at all, and these might be among the extreme values, hidden behind other data points in Fig. 15.9. There is an exercise that gives some hint on how to investigate this observation, cf. Exercise 18.11.

Machine Learning vs. Deep Learning



The above example also demonstrates a key difference between DL and classical ML: while in classical (or statistical) ML we need features, DL does not require features. We could simply use every pixel of a whole image as the input.

18.6 Further Concepts and Techniques

The fully connected neural network introduced above is already quite powerful and can perform complex tasks that would be difficult to achieve with classical ML methods. Nonetheless, there are a number of measures that can be implemented, in order to enhance the performance. We will now introduce some of them briefly.

18.6.1 Mini-Batches

There are two obvious ways for calculating the loss: we can either calculate it for one data record, or we could calculate it for the whole dataset. The former would

not help to improve the performance w.r.t. the *whole* dataset and would result in very unstable gradients. The latter may take a long time and require a huge amount of memory. A *min-batch* is a compromise between the two extremes. A mini-batch contains only a subset of the whole training data, with the number of contained data records called the *batch size*. The batch size is a hyperparameter to the model. During training, all mini-batches are then processed, until the whole training dataset is covered. The step-by-step algorithm is shown in Algorithm 18.2.

Algorithm 18.2: Training with Mini-batches



For the use of mini-batches, the following steps need to be taken:

1. shuffle the training dataset
2. split the training dataset such that each “fold” has as many records as given by the mini-batch size.
3. pick the first mini-batch
4. do a forward and backward pass simultaneously with all data of the current mini-batch
5. sum up the average contribution of the training cost for the mini-batch
6. choose the next mini-batch and go to step 4 until all mini-batches are processed.

Choosing a good batch size is essential for an efficient and accurate training process. This is another parameter that we, as the DL modeler, need to fix. Exercise 18.7 considers this aspect further.

18.6.2 Batch Normalization

When mini-batches are used, each of them is described by different summary statistics. *Batch normalization* is a method that centers and normalizes the data of a mini-batch and does so for each (or at least most of the) layers [4]. This is done just before the input is given to the activation functions, which seems to reduce fluctuations in the hidden layers activations. In practice, batch normalization is added in form of an extra layer. By *centering and normalizing*, we denote the transformation,

$$\hat{\mathbf{x}}^{(k)} = \frac{\mathbf{x}^{(k)} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}}, \quad (18.52)$$

where $\mathcal{B}$ denotes the minibatch under consideration and $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ are the respective values of the mean and standard deviation.

However, such a normalization would imply that from a sigmoidal activation function only the linear regime would be used. Thus, two additional parameters are

introduced for each layer k , the scale $\gamma^{(k)}$ and the shift of the normalized vector $\beta^{(k)}$ with

$$\tilde{\mathbf{x}}^{(k)} = \gamma^{(k)} \hat{\mathbf{x}}^{(k)} + \beta^{(k)} . \quad (18.53)$$

The two new parameter are not minibatch-specific and are learned along with the other weights and biases of the model. This transformation even allows to approximately undo the centering and normalization, if this would be the optimal approach. However, the new parameter are directly able to “tune” the variance and the mean, which are parameters that are otherwise only indirectly accessible by the network. This can significantly change the training behavior.

While there are still some speculations about the mathematical details of why this works, there is significant evidence that batch normalization strongly improves the performance of a network. In particular, the learning rate can be chosen much more generous.

18.6.3 Weight Initialization

So far, we have always initialized the weights with small random numbers, e.g., in the above example, taken from a uniform distribution with a range of $[-0.5, 0.5]$. We have already seen in Sect. 12.3.7 that during a minimization, such as (stochastic) gradient descent, the initial values can make a difference. In a deep neural net, the initialization of the weights may have a different influence depending on which kind of activation function is used. This also was recognized by XAVIER GLOROT and coworker [1], who suggested to scale the uniform distributions when using a linear activation. That strategy is known as the *Xavier initialization* for which the weights w are sampled from the following symmetric distribution:

$$w \sim \text{U} \left(-1/\sqrt{n}, +1/\sqrt{n} \right) , \quad (18.54)$$

where n is the number of input nodes and $\text{U}(a, b)$ denotes the uniform distribution within the limits a and b . As a consequence of the dependency on the number of input nodes, the range of the random values is larger for smaller n . Figure 18.6 shows the distribution of random samples as a function of the number of input nodes.

A variant of this initialization is the *normalized Xavier initialization*, which additionally takes the number of outputs m into account. The number of outputs are usually identical to the number of units in the layer under consideration. Then, the weights w are sampled from the following distribution:

$$w \sim \text{U} \left(-\sqrt{6}/\sqrt{n+m}, +\sqrt{6}/\sqrt{n+m} \right) . \quad (18.55)$$

The two variants of the Xavier initialization work well for sigmoidal and tanh activation functions (even though, the former is only rarely used, today).

Fig. 18.6 Xavier initialization: the range of the random uniform distribution is chosen according to the number of input nodes. For each value of n , 40 random values are drawn

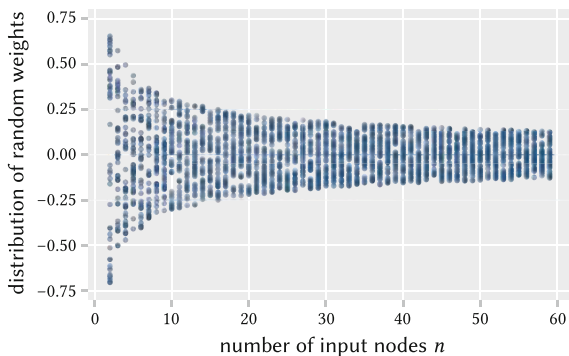
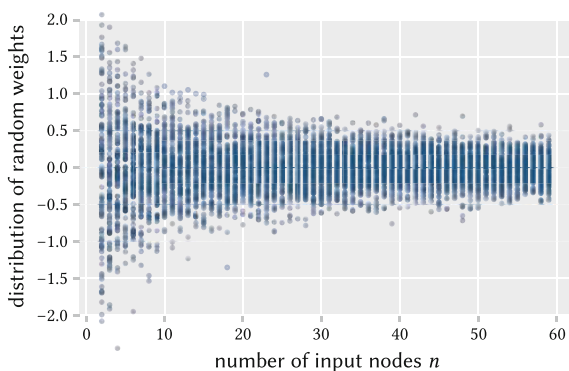


Fig. 18.7 He initialization: as opposed to the Xavier initialization, here a normal distribution is chosen to be sampled from. For each value of n , 100 random values are drawn



Words of advice



The two variants of the Xavier initialization do not work well together with the ReLU activation.

If the ReLU activation function is used, then the alternative formulation, introduced by KAIMING HE [2] is preferred. In this case, the random weight values are drawn from a centered normal distribution:

$$w \sim \mathcal{N}(\mu = 0, \sigma = \sqrt{2/n}) \quad (18.56)$$

Figure 18.7 shows the distribution of random weight values as a function of the number of input nodes.

Note, that the range of the weight values for small values of n is much larger than for the Xavier initialization. For larger n , the range of the values stays approximately constant (at least over a reasonable numbers of input nodes).

18.7 Less Is More: The Concept of Dropout

A serious problem of networks with large numbers of weights is overfitting. This applies in particular to fully connected networks, which, by construction, contain a maximum number of possible weights and thereby tend to be slow during training. *Dropout* is a technique for remedying this, first published in 2014 by Srivastava et al. [9]. The idea of dropout is to randomly drop nodes from the network and also to remove the respective connections. Such a kind of training is then performed for a number of “thinned” networks, each of which uses a random dropout. During testing, a fully connected network is used that represents the average of the trained networks (according to the probability with which a unit is found in one of the thinned realizations). Figure 18.8 shows a sketch of a network with two hidden layers.

The original publication [9] proposed to additionally include the units of the input layer in the set of those that can be randomly removed (they proposed a retention rate of 50% for the units in hidden layers and 80% (i.e. only every fifth unit is dropped) for those in the input layers). There are, however, situations where thinning the input layer might not be the most appropriate approach. For example, if the input consists of one-hot encoded categorical data, in cases where the number of inputs are very small, or if each input carries a unique and significant amount of information, one might want to keep all input nodes.

Why is the dropout technique useful? The quick answer is that it adds a new type of randomness to the training process, which is, generally speaking, always beneficial against overfitting. Part of the explanation is also that removing random units and using such network realizations in an ensemble, acts as a *stochastic regularization*. On a technical level, dropout is implemented by sampling, e.g., from a BERNOULLI distribution (cf. Sect. 10.3) and using the result to determine if a unit is taken into account during forward propagation. E.g., for layer k and the i -th unit

$$r_i^{(k)} \sim \text{Bernoulli}(p) \quad (18.57)$$

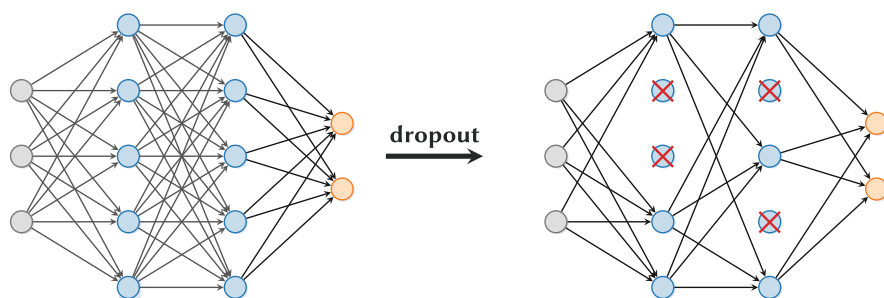


Fig. 18.8 “Drop out” transforms a fully connected network into a network where not all nodes are fully connected. This is achieved by randomly removing nodes from the network. Here, this was only applied to the hidden layers

$$\tilde{\mathbf{x}}^{(k)} = \mathbf{r}^{(k)} \odot \mathbf{x}^{(k)} \quad (18.58)$$

$$y_i^{k+1} = \mathbf{w}_i^{(k+1)} \cdot \tilde{\mathbf{x}}^{(k)} + b_i^{(k+1)} \quad (18.59)$$

where the parameter p of the BERNOULLI distribution is chosen according to the desired retention rate. Thus, $r_i^{(k)}$ effectively “switches” units on or off, ensuring that the desired percentage of units is disabled. The thinned output of this layer is used as input for the following layer. Effectively, the dropout network is one of the 2^n possible “sample” of the fully connected network, with n being the number of units. Ideally, one would then like to obtain a sample average of all 2^n networks. As this is usually unfeasible due to computational cost, the solution is that during testing, each weight obtained during training, is scaled by the probability p , and only a *single* fully connected network is used.

In terms of an implementation, taking out individual units amounts to removing the respective rows and columns from the linear systems of equations. This may significantly reduce the size of the minimization problem. Furthermore, all the other techniques used in “regular” networks, such as momentum and weight decay, seem to work nicely for dropout networks as well. However, keep in mind that this requires that the network is adjusted during the testing phase.

Therefore, for implementation reasons, often a different type of dropout is used, the *inverted dropout*. There, the scaling by the probability is changed to a scaling during the training, but with the inverse probability $1/p$.

When the dropout strategy was developed around 2012, it gave the development and the performance of DL method a significant boost; it is one of the major contributors that helped DL to become able to solve highly complex problems, sometimes even with super-human performance.

18.8 Example: Microstructure Classification and Property Prediction

As an example, we use the MDS-2 light (Ising Model, small images) dataset for the task of predicting if an image with a particular microstructure corresponds to a temperature that is above or below the Curie temperature (cf. Sect. 4.3 for further details). For this, we define a network that consists of a fully connected layer that takes the input (the image in form of a “flattened” vector), which is reduced to 128 output nodes. The following layer has 128 input and 64 output nodes, 64 and 32 input/output nodes, and finally 32 input and 2 output nodes that give the class label of the image for the one-hot encoding. Thus, the model has altogether approximately 43,000 weights to be determined. Figure 18.9 shows the model performance.

We observe that the model reaches a prediction accuracy of around 80%. Comparing this to the baseline of $\approx 50\%$ for a binary classification with approximately balanced classes (cf. Sect. 16.3), this represents a good (but not a great) accuracy.

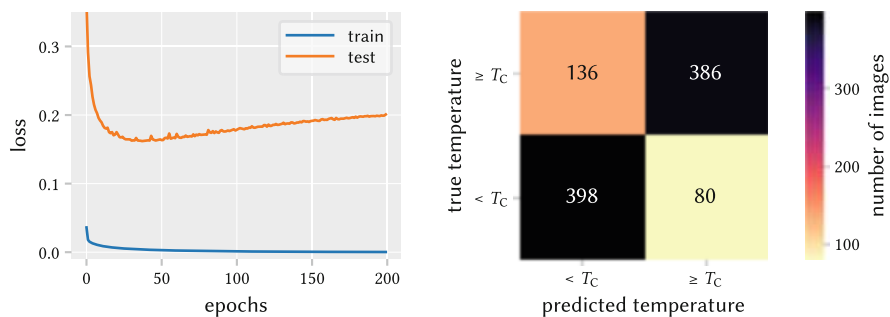


Fig. 18.9 Classification of the Ising microstructures w.r.t. to temperature. The left panel shows the training and testing errors, the right panel shows the confusion matrix

From the confusion matrix, we can see that the model has more difficulties to predict microstructures that were obtained at higher temperatures: 136 examples out of 522 examples for microstructures above the Curie temperature were wrongly classified. Note that due to the random split into training and testing data, there is a small class imbalance, which is why we should not directly compare the number of misclassified images and instead should use relative numbers: for the “high temperature” data, 25% of the images were misclassified, while for the “low temperature” data only 15.6% were misclassified.

To analyze if there specific temperature values for which microstructures are more prone to misclassifications than others, we could turn the two-class classification problem into a multi-class classification problem, e.g., using 16 classes of temperature ranges. Obviously, one could even increase the number of classes such that in the limit case, we use the classifier for solving a regression problem. Alternatively, we can also directly turn this into a regression problem. The network architecture requires then that the output layer contains only a single unit. Additionally, the labels of the data records are numeric labels containing the temperature. Figure 18.10 shows the results after training.

Ideally, all data points should be located on the dark, solid line. To quantify how well the data fits to this line, we calculate the *coefficient of determination* R^2 and the *correlation coefficient* corr , compare Chap. 9. We find $R^2 \approx 0.28$ where a perfect fit would be denoted by 1.0. Furthermore, it is $\text{corr} \approx 0.64$ which according to Sect. 8.7.3 denotes a positive correlation that is in between moderate and strong. It is interesting to observe that the microstructure examples that correspond to the Curie temperature are those that can be easiest predicted. Both implementations show good results from which it is obvious that the network learned something about the microstructures. Using more advanced network architectures, some of which will be introduced in the next chapter, however, can drastically increase the predictive capabilities. A systematic study where the performance of a range of different models is investigated can be found in NGUYEN et al. [5].

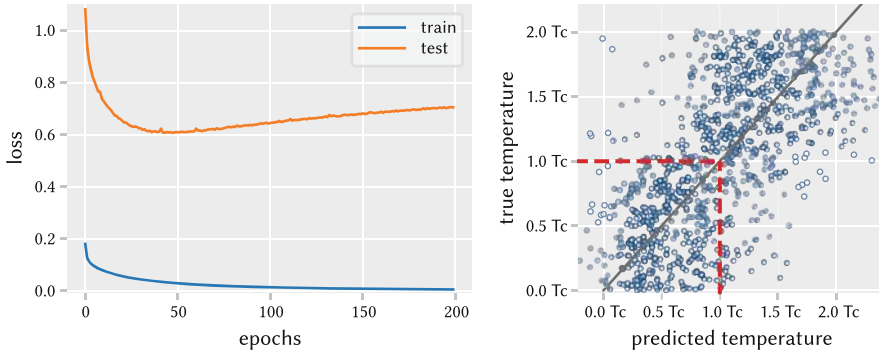


Fig. 18.10 The Ising dataset as a regression problem. The left panel shows the training and testing errors, the right panel shows the predicted vs the true temperature for the testing dataset

18.9 Exercises

18.1. In Sect. 18.3.3, we implemented a first model that performs backpropagation. In this exercise, you should implement the tracking of the mean testing loss and reproduce the evolution of the train/test losses from Fig. 18.3. Hint: in the `for` loop over all epochs, use the current weights and bias values to predict the labels for *training* data. Use the previously introduced function for splitting the dataset into training and testing data.

18.2. The derivation and Python implementation from Sect. 18.3.3 uses three biases.

- Is the third bias really needed or can this be “taken over” by the hidden units? Do an experiment with the code and discuss the results.
- How important is the type of initialization of the weights and biases? Perform an experiment with the Python code.

18.3. Use the code of the fully connected network to create a network architecture with which the XOR problem can be solved. The XOR function takes two input values, each of which is either 0 or 1. If both inputs have the same value, then the XOR function yields zero; if the inputs have different values (0 and 1, or 1 and 0), the output is 1.

Which is a “minimal network architecture”, e.g., in terms of number of layers and number of nodes, that still gives the desired result?

18.4. Compare the sigmoidal activation functions with the tanh and the ReLU activation function. Use the code for a fully connected network from the companion website, pick a dataset of your choice, and compare their performance. Also, measure the compute time required for evaluating the activation functions for a

single number and for an array of 100,000 elements. Which activation function is preferable?

18.5. Extend the code for a fully connected network used in Sect. 18.5.2 and perform some experiments.

- (a) Extend the code such that during the training also the test cost (MSE) is computed and recorded.

Hint: In the method `train` of the class `Network` add your code after back-propagation of the errors. The training cost is then computed with the *current* values of the weights and biases, based on the *testing* dataset.

- (b) What do you observe for the training and testing MSE when you increase the number of epochs or/and the learning rate? Discuss the extreme parameter combinations.
- (c) Can you manage to bring the model to a point where it overfits?

18.6. Use the code of the fully connected network from Sect. 18.5.2 to investigate the influence of different activation functions. In particular, compare the performance and discuss the behavior of the (i) sigmoidal, (ii) tanh, (iii) ReLU, and (iv) a leaky ReLU function. Which one is the best performing function for the investigated problem? For (iv) try different leakage values.

18.7. Extend the code for a fully connected network used in Sect. 18.5.2 such that the training is done using mini-batches. Compare this implementation with the one without mini-batches.

- (a) How do they perform in terms of stability or efficiency?
- (b) What is a good batch-size and what is the difference between a too large or too small mini-batch size?

18.8. Use the code from Sect. 18.5.2 and the extended version from Exercise 18.6 and answer the following questions:

- (a) What is the influence of the learning rate on the performances of the network concerning the training/testing data?
- (b) What is the impact of the number of epochs on the performances? (Hint: Investigate the training in terms of overfitting)
- (c) Does increasing the size of the network (i.e., the number of layers and the number of nodes per layers) lead to better performances? Discuss it with an example

18.9. In Sect. 18.6.3, we saw how weights can or should be initialized. Try different initialization methods (uniform, normal, constant, Xavier, He) and discuss the results and performances.

18.10. We saw that a neural network can solve the XOR problem. Can a neural network solve any combination of logical gate? Try this out for a number of combinations of different logical (Boolean) elements. What would be the condition to ensure that there exists a solution?

18.11. Investigate the “most confused” images further. Identify from the confusion matrix those images that get most often confused with a different digit. Locate these images and try to understand why they are misclassified the way how they are.

Bonus task: perform a principal component analysis on the same dataset as used here and interpret your findings in view of these results.

18.12. Use the code of the fully connected network from Sect. 18.5.2, and train it on the dataset DS-2-small (UCI Handwritten Digits). How do the predictions change if a new image is rotated by multiples of 90° ? Translational invariance is more difficult to be investigated since the image are very small and most of the digits fill the whole image. You can still investigate this for one of the more narrow digits, e.g., for a “1”. What do you observe?

References

1. X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. W. Teh and M. Titterton, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL <https://proceedings.mlr.press/v9/glorot10a.html>.
2. K. He, X. Zhang, S. Ren and J. Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE International Conference on Computer Vision (ICCV)*, Santiago, Chile, 2015, pp. 1026–1034, 2015. DOI <https://doi.org/10.1109/ICCV.2015.123>.
3. C. Hill. *Learning Scientific Programming with Python*. Cambridge University Press, 2nd edition. ISBN 978-1108745918.
4. S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.
5. B. D. Nguyen, P. Potapenko, A. Dermici, K. Govind, and S. Sandfeld. Forward modeling of materials science simulations: learning the microstructure-property relation. *under review*.
6. F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. DOI <https://doi.org/10.1037/h0042519>.
7. F. Rosenblatt. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms, 1962.
8. D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature* 323:533–536, 1986. DOI <https://doi.org/10.1038/323533a0>
9. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL <http://jmlr.org/papers/v15/srivastava14a.html>.
10. P. J. Werbos. Beyond regression—new tools for prediction and analysis in the behavioral sciences, 1974.
11. P. J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley-Interscience, USA, 1994. ISBN 0471598976.
12. B. Yuen, M. T. Hoang, X. Dong, and T. Lu. Universal activation function for machine learning. *Scientific Reports*, 11(1), Sept. 2021. DOI <https://doi.org/10.1038/s41598-021-96723-8>.



Advanced Deep Learning Architectures and Techniques 19

Any sufficiently advanced technology is indistinguishable from magic.

— Arthur C. Clarke (1917–2008),

English Science Fiction Writer

19.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become one of the work horses of microscopy image analysis in materials science and engineering. In the following subsections we will go through all required mathematical and conceptual ingredients.

19.1.1 Motivation of the Architecture

If you just finished the previous chapter, then you might get the idea that Artificial Neural Networks (ANNs) have made a great leap forward and that many more complex application cases are within reach. While this holds true for some classification and regression tasks, which otherwise would have been tackled by a combination of feature engineering and classical *machine learning* (ML), unfortunately, this is only partially true, when it comes to general image data. The back-of-the-envelope calculation in Example 19.1 demonstrates this.

Example 19.1 (The number of Weights in a Fully Connected Network)✂

Imagine the example images from dataset “DS-2-light (Alpaydin Digits).” They have $8 \times 8 = 64$ pixels. Now, imagine a fully connected *artificial neural network* (ANN) consisting of 64 input nodes, a fully connected layer with 80 input and 40 output nodes, and another one with 40 inputs and 10 outputs. This amounts to $64 \times 80 + 80 \times 40 + 40 \times 10 = 8,720$ weights.

Let us continue with the dataset “MDS-2 light (Ising Model, small)” where the images have $16 \times 16 = 256$ pixels and where for the binary classification task the first layer has 128 outputs, the second has 64, a third has 32, and the fourth output layer has 2. This amounts to $256 \times 128 + 128 \times 64 + 64 \times 32 + 32 \times 2 = 43,072$ weights.

Last but not least, imagine a small microscopy image on which we wish to perform a binary classification task. The image has $1024 \times 1024 = 1,048,576$ pixels. For an ANN with three hidden layers, each of which has 512 units, we get a total number of $1,048,576 \times 512 + 512 \times 512 + 512 \times 2 = 537,134,080$ weights, which, if using double precision floating point numbers, would already require about 4 GB of computer memory. Using RGB images with three channels would be another factor of three. Additionally, an ANN with three such layers would not be suitable because having an input size of about a million nodes and an output size of only 512 is far from being optimal for a network (see Exercise 19.1).

The memory requirement and therefore also the computational performance of fully connected networks scales so badly that alternative solution approaches are required. Additionally, ANNs are not translation invariant, and the spatial structure of an image is lost when the two-dimensional image array is “flattened” to a one-dimensional vector. This is a major problem as, typically, nearby pixels are correlated. Hence, in fully connected ANNs, this information is lost.

CNNs are motivated by what was done in image analysis with filter functions. There, the most important mathematical operation is the *convolution* of the image with a “kernel” function. Additionally, CNNs also come with a number of new techniques and layer types, all of which makes the CNN extremely efficient and effective.

19.1.2 Convolutional Layer

The *convolutional layer* is a key ingredient for a CNN. Such a layer is based on the mathematical concept of a convolution of two functions where one is “sliding” over the other (which is called *folding*), similar to what happens when we calculate a moving average.

Definition 19.1 (Convolution of Two Functions) Assume two functions, $f(x)$ and $g(x)$. Then, the convolution of f and g is defined by the integral

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(t)g(x - t) dt . \quad (19.1)$$

The asterisk between f and g represents the convolution operation. The convolution of two functions is a new function, $(f * g)(x)$.

This definition can easily be transferred to the two-dimensional case.

Discrete Convolution

For convolutional layer, however, the two-dimensional image consists of pixel, and therefore, we require a discrete convolution operation. In two dimensions and for an input (or image) array $\mathbf{I}$ and a discrete kernel $\mathbf{K}$, this is obtained for pixel number m, n as:

$$\mathbf{O} = \mathbf{I} * \mathbf{K} \quad (19.2)$$

$$\Leftrightarrow \mathbf{O}[m, n] = \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} \mathbf{I}[m + i, n + j] \cdot \mathbf{K}[i, j] , \quad (19.3)$$

where $\mathbf{O}[m, n]$ is the value in the resulting output array. The limits in the summation are for a kernel that is infinite in all directions. Usually, the kernel has a limited size such that the limits need to be adjusted, accordingly. An example for a discrete convolution is shown in Fig. 19.1. The discrete kernel has 3×3 elements, and the above summations would be for all elements j and $i \in [-1, 0, +1]$. To obtain all values $\mathbf{I} * \mathbf{K}$ in Fig. 19.1, the kernel w.r.t. $\mathbf{I}$ is moved to all possible locations, and, e.g., by moving the kernel to the leftmost and topmost position in $\mathbf{I}$, performing the respective multiplication and summation, the upper left element of $\mathbf{O}$ is obtained.

Such convolution operations are used in image processing, e.g., to detect edges, or to smooth an image; whether it is the former or the latter depends on the details of the kernel. In classical image processing a kernel often has odd numbers of elements along each direction. In *deep learning* (DL), however, this is different, and a kernel also can have even numbers of elements along each direction. For example, the operation shown in Fig. 19.1 can also easily be performed using a 4×4 kernel.

For the data structure that is used in CNNs, several dimensions have to be considered, because an image (unless it is a grayscale image) consists of several channels. For example, an RGB image has one channel for the red components, one for the green, and another channel for the blue components, resulting in the composite color image. Figure 19.2 shows convolution operations that result in a

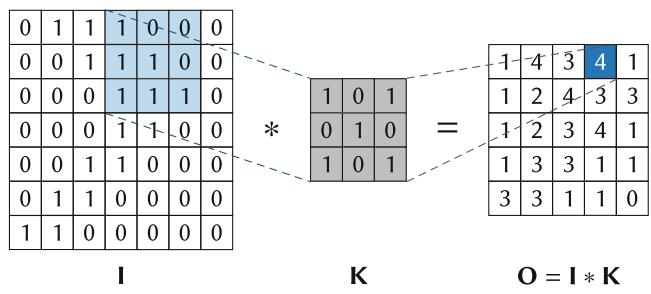


Fig. 19.1 A discrete convolution operation between a two-dimensional matrix **I** and a discrete kernel **K**. The mathematical convolution operation is indicated by a star (*). Each element of the two shaded 3×3 matrices is multiplied, and all nine products are summed up, giving the highlighted value in the right resultant matrix

Fig. 19.2 Convolution of an image with 32×32 pixels and 3 channels with a $5 \times 5 \times 3$ kernel which results in a single “slice”

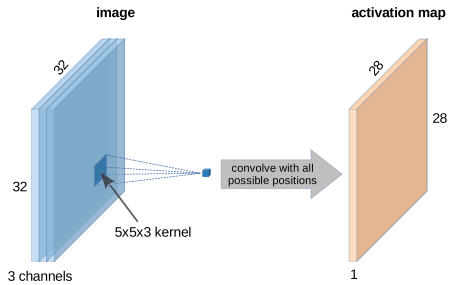
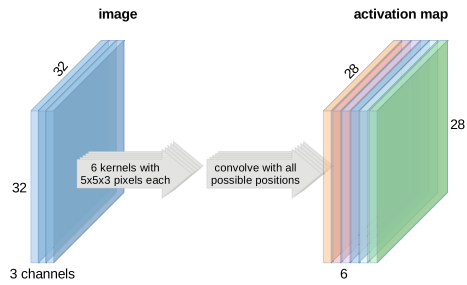


Fig. 19.3 Convolution of an image with 32×32 pixels and 3 channels with a total of six kernel functions that have $5 \times 5 \times 3$ pixels in size, resulting in six different activation maps



$M \times M \times 1$ layer. This requires a kernel function that has the same depth as the image.

Figure 19.3 shows a different type of convolution operation that operates with altogether six kernels, each of which reduces the image to a single layer and thus results in a $M \times M \times 6$ layer.

Additionally, a dataset has many records. Thus, we have to operate with four-dimensional arrays which, in the DL community, are often denoted as “tensors.”

For convolutional layers, there are two parameters that describe further details of the convolution process: the padding and the stride, as introduced below. An overview of “convolution arithmetic for deep learning” can be found in [8].



Fig. 19.4 Ninety-six convolutional kernels learned by the first convolutional layer of the AlexNet [23]. (With permission of the authors A. KRIZHEVSKY, I. SUTSKEVER and G. E. HINTON)

Learning

In convolutional layers, the weights are the discrete kernels; training is tantamount with finding the values of the kernel such that the convolution process is able to “filter out” the relevant features. Often, the kernels at the very beginning of a network are responsible for the most generic features such as edges or dark spots (also called the *low-level features*), while the further one gets to the end of the network, the more concrete but also complex the kernels are getting (*high-level features*), e.g., whole faces or complex compositions of different colors. An example of a visualization of the trained kernels of the AlexNet (which we discuss further below in Sect. 19.1.6) is shown in Fig. 19.4

Padding

Counting the number of elements of the matrices $\mathbf{I}$ and $\mathbf{O}$ we observe that $\mathbf{O}$ is smaller than $\mathbf{I}$, exactly by the width or height of the kernel (minus the kernel center). In case that there are a number of convolutional layers, the image shrinks after each of these layers. This can be a desired effect—but most of the time this is not the case. As another side effect, this also means that the corner elements of the input $\mathbf{I}$ are only represented once in $\mathbf{O}$ while all inner elements are represented as many times as there are elements in $\mathbf{K}$.

Padding remedies this by adding an extra “frame” around $\mathbf{I}$. The padding can either be initialized to zero or it can be set to the value of the boundary elements (as in a constant extrapolation).

Example 19.2 gives an example about dimensions of the padding, input, and output image.

Example 19.2 (Padding)



If the original image $\mathbf{I}$ has 8×8 pixel, then by adding a padding of one element around it and using a 3×3 discrete kernel, we would get an output $\mathbf{O}$ that has the same dimensions as $\mathbf{I}$ after the convolution operation is performed.

In general, it is easy to see that to preserve the size of the input also as output, and a kernel of $2n + 1 \times 2n + 1$ is used, then the padding must be of width n , resulting in an input image that is $2n$ larger in each direction. This is also called *same padding*. If no padding is used, this is called *valid padding*.

Stride

So far, the convolution kernel is moved across the input by steps of one pixel. If every other (or more) pixel is skipped, then the output will get a smaller dimension. The number of pixels to be skipped is governed by the parameter called *stride*.

Things to Remember About Convolution Layer

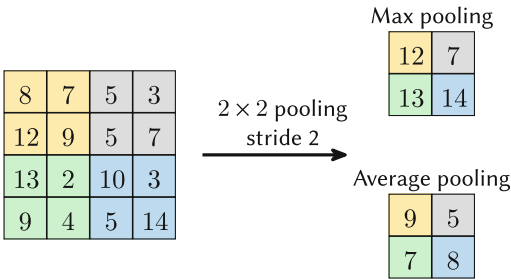


- Convolution operations are nonlinear transformations and are therefore capable of learning very complex aspects of an image.
- Padding is used such that during the convolution the output does not "shrink."
- If during shifting across the input image the convolution kernel should not cover pixel multiple times, *striding* is used to repeatedly skip some pixel.
- Stride effectively downsamples the input.

19.1.3 The Pooling Layer

The convolution can represent an arbitrarily complex transformation with an increasing number of weights. The computational performance, on the other hand, suffers from large numbers of to-be-trained weights. *Pooling* is an alternative to using stride regarding downsampling of the input. It summarizes the content of a number of neighboring pixel. Using only the maximum value is called *max pooling*, averaging the values is called *average pooling*. Figure 19.5 shows a sketch of pooling operations. Pooling typically is accompanied by a corresponding stride. Even though this is not a necessity, it is often encountered because pooling is used for reducing the number of weights. Max pooling has the advantage that it preserves fluctuations and maximal signals. During pooling, some information is lost, which can make it more difficult for the network to learn. If the benefits from the reduced number of weights outweigh the disadvantage of lost information, it is not always obvious and needs to be investigated for the respective problem at hand. Pooling layers used to be very beneficial for computational reasons and in times when the hard- and software were not so developed. However, today, their importance is somewhat reduced, and there are even approaches where exclusively the convolution layer is used.

Fig. 19.5 Sketch of max pooling and average pooling, used together with a stride value of two. The colored squares on the left correspond to the numbers on the same colors on the right



19.1.4 The Receptive Field and Dilated Convolutions

The receptive field of a CNN describes the size of the region of the input that influences a pixel in the output. For example, for object detection a receptive field that is much smaller than the object may not be able to properly find the object. ARAUJO et al. [1] “observe a logarithmic relationship between classification accuracy and receptive field size, which suggests that large receptive fields are necessary for high-level recognition tasks, but with diminishing rewards,” where the authors investigated network performance based on classification accuracy of ImageNet. The receptive field can be increased in an obvious manner by adding more convolutional layers, i.e., by increasing the network depth or by adding pooling layers or higher strides.

Another approach is to apply *dilated convolutions*. There, the convolution kernel is reduced to only every r th row and column where r is called *dilation rate*. All other kernel elements are effectively “dropped” and do not have to be considered for the calculation. A sketch of the dilated convolution for $r = 2$ is shown in Fig. 19.6. From the sketch it is obvious that a 3×3 kernel with a dilation rate of 2 has the same receptive field as a 5×5 kernel. We can compute the effective kernel size m_{eff} as

$$m_{\text{eff}} = (m - 1) \cdot r + 1 , \tag{19.4}$$

where m is the number of considered elements in the dilated kernel for each dimension. For example, for the situation in Fig. 19.6 it is $m = 3$, $r = 2$, and $m_{\text{eff}} = 5$.

Dilated Convolutional Layer

In DL networks a dilated convolutional layer is often used instead of the last standard convolutional layer.

?

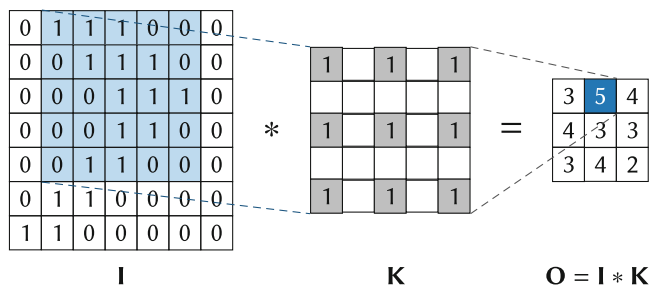


Fig. 19.6 Schematic of a dilated convolution. Even though the kernel has the same number of elements as the one shown in Fig. 19.1, the receptive field has increased

19.1.5 Feed Forward Step and Backpropagation

The feed forward step of a convolutional layer consists in performing a convolution operation, where additionally needs to be considered that images often consist of more than one channel (i.e., a color image consists of one pixel layer for each of the colors red, green, and blue). Furthermore, a convolutional layer typically contains not only a single convolution operation for each input channel but several, so that during training each filter kernel of a layer learns to identify a different characteristic of the image. This is what adds “depth”, i.e. a third dimension, to a convolutional layer.

The derivation of the mathematical formulation for backpropagation is based on the fact that the mathematical convolution operation consists of a summation of products between the kernel elements (the weights) and the input values of the layer. Other than that, the same steps as also required for a fully connected layer need to be performed.

Last but not least, the convolution operation is typically followed by an activation operation where an activation function is applied to each output value of the convolution. The formulations of the feed forward step and the backpropagation are again obtained in analogy to a fully connected network.

19.1.6 CNN Architectures

Subsequently, we introduce two early CNN architecture that had a significant influence on the development of CNN DL technology.

LeNet

One of the earliest convolutional neural networks was *LeNet* developed in 1989 by YANN LECUN et al. [24]. After several research iterations, in 1995 *LeNet-5* was developed, shortly after the MNIST dataset of handwritten digits (in this text used as dataset DS-2 [MNIST]) was created. A comparison with several other methods for

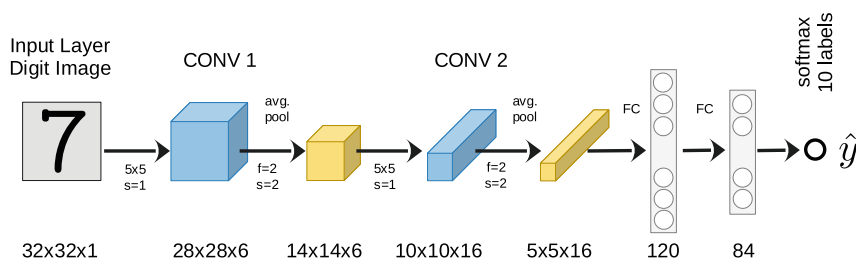


Fig. 19.7 Summary of the architecture of the LeNet-5 [24]

identifying the handwritten digits revealed that the CNNs architecture outperformed all other methods. The architecture is shown in Fig. 19.7. It consists of an input layer that takes a single-channel image of 32×32 pixel. This is followed by convolutional layers with a 5×5 kernel and 2-element padding. Then, average pooling was performed with a stride of 2, effectively reducing the 28×28 output from the convolution layer to half the size. This was followed by another convolution layer with the same kernel and padding, followed by an average pooling layer. Finally, there were two fully connected layers. Note that pooling requires no learnable weights, so weights need to be allocated only for the convolutional and the fully connected layers. As activation functions (after the convolution layer and the fully connected layers), a sigmoidal function was used.

The LeNet was a milestone for the development of CNNs, and even though today, the CNN architectures are quite different, all important elements were already present.

AlexNet

As opposed to the original LeNet, the AlexNet was designed to run on multiple GPU hardware. It is one of the early, deep CNNs. It consisted of eight layers: five convolutional layers, partially followed by max-pooling layers, and at the end completed by three fully connected layers; cf. Fig. 19.8. The “specialty” of this architecture was that it ran on two GPUs, each of which contained a copy of the whole network except for the last, fully connected layer. Each GPU then learned different kernels, some of which were already shown above in Fig. 19.4. AlexNet easily won the “ImageNet competition” in 2012. It is considered the most important impetus for subsequent advancements in DL for computer vision.

ResNet

The residual neural network (ResNet) is a more recent network and, in 2015, won the ImageNet¹ competition. One of the problems for deep network is the *problem of*

¹ ImageNet (<https://www.image-net.org/>) is a huge image database with millions of annotated images. Since 2010, the ImageNet regularly organizes contests about classification and detection of objects where networks compete against each other.

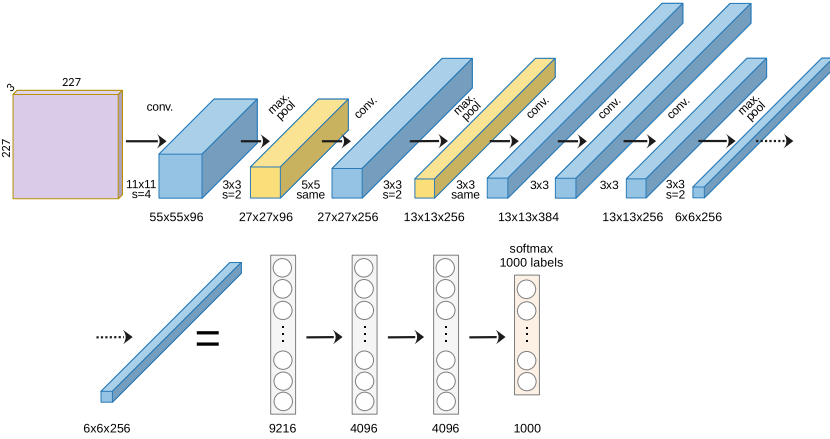


Fig. 19.8 The AlexNet [23]: summary of the architecture

vanishing gradients, where gradients are becoming smaller and smaller while being propagated through the many layers of a deep network until they nearly disappear, and the performance of the network saturates.

The solution, used by the ResNet architecture [13], is based on the fact that in a deep neural network, a significant part of the learning takes place in the first layers, while the learning increment from the layers closer to the output is much smaller, sometimes even approaching zero. Thus, these layers could, in principle, be replaced with the identity function. However, it was shown by K. HE et al. [13] that a better approach is to add a small increment to the identity mapping $\mathbf{1}(x) = x$ of the input x in the form

$$y = \mathcal{F}(x; \{w_i\}) + \mathbf{1}(x) , \quad (19.5)$$

where $\mathcal{F}$ is the “residual mapping” to be learned,

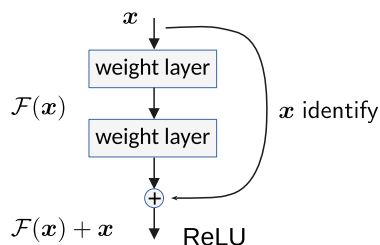
$$\mathcal{F}(x; \{w_i\}) = y - \mathbf{1}(x) . \quad (19.6)$$

from which we see why $\mathcal{F}$ has the function of a residual.

For this purpose the ResNet contains a new element, the *residual block*. This is a network part that, in the simplest version, consists of two connected 3×3 convolutional layers and a *residual connection*, also called *skip connection*; see Fig. 19.9.

These connections have the ability of initially “doing nothing,” i.e., the information bypasses the two convolutional layers and reuses the activations from the previous layer. This speeds up the training during the initial phase, as effectively a much shallower network is trained. Later on, the remaining, previously deactivated layers are also included in the training process.

Fig. 19.9 Residual block in a deep residual Network. The identity function skips two layers



In further developed ResNet-type networks one can also find *bottleneck blocks*, a variant of the residual block, consisting of three convolutional layers. Currently, there are architectures (the *HighwayNets*) which have the ability to decide how many layers to skip, adapted to the respective situation.

19.2 Deep Learning Techniques

In what follows, we briefly explain a few advanced DL techniques that help to increase the performance of networks.

19.2.1 Optimization Methods

In the previous parts of this text, we encountered a number of times the *gradient descent* (GD) methods which were used for minimizing a cost function. The *stochastic gradient descent* (SGD) is the counterpart of the GD method. In each iteration it is applied to a randomly chosen subset of the data. The benefit for high-dimensional minimization problems is that the computational cost is reduced, which speeds up the minimization. However, this often comes at the price of slower convergence. Note that it is not obvious that performing the minimization on a subset of the dataset works (even though it does).

If instead of randomly choosing subsets of data one uses mini-batches, then we are somewhere in between a GD and the SGD. This method is called *mini-batch gradient descent*. In DL this is usually the preferred algorithm.

The variants of the GD method are, by far, not the only methods for cost minimization and finding parameters. We only mention two of them and refer the reader to the extensive literature on numerical optimization. The first is the *Adam* optimization algorithm [20] which was designed for non-convex optimization tasks and which is able to operate with adaptive learning rates. Last but not least, there is a classical optimization algorithm, called *BFGS* (after the four persons Broyden, Fletcher, Goldfarb, and Shanno) which includes information of the curvature through the Hessian matrix.

19.2.2 Momentum

In cases where the minimizer encounters very shallow “valleys,” a technique called *momentum* is useful (in fact, it is useful in many situations). While GD (or SGD) uses only the current gradient, in this method, the gradients are “accumulated” over a few epochs. A typical parameter that needs to be fixed is a coefficient that scales the strength of the summed gradients, the *coefficient of momentum*. With this, if the optimizer just reaches a valley, then there is still “enough gradient left” from before, to do larger update steps.

19.2.3 Early Stopping

In practical DL frameworks, one often stumbles across the notion of *early stopping*. The purpose is to automatically detect the point during the training when any further training would deteriorate the ability to generalize. A number of different methods exist, some of them can be analytically derived, some are rather heuristic methods, and others are highly tailored to the minimizer or to the network under consideration.

19.2.4 Transfer Learning

Transfer learning is a technique often used in DL where a model is first trained on one dataset of particular objects. The trained model is then used for further training or for inference on data of different objects. For example, a network that was trained on peas (e.g., for counting them) probably is not too different from a network trained on lentils.

19.2.5 Data Augmentation

Last but not least, an important technique to increase the number of training images are image transformations, such as flipping, rotating, cropping, (color) jittering, warping, adding image “defects” or noise, etc. Even though these are not entirely new images, usually this technique is already sufficient to significantly increase the accuracy of a model. Data augmentation can also be understood as an oversampling technique (cf. Sect. 8.4.4) as the amount of data is increased by “reusing” existing images.

Care must be taken that none of the used augmentation transformations violates the physical reality. For example, if the straight lines in an image should always be parallel, then warping the training images might deteriorate the prediction accuracy.

19.3 Two Examples for Deep Learning in Microscopy

19.3.1 Semantic Segmentation of TEM Images of Nanoparticles

The first example uses a dataset that has been published along with the journal publication by SYTWU et al. in [37]. The data is obtained from an *transmission electron microscopy* (TEM) experiment with gold nanoparticles of 2.2 nm diameter. The resolution of the TEM was such that the crystalline structure of the nanoparticle can be distinguished from the amorphous background. For more information please refer to the original publication [37]. The dataset also contains a label, i.e., a domain expert annotated the images, indicating which part of the image is a nanoparticle and which part is not. Finding such disjunct regions is called *semantic segmentation*.

For training, we used the U-Net architecture by RONNEBERGER et al. [33]. The U-Net is a CNN which consists of a contracting path and an expansive path, similar to an encoder/decoder structure of an autoencoder architecture (cf. Sect. 19.4). The contracting path consists of a sequence of convolutional layers together with max pooling downsampling operation, resulting in reduced spatial resolution of the layers where at the same time the represented feature complexity can be very high. The expansive path projects the learned lower-dimensional features back into the pixel space. These two steps give the network the u-shaped architecture. U-Nets have been proven to be very efficient and accurate for semantic segmentation, but can also be used for other types of problems. Figure 19.10 shows examples of the images and the corresponding predictions.

One can see that for some of the sections in the images it is even for us very difficult to see differences and to exactly pinpoint where the crystalline regions

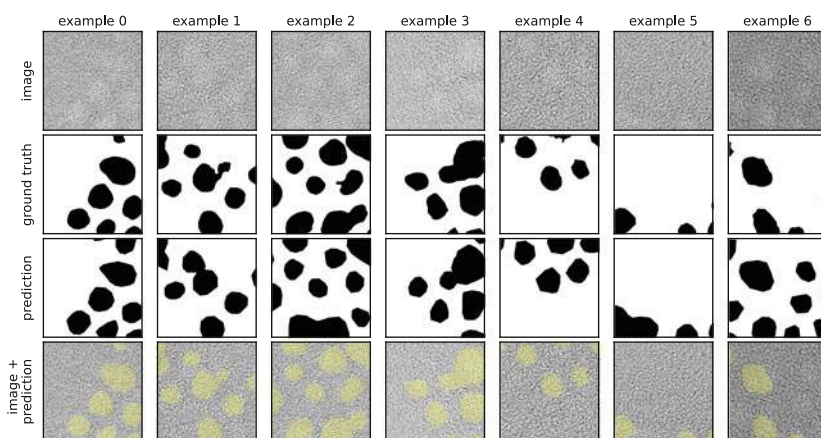


Fig. 19.10 Semantic segmentation of crystalline Au nanoparticles, embedded into an amorphous matrix. The DL task is to identify the crystalline regions. The ground truth was prepared by a domain expert. The superposition (bottom row) shows both the microscopy image and the predicted masks

end and the amorphous phase begins. In example 2, our model did not separate the two bottom nanoparticles correctly. In example 6 the prediction of the DL model looks more accurate. Given that we only used an “off-the-shelf” U-Net, initialized the weights from the ImageNet, and performed basic data augmentation (such as rotation, warping, jittering, etc.), the results are very satisfying. The performance of the U-Net seems to be at least comparable to human performance. Given how much time it takes to manually annotate such images, it is clear that such DL approaches are extremely useful tools for image analysis in microscopy. The analysis does not have to stop here, though. With the information about the nanoparticle, one could now start to analyze their shape or to perform statistical analyses and thereby turn these images possible even into knowledge (cf. Sect. 2.3). The next example shows some possible solution strategies if labeled images are not available.

19.3.2 Binary Segmentation Using Synthetic Training Data for Grain Microstructures

This example is a typical materials science and microscopy application case, where a convolutional neural network is used for binary segmentation of microscopy images. The study is the own work of—among others—the author of this text and has been previously published in [38] under the terms and conditions of the Creative Commons Attribution (CC BY) license.²

Binary segmentation of microscopy images is one of the tasks where CNN-based DL methods already became a very useful tool. Binary segmentation typically involves finding contours or edges of features, e.g., of the grain boundaries in grain microstructures. The top panel of Fig. 19.11 shows typical grayscale images with grain distributions, obtained from *scanning electron microscopy* (SEM) microscopy.

A main problem of ML and DL approaches is that any model requires data for training. As a rule of thumb one can say, the higher the degree of complexity of the features contained in the images to be analyzed, the more training data is needed. This makes microscopy images rather difficult to learn, but on the other hand, the effort for *hand-labeling* (i.e., manually identifying and “drawing” the contours) is also significant.

Therefore, in this study, another approach was chosen. Synthetic grain microstructures were created using a variant of the Voronoi tessellation. Then, a range of different types of “randomness” was added to increase the variability of the dataset. From the Voronoi tessellation, we also could directly obtain the “masks,” i.e., the black lines that indicate the objects of interest. Since this was generated by a Python script, it was easily possible to create thousands of images with the corresponding masks for training. The question was, how far a CNN (see³

² <https://creativecommons.org/licenses/by/4.0/>

³ In this study a ResNet-50-based U-Net was used where the original U-Net architecture was adapted, such that standard convolutional layers were replaced by residual blocks. Furthermore,

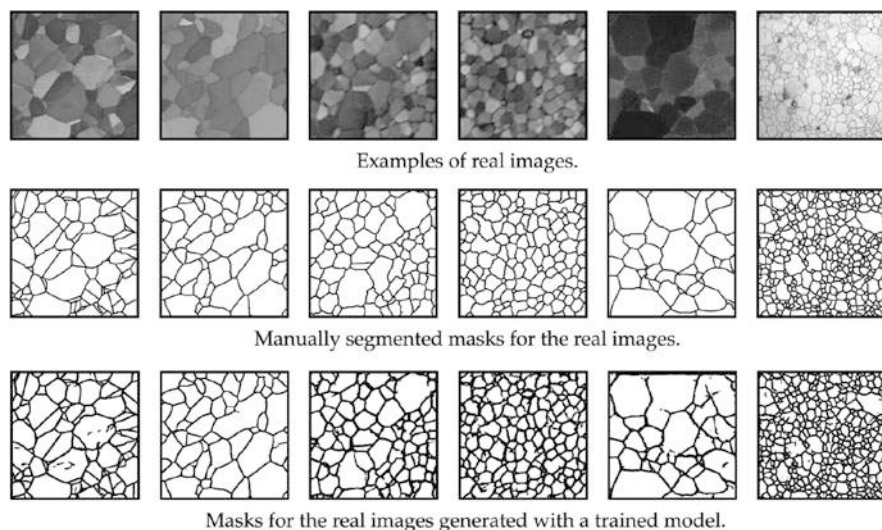


Fig. 19.11 Images of grain microstructures, obtained from scanning electron microscopy (top row). The middle row shows the masks, obtained from manually “labeling” the grain boundaries, while the bottom row shows the DL predictions using a network that was trained on synthetic images. Previously published in [38] under the terms and conditions of the Creative Commons Attribution (CC BY) license

for further details), which was only trained on synthetic images, can be used for predictions on real microscopy images. There, an additional technique was used: *adaptive data generation* and *fine-tuning*. We used the trained model after each training cycle to find the images with the worst predictions. Since the generation procedure of the synthetic images was fully parameterized, we could then create small variations of these images. This effectively increased the available data of images with particular features.

Taking a look at the predictions in Fig. 19.12 for the synthetic images shows nearly perfect results. This may be not entirely surprising since these images have significantly less noise, randomness, and artifacts than real microscopy images.

Therefore, the real challenge is then to make predictions for the real microscopy images (Fig. 19.11 [bottom row]). There, we observe that the accuracy of the binary segmentation is very high. This was particularly surprising, as the microscopy images came from very different sources and materials. Some small sections of grain boundaries were missing; however, with more specialization for a particular microscopy experiment, even this could be mended. However, in this study, the goal was to develop a very generic approach and to not use anything but synthetic training data and a convolutional neural network.

a ResNet50 was used as backbone of the hourglass-like U-Net architecture for the segmentation process.

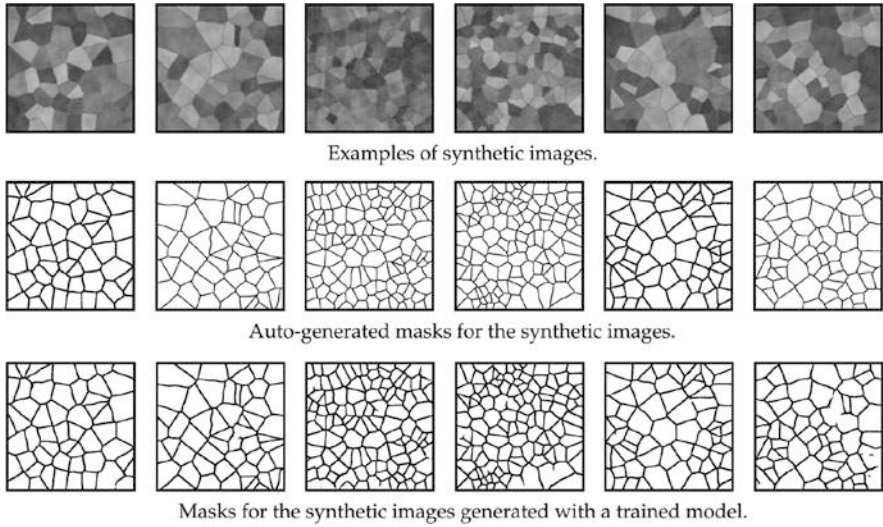


Fig. 19.12 Synthetically generated grain microstructures (top row). The middle row shows the masks, also automatically obtained, while the bottom row shows the DL predictions. Previously published in [38] under the terms and conditions of the Creative Commons Attribution (CC BY) license

19.4 Autoencoder: How to Learn with Networks Without Supervision

An autoencoder is a specific type of a feedforward neural network, where for each input node there exists a corresponding output node. An autoencoder is a neural network that learns in an unsupervised manner two different functions. The first is the *encoder* which takes the input data and turns it into an encoded (i.e., compressed) representation, which is then contained in the *code* (also called *latent space*); see the left panel of Fig. 19.13. The other function, which is learned, is the *decoder* which reconstructs the input data again. One of the applications of autoencoders is, similar to *principal component analysis* (PCA) from Sect. 15.2, the *dimensionality reduction* of a given dataset. Similarly, it can also be used as a generative model which, after training, is able to generate new random data that is similar to the training data. The first version of an autoencoder is attributed to MARK A. KRAMER [22] who introduced it as a nonlinear kind of a PCA method.

19.4.1 Single-Layer and Multilayer Architectures

In the simplest case, an autoencoder consists of only one hidden layer, as shown in the sketch in the right panel of Fig. 19.13. However, one has to avoid that the autoencoder just duplicates the training data. This is why typically, the code

Fig. 19.13 The autoencoder architecture consists of an encoder and a decoder (left panel). The low “code” is in the intersection of the two. The right panel shows a simple one-layer autoencoder network

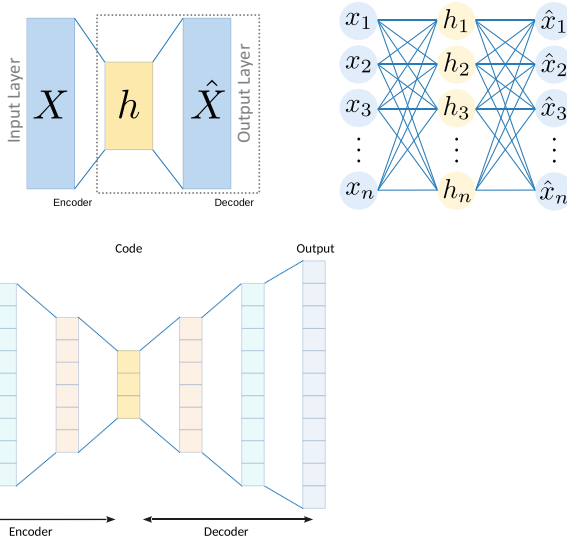


Fig. 19.14 Visualization of a multilayer autoencoder: during the forward pass the input information moves through the encoder, which here is a fully connected ANN with multiple hidden layers, to produce the code. The decoder consists of a similar but “mirrored” layer structure and produces output only using the code as input

represents a lower-dimensional space, i.e., the corresponding single-code layer has a lower number of units than the number of inputs. In more complex situations, e.g., in case of complex features or large differences between the input dimension and the code dimension, multilayer autoencoder architectures are required. Figure 19.14 shows an architecture with three hidden layers for both the encoder and decoder in a compact visualization. The code always consists of only one layer; its exact size is a hyperparameter.

The training of all autoencoders relies on a cost function that characterizes how different the input from the output is. Typical loss functions (remember, “loss” in the DL community always means “mean loss,” i.e., the cost) are in general the *mean squared error* (MSE) and in particular for image data the binary cross-entropy function. Then a minimization algorithm then tries to find the weights that minimize the cost function. With this, predictions can be made for *any* input.

19.4.2 The Relation to Principal Component Analysis

There is a surprising similarity between autoencoder and principal component analysis: both store the encoded information as a lower-dimensional representation, the “code,” and both are unsupervised learners that can be used for similar tasks. The main difference is the way how the data is compressed: PCA is a linear method, i.e., the relation between the original data, the code, and the reconstructed data is given

by linear transformations. Autoencoder, on the other hand, usually uses nonlinear transformation because the activation functions are nonlinear.

In the case that only linear activation functions are used together with the same cost function as for [PCA](#)⁴ then the result will be comparable—however, note that [PCA](#) enforced that the subspace of the code is constructed such that all basis vectors are orthogonal. This is not the case for autoencoders.

19.4.3 Convolutional Autoencoder

In particular for image data, fully connected architectures quickly become inefficient and *deep CNN autoencoders* are preferred. The encoders consist of a number of 2D convolutional and max-pooling layers, while the decoder consists of 2D transposed convolutional layers for upsampling. One of the typical applications in materials science is *denoising* and image reconstruction. Furthermore, autoencoders are also used for anomaly detection.

19.5 Generative Adversarial Networks: How to Create Data?

A branch of generative models that have become more and more popular are *generative adversarial networks* ([GANs](#)), first introduced in 2014 by IAN GOODFELLOW et al. [[11](#)]. In the following, we will introduce the most important aspects of [GANs](#) based on an image dataset. However, everything that is said can be generalized to arbitrary data as well.

19.5.1 Model

Generative implies that the model is able to produce new data, while *adversarial* denotes the fact that the model will compete against another one. The two models are termed *generator* and *discriminator*.

The task of the generator is to produce plausible data that should represent the main aspects of the training dataset. The goal of the discriminator is then to distinguish between the real and the generated data. In the most simple form, this is just a classifier.

⁴ There is an alternative derivation to the one shown in Sect. [15.2](#) which is based on the [MSE](#) and which yields the same results.

so-called *min-max GAN loss* which is obtained from the *binary cross-entropy loss*

$$\mathcal{L}(\mathbf{y}^{\text{real}}, \mathbf{y}^{\text{pred}}) = - \left(\mathbf{y}^{\text{real}} \cdot \ln(\mathbf{y}^{\text{pred}}) + (1 - \mathbf{y}^{\text{real}}) \cdot \ln(1 - \mathbf{y}^{\text{pred}}) \right). \quad (19.7)$$

Here, $\mathbf{y}^{\text{real}}$ and $\mathbf{y}^{\text{pred}}$ is one image (i.e., one data record). The generalization to several data records will be done after the resulting loss has been derived for a single record.

The Min-Max Game



The word “game” in the title of this subsection is reminiscent of the origin of the below introduced algorithm, which stems from the field of game theory. There, it was originally formulated for a “two-player zero-sum game,” where the player either makes simultaneous moves or alternately takes their moves. One player tried to maximize a score while the other tries to minimize it.

Discriminator Loss

There, $\mathbf{y}^{\text{pred}}$ is the prediction obtained from the discriminator (for real or fake data), and $\mathbf{y}^{\text{real}}$ is the correct label. As the discriminator is a classifier, we use a value of 1 as being a real image and 0 as being fake. We can then simplify the equation and get the loss function for the discriminator as:

$$\mathcal{L}_{\text{disc}}(\mathbf{y}^{\text{pred,real}}, \mathbf{y}^{\text{pred,fake}}) = \mathcal{L}(1, \mathbf{y}^{\text{pred,real}}) + \mathcal{L}(0, \mathbf{y}^{\text{pred,fake}}) \quad (19.8)$$

$$= - \left(\ln(\mathbf{y}^{\text{pred,real}}) + \ln(1 - \mathbf{y}^{\text{pred,fake}}) \right) \quad (19.9)$$

There, we used the notations from Algorithm 19.1 with $\mathbf{y}^{\text{pred,real}} = D(\mathbf{x})$ and $\mathbf{y}^{\text{pred,fake}} = D(G(\mathbf{z}))$. The goal of the discriminator is to minimize its loss:

$$\mathcal{L}_{\text{disc}}(\mathbf{y}^{\text{real}}, \mathbf{y}^{\text{pred}}) \rightarrow \text{MIN}, \quad (19.10)$$

which is identical to maximizing the sum of the two $\ln(\cdot)$ terms from Eq. 19.9:

$$\ln(\mathbf{y}^{\text{pred,real}}) + \ln(1 - \mathbf{y}^{\text{pred,fake}}) \rightarrow \text{MAX}. \quad (19.11)$$

Generator Loss

When the generator is trained, the discriminator is kept fixed. The only possible input to the discriminator is therefore a fake image. As a consequence, the first term in the equations of the discriminator loss becomes zero. The generator is supposed to generate fake images $G(\mathbf{z})$ that cannot be differentiated from real images.

Consequently, the discriminator must predict it as real with label 1. Effectively, the discriminator has the goal of obtaining a label of 1 for $D(G(z))$, while the generator attempts to obtain a label of 0. The loss for the generator is again obtained from the cross-entropy formulation:

$$\mathcal{L}_{\text{gen}}(\mathbf{y}^{\text{pred, fake}}) = \ln(1 - \mathbf{y}^{\text{pred, fake}}) , \quad (19.12)$$

which the generator tries to minimize during the training, resulting in:

$$\ln(1 - \mathbf{y}^{\text{pred, fake}}) \rightarrow \text{MIN} . \quad (19.13)$$

Resulting Loss for the GAN

Both networks have to achieve their optimization goals: the generator seeks to minimize $D(G(z))$ (Eq. 19.13), while the discriminator attempts to maximize $D(G(z))$ (Eq. 19.11). This is why this section is called “Loss Functions for the Min-Max Game.” The Nash equilibrium will be reached if in the discriminator the probability for an image to be real or fake is 50%, which translates to getting a loss value that is close to 0.7.

This is the simplest loss function which, however, has the drawback that it can cause the generator to get stuck in the early stage of the training where the task for the discriminator is very easy. A more robust version of this loss is also proposed in the original publication [11]. That type of loss would be a non-saturating loss form:

$$\mathcal{L}_{\text{gen}}(\mathbf{y}^{\text{pred, fake}}) = \mathcal{L}(1, \mathbf{y}^{\text{pred, fake}}) = -\ln(\mathbf{y}^{\text{pred, fake}}) . \quad (19.14)$$

Applying this strategy to multiple records at once and using $E[\cdot]$ for the expected value, we obtain the following formulation that will result in the training of the GAN network:

$$\text{MIN}_G \text{ MAX}_D [E[\ln D(\mathbf{X})] + E[\ln (1 - D(G(z)))]] . \quad (19.15)$$

Training Algorithm

For the whole training, we note that the generator and discriminator are not trained at the same time but sequentially. During training of the generator, the discriminator is frozen and vice versa. Then the training can be summarized in the following algorithm, adapted from [11]:

Algorithm 19.2: Algorithm for Training of a GAN

FOR number of epochs DO :

FOR k steps DO :

- sample minibatch of m noise samples $\{z_1, \dots, z_m\}$ from $\mathcal{N}$
- sample minibatch of m examples $\{x_1, \dots, x_m\}$ from data generating distribution
- update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\ln D(x_i) + \ln(1 - D(G(z_i)))] ,$$

where ∇_{θ_d} is the directional derivative.

END FOR

- Sample minibatch of m noise samples $\{z_1, \dots, z_m\}$ from $\mathcal{N}$
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m [\ln(1 - D(G(z_i)))] .$$

19.5.4 Common Problems of GANs

GANs are less robust than, e.g., CNNs and are more difficult to train. Intuitively, this is clear, as it is easier to predict the temperature of one of the images of the MDS-2 (Ising model) than to generate a new image according to a given temperature. Due to the complex architecture, there are a number of additional points where the training can fail, two of which will be briefly covered here.

Mode Collapse

Probably the most important one is *mode collapse*. This may occur when a generator creates a fake image (or any other data structure) that is particularly convincing. This can easily happen since at the early stage of the training, only a few modes are covered. The discriminator then will concentrate on only the features that occur in the limited set and can quickly overfit. Then, the generator may learn to produce only this particular output without any variance. This scenario can happen whenever the generator is learning much faster than the discriminator. While a complete collapse is not too common but a partial collapse happens often. Mode collapse is a problem for which remedies are still being investigated.

Vanishing Gradients

We encountered the *problem of vanishing gradients* already before; however, in the context of GANs this is a slightly different kind of a problem. Here, the problem can occur when the discriminator is learning much faster than the generator. Then, the discriminator is perfectly able to differentiate between fake and real data, and the generator would always face a very high loss, resulting in a very shallow “loss landscape,” which is a very difficult optimization problem.

19.5.5 Alternative Architectures and Variations

A number of variations exists, and the original architecture is rarely used today. One of the closest to the original architecture is the *conditional GAN* (CGAN) which is only a slight variation of the standard GAN [11]. The main idea is to provide the class label also to the generator such that there is a “side condition” for the generator (the respective class), which restricts the space of possibilities of the generator.

In 2017, an alternative of CGAN introduced as auxiliary classifier GAN (AC-GAN) was published in [29]. Here, instead of simply providing the label to the networks, an auxiliary network is also used to classify the real and the generated images. A new loss term is then used to ensure that the generated images correspond to the expected class. Additionally, the developer of the AC-GAN also introduced a number of other technical improvements to achieve a better performance.

The Wasserstein GAN (WGAN), proposed in 2017 in [2], is a further development of the original GAN where the discriminator is not acting as classifier. Rather, for each new instance, the discriminator predicts a number $0 \dots 1$ which serves as a score. The Wasserstein GAN is a way of solving the problem of vanishing gradients, due to the used *Wasserstein distance* which provides a better gradient of the loss as compared to the regular GAN. However, it has also been shown that the loss does not become stable [28], which was a major drawback. A number of efforts, such as [12], were made to make the training more stable.

Other developments have been made to determine better loss functions (see, e.g., [5, 27]) or to stabilize the training by introducing regularization terms to the loss [28]. One of the key milestones in the developments of GAN was reached with the publication of pix2pix [14] and the CycleGAN [42]. This allows to use generative models to produce an image based on a given input, e.g., for generating colored photos from black and white images. This also might enable a number of exciting applications in materials science.

With the development of ProGAN [16], it has become possible to progressively train a GAN to produce high-resolution images with an outstanding quality based on low-resolution input. The StyleGAN [18] is based on this approach; the current state of research is given by [17]. A somewhat different direction is used by approaches

based on *transformers* [39] which open a door for new GAN development [9]. The other major improvement results from the combination of large language models (LLMs) such as CLIP [30] as a conditioning mechanism for generative models. By combining the VQGAN and the CLIP transformer, [7] allows prompt base-generated images—which might not seem to have a direct application in materials science; however, thinking along the line of metadata and ontologies, one also could see potential applications.

19.5.6 Python Implementation

We will now return to a minimalistic GAN architecture. Even for this, the algorithm of a GAN is, due to the different learning strategies, more complex than the one that we introduced for fully connected networks in Sect. 18.5 (we assume that the reader is already familiar with that). Even though we cannot show the whole code, we will still introduce the most important steps and hints that would be required for a full Python implementation, because we are convinced that writing code is very useful for an in-depth understanding. The full code, which in part is still quite similar to the one for the fully connected network, can be found in the supplementary material.

We start with the class that also contains the training (this was the class `Network` for a fully connected network), which is here called `GAN`. Here, we show the first two methods:

```
In [1]: # First part of the implementation
class Gan:
    def __init__(self, cost_function, derivative_of_cost):
        self.layers_gen = []
        self.layers_disc = []
        self.cost_function = cost_function
        self.derivative_of_cost = derivative_of_cost

    def add_layer(self, layer, model):
        if model == 'gen':
            self.layers_gen.append(layer)
        elif model == 'disc':
            self.layers_disc.append(layer)

    def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
        ...

    def generate(self, batch, z_dim): ...
```

Because for the GAN we need two types of layers, the layers for the generator and for the discriminator (`self.layers_gen` and `self.layers_disc`, respectively), the `__init__` method was slightly extended as compared to the `Network` class. This also reflects in the method for adding new layers where the parameter `model` indicates to which list of layers a new layer should be added.

The training method requires further adaption as compared to the original, fully connected version. We will now go step by step through the whole training method:

```
In [1]: # 2nd part of the implementation

def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
    all_costs = []
    for epoch in range(epochs): # or use trange from package tqdm
        np.random.shuffle(X_train)
        X_train_batch = np.array_split(X_train,
                                       int(len(X_train) / batch))
        for x_train in X_train_batch:
            # 1 - sample the noise vector from a normal distribution
            noise = np.random.normal(0, 1, (x_train.shape[0], z_dim))

            # 2 - generate 'fake' example by giving noise as input
            #       to the forward propagation of the generator
            fake = noise
            for layer in self.layers_gen:
                fake = layer.feed_forward(fake)
```

What is new in the signature of the above method is that we now have the size of the random vector, `z_dim`, as additional parameter. The parameter `decay` implements a decreasing learning rate η of the form:

$$\eta^{\text{new}} := \eta^{\text{old}} \cdot \frac{1}{1 + \text{decay} \cdot \text{epoch}}, \quad (19.16)$$

by which initially a larger learning rate is used than later during the training. However, note that the learning rate *in general* needs to be significantly smaller than that of a “non-GAN” network, as can be observed in the two examples, Sect. 19.5.7. The parameter `batch` defines the mini-batch size. For each epoch, the split into mini-batches is performed randomly anew. The first part inside the two nested `for` loops starts by sampling from a random normal distribution which is then used as input for the generator, propagated through all layers of the generator. At the end, a fake image is the result.

The next block of code (still in the training method) is responsible for obtaining predictions:

```
In [1]: # 3rd part of the implementation

def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
    ...
    ...
    # 3 - obtain prediction for the fake image
    fake_pred = fake
    for layer in self.layers_disc:
        fake_pred = layer.feed_forward(fake_pred, store=True)

    # 4 - obtain prediction for the real image
    real_pred = x_train
    for layer in self.layers_disc:
        real_pred = layer.feed_forward(real_pred)
```

... which is done by forward propagating both the fake and the real image through all layers of the discriminator. The method `feed_forward` is mostly the same as in

Sect. 18.5. The only difference is that using `store=True` the current value of the input is stored in the object for later reuse.

We can now compare the predictions for the real and for the fake image. For this, we compute the loss for both of them and take the mean of them and for all items in the mini-batch:

```
In [1]: # 4th part of the implementation

def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
    ...
    ...
    # 5 - compare predictions of real and fake with reference
    real_loss = self.cost_function(1., real_pred)
    fake_loss = self.cost_function(0., fake_pred)
    cost_D = 0.5 * np.mean(fake_loss + real_loss)
```

Note that for the real example, the discriminator should output 1, while for the fake example the discriminator should output 0.

Having the cost (i.e., the mean loss, for those readers, who are not used to the notion of “cost”) of the discriminator at hand, we can now back-propagate the error for both predictions:

```
In [1]: # 5th part of the implementation

def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
    ...
    ...
    # 6 - back-propagate errors from both predictions
    real_error = self.derivative_of_cost(1., real_pred)
    for layer in reversed(self.layers_disc):
        real_error = layer.backward_propagation(
            real_error, learning_rate,
            store_update=True, update_weights=False
        )

    fake_error = self.derivative_of_cost(0., fake_pred)
    for layer in reversed(self.layers_disc):
        fake_error = layer.backward_propagation(
            fake_error, learning_rate, use_memory=True
        )
```

In the above code, it becomes also obvious that the implementation of the back-propagation also will need to be changed: as compared to the original implementation, the “state” of the layer (which was stored during the forward propagation; see the parameter `store=True` above) is recalled and used.

The remainder of the training method consists of two blocks inside the two nested `for` loops and two “update” lines after the inner `for` loop is finished:

```
In [1]: # 6th part of the implementation

def train(self, X_train, z_dim, epochs, learning_rate, decay, batch):
    ...
    # 7 - update generator based on feedback from discrimi.
    cost_G = np.mean(self.cost_function(1., fake_pred))

    # 8 - backpropagate errors for the generator
    gen_error = self.derivative_of_cost(1., fake_pred)
    for layer in reversed(self.layers_disc):
        gen_error = layer.backward_propagation(
            gen_error, learning_rate,
            update=False, use_memory=True
        )

    for layer in reversed(self.layers_gen):
        gen_error = layer.backward_propagation(
            gen_error, learning_rate
        )

    all_costs.append([cost_D, cost_G])
    learning_rate *= (1. / (1. + decay * epoch))

    return np.array(all_costs)
```

Here, the cost of the generator was computed based on the fake prediction. For the final update of the weights, all errors are back-propagated through all layers of the discriminator and the generator. In the last two lines inside the `for` loop over all epochs, the costs are stored in a list for monitoring their changes, and the learning rate was adapted according to Eq. 19.16. Finally, the list with the training errors for generator and discriminator is returned.

The last method of the `GAN` class is the method corresponding to `predict` from the fully connected network:

```
In [1]: # 7th part of the implementation

def generate(self, batch, z_dim):
    noise = np.random.normal(0, 1, (batch, z_dim))
    fake = noise
    for layer in self.layers_gen:
        fake = layer.feed_forward(fake)
    return fake
```

... where based on the learned weights, new images are created, starting from noise.

What else is needed for the implementation? As the layer needs to be able to store the input of a forwards pass, the class `FullyConnectedLayer` as well as the classes for different activation layers needs to be adapted. Since there is nothing fundamentally complex in there, we do not show the implementation and refer the reader to the supplementary material.

19.5.7 Two Examples: Handwritten Digits and Microstructures

Training GANs takes a considerable amount of time and computational power, and any real-world application would require GPU computing. However, for small images, we still can set up “toy networks,” which help to understand the architecture, as introduced next.

Example 1: Generating Handwritten Digits

We use the dataset DS-2-light (Alpaydin Digits) for training and are interested in generating images that look similar to the original ones. The images are small enough (8×8) to have a fast training process. We begin by importing the data:

```
In [2]: from sklearn.datasets import load_digits
digits = load_digits()
X = 2 * (digits.data / 16.) - 1
Y = digits.target
```

Next, we define the network architecture both for the discriminator and the generator:

```
In [3]: nn = Gan(BCE, dBCE_dy)

nn.add_layer(FullyConnectedLayer(X.shape[-1], 32), 'disc')
nn.add_layer(LeakyReLULayer(0.02), 'disc')
nn.add_layer(FullyConnectedLayer(32, 1), 'disc')
nn.add_layer(SigmoidLayer(), 'disc')

nn.add_layer(FullyConnectedLayer(z_dim, 32), 'gen')
nn.add_layer(ReLULayer(), 'gen')
nn.add_layer(FullyConnectedLayer(32, 48), 'gen')
nn.add_layer(ReLULayer(), 'gen')
nn.add_layer(FullyConnectedLayer(48, X.shape[-1]), 'gen')
nn.add_layer(TanhLayer(), 'gen')
```

... where `BCE` and `dBCE_dy` is the function for computing the binary cross-entropy loss and its derivative, respectively. Now, we train the network for 2000 epochs, where we start the training with a learning rate of 10^{-4} and plot the cost functions:

```
In [4]: train_gan = nn.train(X, z_dim, epochs=2000, learning_rate=1e-4,
                             decay=1e-5, batch=64)
plt.plot(train_gan[:, 0], label='Discriminator loss')
plt.plot(train_gan[:, 1], label='Generator loss')
plt.legend()
```

The resulting plot is similar to the one shown in the left panel of Fig. 19.16. Images can be generated by:

```
In [5]: n_images = 4
generated_imgs = nn.generate(n_images, z_dim)
generated_imgs = generated_imgs.reshape(-1, 8, 8)
plt.imshow(generated_imgs[1], cmap='gray') # use index 0..(n_images-1)
```

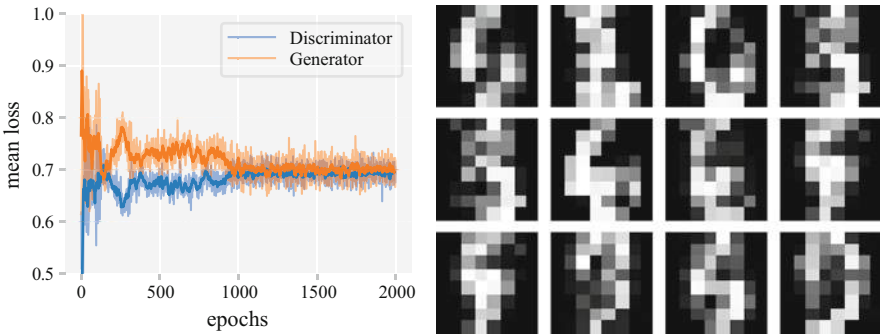


Fig. 19.16 Generating handwritten digits: training loss for the GAN (left panel) where the darker lines show moving averages; the right panel shows images of handwritten digits, generated by the simple GAN implementation

The generated images are shown in the right panel of Fig. 19.16. Taking a look at them, we do see that in most of them, we can clearly identify the number that would correspond to the patterns. Even though the results are not perfect, this is still a very good outcome, given that the training took a few seconds and no GPU was required. The training loss is plotted separately for the discriminator and for the generator. The strong fluctuations are a typical behavior for GANs and also underline that the training for such architecture is more complicated. Above, we mentioned that if the Nash equilibrium is achieved, then the loss values are approximately 0.7. While we are still away from that, both loss values are, on average, about equally far away from that value, which suggests that none of the two network parts is much “stronger” than the other.

Example 2: Generating Microstructure Images

In this example our goal now is to generate artificial microstructure images. For this we start with the dataset MDS-2 light (Ising model, small), which consists of larger images, 16×16 , and therefore contain four times as many pixel as the DS-2-light (Alpaydin Digits). Additionally, just by looking at the images, we probably would estimate that the variance of the features in the dataset MDS-2 light (Ising Model, small) is larger than in the handwritten digit dataset. This will be confirmed during the training, where it turns out that we need significantly more epochs (10000) and also need to use 5000 (instead of ≈ 1800) training images. Additionally, we also found that just using leaky ReLU as activation for the generator was beneficial. The training takes more than two hours on a regular laptop (as opposed to a few seconds for the previous example). In Fig. 19.17 the losses are shown as well as true and the generated images. The losses exhibit much larger fluctuation, and it might be helpful to show a moving average in such cases. While the real images are binary images (only black and white), the generated one also contains gray values in between. This could easily be removed by a simple thresholding. Visually, the images are quite similar, but a person who already looked at several such images would be

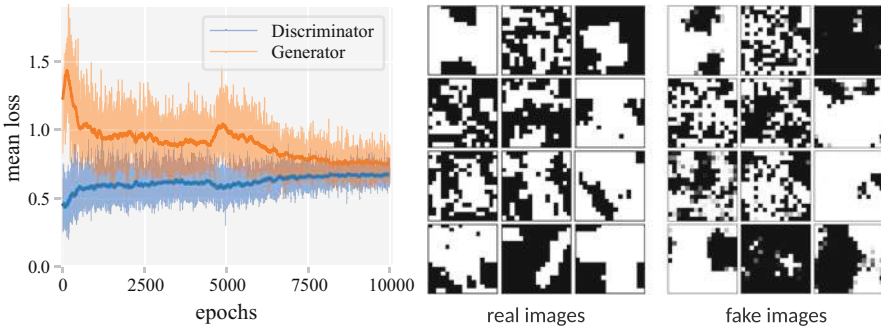


Fig. 19.17 Generating microstructure images: training loss for the GAN (left panel) and true and fake Ising microstructures (middle and right panel). Note that for comparison, the real images are a particular choice of images that roughly exhibit the same characteristics of the structures as the generated once. Otherwise, the “image distribution” would be random

able to tell the difference. For example, in the fake images the noise on the level of individual pixel looks as if it were uniformly random generated and superimposed with other, more structures of larger wavelengths.

Furthermore, it seems as if there are mainly two types of microstructures (i.e., images with two predominant wavelengths). This might indicate the early stage of a *mode collapse* (cf. Sect. 19.5.4).

19.6 Physics-Informed Machine Learning and Beyond

A specialized subfield of DL that has seen a huge interest in recent years is the field of *scientific machine learning*. There, the goal is to incorporate knowledge of a physical problem, e.g., in form of given *partial differential equations* (PDEs) or boundary conditions. Clearly, this might be a very useful approach because the space of possible solutions that a deep neural net could learn can thereby strongly be reduced. Additionally, the consideration of physical constraints should help to increase the accuracy of a training process. Such approaches are also known as *physics informed machine learning* [3]. Physical knowledge can be included, e.g., via a number of different bias types (see, e.g., the overview by KARNIADAKIS et al. [15]).

For example, using *observational bias* is a very simply way of leveraging physical knowledge. Such knowledge can be “injected” into a ML model by simply building a dataset of physics-based data, for instance, from simulations. Then, enforcing certain physical properties in the dataset, such as symmetry or periodicity, helps to improve the training process.

Introducing a *learning bias* is another way of leveraging physical knowledge. Consider, for example, the heat equation which consists of a PDE along with boundary conditions and initial values. Assume that a DL model is to be trained.

Then, it is possible to introduce the information from the heat equation during the training process by incorporating additional terms for the PDE, the initial and the boundary condition into the loss function. Such models have recently gained considerable amount of interest and are called *physics-informed neural network* (PINN) [31, 32]. Typically, the network architecture is a simple, fully connected neural network. PINNs have been applied for predicting PDEs, i.e., forward problems [4, 31, 35] as well as inverse problems [32, 41]. If the total loss function is the sum of the loss for the PDE, the initial and boundary conditions, then such a method is known as *deep Galerkin method* [36]. Additionally, for these methods, the dataset is not given but rather generated by a random sampling strategy *during* the training (this is possible as the PDE is “provided”).

A number of further developments and variations exist, and there have been many attempts to use various architectures as a basis for PINNs instead of fully connected layers, such as convolutional neural networks [10], recurrent neural networks [40], graph networks [34] networks, and even mixed architectures, such as convolutional transformers [19]. However, many of these methods are still rather limited (e.g., for each new boundary condition or initial condition the network needs to be trained from scratch) and far from being able to replace simulations. Also, they still require a substantial amount of development work.

As a last and very promising subject, we briefly mention the topic of *operator learning*. While PINNs attempt to learn the solution of a set of equations, an alternative approach is to learn the mathematical *operators* instead. The most common approach and network architecture is *DeepONet* [26]. By applying the so-called Chen and Chen theorem [6] for nonlinear operator approximation, it is possible to separate the network into two parts: the trunk NN_t network and the branch NN_b network. The trunk network handles the spatial coordinate-based information and performs nonlinear mapping into a higher-dimensional space. At the same time, the branch network operates on the mesh-specific data by providing the information about an input function at a number of different points (the “sensors”). The DeepONet architecture provides a number of potential customization options and is one of the very promising and new models (see, e.g., [25] for an application). A thorough overview of neural operators can be found in the work of KOVACHKI et al. [21].

Clearly, this is a very exciting field in which very active development is taking place. Unfortunately, it is beyond the scope of this book to go into further details of the theory or the implementation and we hope that the above short review serves as a good starting point for further reading or own research.

19.7 Summary and Conclusion

With this, we are at the end of the DL part. We began this journey in the mid of the twentieth century and followed the historical developments starting with single artificial neurons. We then, step by step, developed the single neurons into various assemblies of neurons, arriving at larger and more complex networks.

In the current chapter, we covered many of the more recent developments, ranging from convolutional neural networks and autoencoders to generative models. At the same time, also a number of specialized techniques for creating efficient and stable deep learning models were introduced.

One of the goals of these chapters was to show how simple neural networks and a number of deep learning approaches can still be implemented by just a few lines of Python code. For more advanced network architectures, this is no longer possible and one should rather switch to some of the very mature and computationally efficient DL frameworks such as PyTorch⁵ or TensorFlow.⁶

As a consequence, we did not cover some relevant architectures in depth such as the U-Net or recurrent neural networks for time-dependent problems. Nonetheless, we hope that the reader is now in possession of all relevant foundations and a good amount of hands-on experience; she or he should now be easily able to explore new network types and exciting machine learning methods with the help of the large body of offline and online literature.

19.8 Exercises

19.1 Use the implementation of an ANN with hidden layers from the previous chapter to study the influence of the number of nodes in the hidden layers. Following Sect. 18.5.2, vary the number of output nodes of the hidden layer and compare the testing accuracy for different numbers. Which constellation of node numbers for the hidden layers would be optimal?

19.2 Using the example of the DS-2-light (Alpaydin Digits) dataset using fully connected layers in Sect. 18.5.2, extend the code so that a convolution layer can be used.

- (a) What shape does the input need?
- (b) Comparing the performance of both model, which one gives the best results?
- (c) How does the number of weights in each model influence the performances in terms of accuracy and in terms of computational time? Show the scaling behavior in two diagrams.

19.3 Using the Python implementation of the GAN for the DS-2-light (Alpaydin Digits) dataset from above (the full code can be obtained from supplementary webpage (<https://MDS-book.org>)), discuss the following points:

- (a) How does the number of layer impact the performance of the GAN?
- (b) What happens if you only increase the capacity of one of the layers?
- (c) What is the influence of the number of training example on the generated images?

⁵ <https://pytorch.org/>

⁶ <https://www.tensorflow.org/>

References

1. A. Araujo, W. Norris, and J. Sim. Computing receptive fields of convolutional neural networks. *Distill*, 4(11), Nov. 2019. DOI <https://doi.org/10.23915/distill.00021>.
2. M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 214–223. PMLR, 06–11 Aug 2017. URL <https://proceedings.mlr.press/v70/arjovsky17a.html>.
3. N. Baker, F. Alexander, T. Bremer, A. Hagberg, Y. Kevrekidis, H. Najm, M. Parashar, A. Patra, J. Sethian, S. Wild, K. Willcox, and S. Lee. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. Technical report, Feb. 2019. DOI <https://doi.org/10.2172/1478744>.
4. T. Bandai and T. A. Ghezzehei. Forward and inverse modeling of water flow in unsaturated soils with discontinuous hydraulic conductivities using physics-informed neural networks with domain decomposition. *Hydrology and Earth System Sciences*, 26(16):4469–4495, 2022. DOI <https://doi.org/10.5194/hess-26-4469-2022>. URL <https://hess.copernicus.org/articles/26/4469/2022/>.
5. D. Berthelot, T. Schumm, and L. Metz. Began: Boundary equilibrium generative adversarial networks, 2017.
6. T. Chen and H. Chen. Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Trans Neural Netw*, 6(4):911–917, 1995.
7. K. Crowson, S. Biderman, D. Kornis, D. Stander, E. Hallahan, L. Castricato, and E. Raff. Vqgan-clip: Open domain image generation and editing with natural language guidance, 2022.
8. V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.
9. P. Esser, R. Rombach, and B. Ommer. Taming transformers for high-resolution image synthesis, 2020.
10. H. Gao, L. Sun, and J.-X. Wang. PhyGeoNet: Physics-informed geometry-adaptive convolutional neural networks for solving parameterized steady-state PDEs on irregular domain. *Journal of Computational Physics*, 428:110079, mar 2021. DOI <https://doi.org/10.1016/j.jcp.2020.110079>.
11. I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014.
12. I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville. Improved training of wasserstein gans. In *Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS’17*, page 5769–5779, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.
13. K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.
14. P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5967–5976, 2017. DOI <https://doi.org/10.1109/CVPR.2017.632>.
15. G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, Jun 2021. ISSN 2522-5820. DOI <https://doi.org/10.1038/s42254-021-00314-5>.
16. T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of GANs for improved quality, stability, and variation. In *International Conference on Learning Representations*, 2018.
17. T. Karras, M. Aittala, S. Laine, E. Härkönen, J. Hellsten, J. Lehtinen, and T. Aila. Alias-free generative adversarial networks. In *Neural Information Processing Systems*, 2021a.
18. T. Karras, S. Laine, and T. Aila. A style-based generator architecture for generative adversarial networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(12):4217–4228, Dec 2021b. ISSN 1939-3539. DOI <https://doi.org/10.1109/TPAMI.2020.2970919>.

19. S. Kim, S.-B. Yun, H. Bae, M.-Y. Lee, and Y. Hong. Physics-informed convolutional transformer for predicting volatility surface, 2022.
20. D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.
21. N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar. Neural operator: Learning maps between function spaces, 2021. URL <https://arxiv.org/abs/2108.08481>.
22. M. A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243, Feb. 1991. DOI <https://doi.org/10.1002/aic.690370209>.
23. A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017. DOI <https://doi.org/10.1145/3065386>.
24. Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Dec. 1989. DOI <https://doi.org/10.1162/neco.1989.1.4.541>.
25. W. Li, M. Z. Bazant, and J. Zhu. Phase-field deepoNet: Physics-informed deep operator neural network for fast simulations of pattern formation governed by gradient flows of free-energy functionals, 2023.
26. L. Lu, P. Jin, G. Pang, Z. Zhang, and G. E. Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, mar 2021. DOI <https://doi.org/10.1038/s42256-021-00302-5>.
27. X. Mao, Q. Li, H. Xie, R. Y. Lau, Z. Wang, and S. Paul Smolley. Least squares generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
28. L. Mescheder, A. Geiger, and S. Nowozin. Which training methods for gans do actually converge? In *International Conference on Machine learning (ICML)*, 2018.
29. A. Odena, C. Olah, and J. Shlens. Conditional image synthesis with auxiliary classifier GANs. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2642–2651. PMLR, 06–11 Aug 2017.
30. A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision, 2021.
31. M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations, 2017a. URL <https://arxiv.org/abs/1711.10561>.
32. M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations, 2017b. URL <https://arxiv.org/abs/1711.10566>.
33. O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.
34. S. Seo and Y. Liu. Differentiable physics-informed graph networks, 2019. URL <https://arxiv.org/abs/1902.02950>.
35. Y. Shin. On the convergence of physics informed neural networks for linear second-order elliptic and parabolic type PDEs. *Communications in Computational Physics*, 28(5):2042–2074, Jun. 2020. DOI <https://doi.org/10.4208/cicp.aa-2020-0193>.
36. J. Sirignano and K. Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364, 2018. ISSN 0021-9991. DOI <https://doi.org/10.1016/j.jcp.2018.08.029>. URL <https://www.sciencedirect.com/science/article/pii/S0021999118305527>.
37. K. Sityu, C. Groschner, and M. C. Scott. Understanding the influence of receptive field and network complexity in neural network-guided TEM image analysis. *Microscopy and Microanalysis*, 28(6):1896–1904, Dec. 2022. DOI <https://doi.org/10.1017/s1431927622012466>.

38. P. Trampert, D. Rubinstein, F. Boughorbel, C. Schlinkmann, M. Luschkova, P. Slusallek, T. Dahmen, and S. Sandfeld. Deep neural networks for analysis of microscopy images—synthetic data generation and adaptive sampling. *Crystals*, 11(3):258, Mar. 2021. DOI <https://doi.org/10.3390/cryst11030258>.
39. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
40. B. Wu, O. Hennigh, J. Kautz, S. Choudhry, and W. Byeon. Physics informed rnn-dct networks for time-dependent partial differential equations. In D. Groen, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, editors, *Computational Science—ICCS 2022*, pages 372–379, Cham, 2022. Springer International Publishing. ISBN 978-3-031-08754-7.
41. M. Yin, X. Zheng, J. D. Humphrey, and G. E. Karniadakis. Non-invasive inference of thrombus material properties with physics-informed neural networks. *Computer Methods in Applied Mechanics and Engineering*, 375:113603, Mar. 2021. DOI <https://doi.org/10.1016/j.cma.2020.113603>.
42. J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks, 2020.

In the following we give an overview over the most important definitions and concepts of linear algebra. This serves as an refresher for those readers who already took a class in linear algebra. Even though a big part of this book has been written such that readers without a deep background in vector, matrix, and tensor calculus and analysis should still be able to follow, there are a number of derivations where vector calculus is indispensable.

Nonetheless, this appendix is not an rigorous mathematical introduction. It only aims to introduce all relevant notions and definitions. As a simplification, we always assume that the problems considered are real-valued problems. There are also nearly no examples; for this the reader is referred to the many great text books on this topic. Instead, we use this subject as an opportunity to introduce the corresponding Python (and in particular `numpy`) functions. For an introduction to (scientific) Python programming, refer to one of the many good introduction texts, e.g., [1, 2].

For symbolic mathematical computations the Python package `sympy` is a good choice.¹ `sympy` is part of Python's scientific stack and also can be used for symbolic calculus and analysis; `sympy` expressions also can be converted into `numpy` expression. While we encourage the interested reader to dive into `sympy`, we nonetheless will only focus on `numpy` examples here, as `sympy` requires the user to be a bit familiar with its “philosophy.”

We differentiate between three different types of notations:

1. A purely mathematical formulation using vector/matrix calculus such as scalar products, or, more general, contractions over one or more indices. This notation does not consider multiple data records and just very “mathematically” writes down equations for a single data point. An example is:

¹ <https://docs.sympy.org/>

$$y = ax^2 + bx \quad (\text{A.1})$$

2. The next variant is similar to the first (in terms of notation) but considers the whole data sets and therefore operates with, e.g., vectors for, e.g., a scalar output. As an intermediate step we sometimes show index notations. As an example consider the variable $\mathbf{x}$, which represents a number of measurements of a scalar variable (i.e., each element of $\mathbf{x}$ is a single measurement x_i):

$$y_i = ax_i^2 + bx_i \quad \Leftrightarrow \quad \mathbf{y} = a\mathbf{x}^2 + b\mathbf{x} \quad (\text{A.2})$$

(note that $\mathbf{x}^2$ is the square of all elements, x_i^2).

3. The last step is with view on Python implementation and we often have to think about whether an object is a row or column vector. Also here, index notation comes in handy, in particular when the mathematical operations consist of vector-matrix products or similar operations. For example, assuming that $\mathbf{x}$ is a column vector that contains a number of measurement, the result $\mathbf{y}$ is also a column vector. This can be directly implemented in Python:

$$y_i = ax_i^2 + bx_i \quad \rightarrow \quad \mathbf{y} = \mathbf{a} * \mathbf{x} ** 2 + \mathbf{b} * \mathbf{x} \quad (\text{A.3})$$

For this simple equation it does not matter which shape the numpy array $\mathbf{x}$ has. The difference between Eq. A.2 and Eq. A.3 becomes more significant once Eq. A.2 consists of more complex operations, such as computing the mean of all records, as we already saw in, e.g., Chap. 13.

A.1 Vector Calculus

A.1.1 Definition of Vectors

A *position vector* to a point P starts at the origin O of the chosen coordinate system and ends at the point. It is thus a means to describe the position of points with respect to the coordinate origin. A vector in an n -dimensional *vector space* has n components.

A.1.2 Conventions: Row and Column Vectors

As a commonly accepted convention, vectors are, if not stated differently, considered as *column vectors*. A column vector $\mathbf{a}$ and a row vector $\mathbf{b}$ are shown as:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad \text{and} \quad \mathbf{b} = [b_1 \ b_2 \ \cdots \ b_n] \quad \text{where} \quad a_i, b_i \in \mathbb{R}. \quad (\text{A.4})$$

There, the a_i and b_i are the *vector components*. The reason why column vectors are the “default” is that one of the common operations is the matrix-vector product where the matrix represents a linear function and “acts” on the vector. This product (for more details see below) is only defined if $\mathbf{a}$ is a column vector.

Row vectors are also used but at least in this text we are very explicit when we consider a vector a row vector.

A column vector can be turned into a row vector (and vice versa) by transposing it; see Appendix A.1.3.

A.1.3 Transposition of a Vector

A row vector is converted into a column vector by *transposing* it. This is indicated by the superscript $^\top$, e.g., the column vector $\mathbf{a}$ becomes a row vector $\mathbf{b}$ by $\mathbf{b} = \mathbf{a}^\top$, and the row vector $\mathbf{b}$ becomes a column vector $\mathbf{a}$ by $\mathbf{a} = \mathbf{b}^\top$.

A.1.4 Scalar Product of Vectors

The scalar product of two vectors results in a scalar, i.e., a single number. It is also called *or dot product* and is defined as follows:

Definition A.1 (Scalar Product of Two Vectors) Given are two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, i.e., each vector consists of n entries. The scalar product $\mathbf{a} \cdot \mathbf{b}$ is then defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n a_i b_i \in \mathbb{R} \quad (\text{A.5})$$

The dot (“ $\cdot$ ”) between the two vectors symbolizes the scalar product.

In this definition, it is not necessary to differentiate between row and column vector. However, if we assume as convention that $\mathbf{a}$ and $\mathbf{b}$ are column vectors, then their scalar product is written without the dot using the transposition symbol (which turns column vectors into row vectors and vice versa; cf. the general definition Appendix A.2.7):

$$\mathbf{c} = \mathbf{a}^\top \mathbf{b} . \quad (\text{A.6})$$

This notation is often followed in the context of matrix-vector multiplications, and it is not advisable to mix the two notations.

A.1.5 Orthogonal Vectors

Geometrically speaking, two vectors are orthogonal if they are perpendicular to each. The mathematical definitions use the dot or scalar product:

Definition A.2 (Orthogonal Vectors) Two vectors $\mathbf{a}$ and $\mathbf{b}$ are said to be orthogonal if their scalar product vanishes:

$$\mathbf{a} \cdot \mathbf{b} = 0 . \quad (\text{A.7})$$

As the dot product gives the projection of one vector onto the other vector, it is clear that $\mathbf{a} \cdot \mathbf{b} = 0$ implies an angle of 90° between them—at least in the considered two-dimensional plane. This definition can easily be generalized to an arbitrary set of vectors:

Definition A.3 (Mutually Orthogonal Vectors) A set of n vectors $\{\mathbf{v}_1, \dots, \mathbf{v}_n\}$ are said to be *mutually orthogonal* if each possible pair of vector is orthogonal, i.e.:

$$\mathbf{v}_i \cdot \mathbf{v}_j = 0 \quad \text{for all } i \text{ and } j \text{ with } i \neq j . \quad (\text{A.8})$$

A.1.6 Euclidean Norm of a Vector

The *Euclidean norm* of a vector can be written in terms of the scalar product.

Definition A.4 (Euclidean Norm of a Vector) Given is a vector $\mathbf{a} \in \mathbb{R}^n$, i.e., the vector consists of n entries. The Euclidean norm $\|\mathbf{a}\|$ is then defined as:

(continued)

$$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}} = \sqrt{\sum_{i=1}^n (a_i)^2} \in \mathbb{R}. \quad (\text{A.9})$$

The Euclidean norm is also called L_2 norm (or L_2 -norm). It gives the distance of a point $\mathbf{a}$ from the origin of the Euclidean coordinate system which is why we often say that “ $\|\mathbf{a}\|$ is the length of the vector $\mathbf{a}$.” To differentiate the L_2 -norm from other norms, sometimes the “2” is added and we use $\|\cdot\|_2$. In the context of *machine learning* (ML) also other norms are used; however, in the context of linear algebra this is by far the most commonly used norm.

If the length of a vector $\mathbf{a}$ is 1, then the vector is called a *unit vector*:

$$\|\mathbf{a}\| = 1 \quad \Leftrightarrow \quad \mathbf{a} \text{ is a unit vector.} \quad (\text{A.10})$$

The most well-known unit vectors are the *Euclidean unit basis vectors*, e.g., $\mathbf{e}_1 = [1, 0, \dots, 0] \in \mathbb{R}^n$ or $\mathbf{e}_2 = [0, 1, 0, \dots, 0] \in \mathbb{R}^n$.

A.1.7 Euclidean Distance

Closely related to the Euclidean norm is the *Euclidean distance* which gives the distance between two points:

Definition A.5 (Euclidean Distance) Given two points p_1 and p_2 , represented by the vectors $\mathbf{v}_1$ and $\mathbf{v}_2 \in \mathbb{R}^n$, respectively, the Euclidean distance D_2 between the points is defined as:

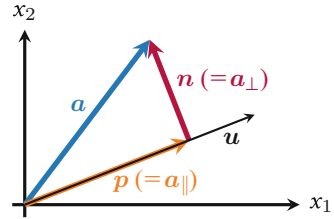
$$D_2(\mathbf{v}_1, \mathbf{v}_2) = \|\mathbf{v}_2 - \mathbf{v}_1\| \in \mathbb{R}. \quad (\text{A.11})$$

In this definition, the subscript “2” indicates that the L_2 norm is used. Further distance metrics are defined in the main text, e.g., in Chap. 14.

A.1.8 Orthogonal Projection of a Vector

Projecting one vector $\mathbf{a}$ onto another vector $\mathbf{u}$ is a task that is encountered frequently in machine learning and data mining, e.g., in the context of dimensionality reduction and PCA in Sect. 15.2. Figure A.1 shows the geometry: there, $\mathbf{a}$ is the vector to be projected orthogonally onto $\mathbf{u}$. *Orthogonal vector projection* implies

Fig. A.1 Projection of a vector $\mathbf{a}$ on a vector $\mathbf{u}$: $\mathbf{a}$ is decomposed into a vector $\mathbf{p}$ parallel to $\mathbf{u}$ and a vector $\mathbf{n}$ perpendicular to $\mathbf{u}$ such that $\mathbf{a} = \mathbf{p} + \mathbf{n}$



that $\mathbf{a}$ is decomposed into a vector that is parallel to $\mathbf{u}$ and one that is perpendicular to $\mathbf{u}$ such that:

$$\mathbf{a} = \mathbf{a}_{\parallel} + \mathbf{a}_{\perp} . \quad (\text{A.12})$$

The derivation of $\mathbf{a}_{\parallel}$ is based on the fact that there exists a scalar value c such that $\mathbf{a}_{\parallel} = c\mathbf{u}$, i.e., $\mathbf{a}_{\parallel}$ is just a scaled version of $\mathbf{u}$. As $\mathbf{a}_{\perp}$ and $\mathbf{a}_{\parallel}$ are perpendicular to each other, their scalar product must vanish, and it follows:

$$\mathbf{a}_{\perp} \cdot \mathbf{u} = 0 \quad (\text{A.13})$$

$$\Leftrightarrow (\mathbf{a} - \mathbf{a}_{\parallel}) \cdot \mathbf{u} = (\mathbf{a} - c\mathbf{u}) \cdot \mathbf{u} = 0 \quad (\text{A.14})$$

We can now use the distributive law for vectors, separate the two terms, and obtain the constant c :

$$\mathbf{a} \cdot \mathbf{u} = c\mathbf{u} \cdot \mathbf{u} \quad (\text{A.15})$$

$$\Rightarrow c = \frac{\mathbf{a} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} . \quad (\text{A.16})$$

From this it follows for the parallel component of $\mathbf{a}$ that:

$$\mathbf{a}_{\parallel} = \frac{\mathbf{a} \cdot \mathbf{u}}{\mathbf{u} \cdot \mathbf{u}} \mathbf{u} . \quad (\text{A.17})$$

In the notation used in the ML chapters, quite a number of different super- and subscripts are used. Therefore, we often rather use $\mathbf{p}$ (for *parallel*) instead of $\mathbf{a}_{\parallel}$ and $\mathbf{n}$ (for *normal*) instead of $\mathbf{a}_{\perp}$.

A.1.9 Gradient of a Scalar Field and a Vector Field

Assume that at each point $\mathbf{x} = (x_1, x_2, x_3) \in \mathbb{R}^3$ there is a vector-valued quantity, $\mathbf{a}(x_1, x_2, x_3)$. This defines a *vector field*. Then the gradient of the vector field, $\nabla \mathbf{a}$, is given by:

$$\nabla \mathbf{a} = \begin{bmatrix} \frac{\partial a_1}{\partial x_1} & \frac{\partial a_2}{\partial x_1} & \frac{\partial a_3}{\partial x_1} \\ \frac{\partial a_1}{\partial x_2} & \frac{\partial a_2}{\partial x_2} & \frac{\partial a_3}{\partial x_2} \\ \frac{\partial a_1}{\partial x_3} & \frac{\partial a_2}{\partial x_3} & \frac{\partial a_3}{\partial x_3} \end{bmatrix} \quad (\text{A.18})$$

and the j th derivative of the i th component of $\mathbf{a}$ is:

$$\frac{\partial a_i}{\partial x_j}, \quad (\text{A.19})$$

which is the same as the short form $\partial_j a_i$. A dimensionality of the field can be handled in analogy

A *scalar field* $\varphi(x_1, x_2, x_3)$ is understood as a special case of the vector field. Therefore, it is:

$$\nabla \varphi = \begin{bmatrix} \frac{\partial \varphi}{\partial x_1} \\ \frac{\partial \varphi}{\partial x_2} \\ \frac{\partial \varphi}{\partial x_3} \end{bmatrix} \quad (\text{A.20})$$

and the j th derivative of φ is $\frac{\partial \varphi}{\partial x_j}$ which can also be abbreviated as $\partial_j \varphi$.

A.2 Matrices and Matrix Operations

A.2.1 Basic Definition and Creation of Matrices in Python

In analogy to Sect. 3.2.4 we define a matrix $\mathbf{A}$ consisting of m rows and n columns of real-valued elements.

Definition A.6 (Matrix Consisting of m Rows and n Columns) A matrix $\mathbf{A}$ consisting of m rows and n columns of real-valued elements has the following structure:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad \text{with } a_{ij} \in \mathbb{R}. \quad (\text{A.21})$$

There, an element in row number i (counted from the top) and column number j (counted from left) is denoted by a_{ij} . The matrix in Eq. A.21 is called a $m \times n$

matrix. Sometimes, we use more complex expressions as indices. In these cases we might use a comma in between the row and the column index, as in, e.g., $A_{i,2(j+1)}$.

For matrices a number of basic mathematical operations exists which are introduced in the following sections.

Python implementation: In Python, a number of methods for creating matrices exists. A matrix consisting of 4 rows and 4 columns of zeros can be created by:

```
In [1]: import numpy
A = numpy.zeros((3, 4), dtype=float)
A
```

```
Out [1]: array([[0., 0., 0., 0.],
               [0., 0., 0., 0.],
               [0., 0., 0., 0.]])
```

A 3×4 matrix consisting of the Boolean values True can be created by:

```
In [2]: import numpy
A = numpy.ones((3, 4), dtype=bool)
A
```

```
Out [2]: array([[ True,  True,  True,  True],
               [ True,  True,  True,  True],
               [ True,  True,  True,  True]])
```

A matrix populated with random values in between 0 and 1 can be created using `numpy.random.rand(n_rows, n_columns)`, and to create a matrix **A** with undefined (or uninitialized) entries and with the same shape as a matrix **B**, one can use `numpy.empty_like(B)`, to name but a few options.

A.2.2 Addition of Matrices

Definition A.7 (Addition of Matrices) Addition of matrices is defined by element-wise addition if the two matrices **A** and **B** have the same number of rows and columns:

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix}, \quad (\text{A.22})$$

Each element of the resulting matrix is again a real number. Furthermore, it is directly clear that $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$.

Python implementation: Addition works as expected in numpy—simply by adding up the two matrices:

```
In [1]: import numpy
A = numpy.array([[1.1, 2.3, -0.1],
                 [0., 0.5, 2.1]])
B = numpy.array([[4.3, 0.2, 0.],
                 [0.5, 1.7, 0.3]])
A + B
```

```
Out [1]: array([[ 5.4,  2.5, -0.1],
                [ 0.5,  2.2,  2.4]])
```

A.2.3 Multiplication of a Matrix with a Scalar

Definition A.8 (Multiplication of a Matrix with a Scalar) A matrix $\mathbf{A}$ is multiplied with a scalar c by multiplying each element with the scalar:

$$c\mathbf{A} = \mathbf{A}c = \begin{bmatrix} ca_{11} & \cdots & ca_{1n} \\ \vdots & & \vdots \\ ca_{m1} & \cdots & ca_{mn} \end{bmatrix} \quad (\text{A.23})$$

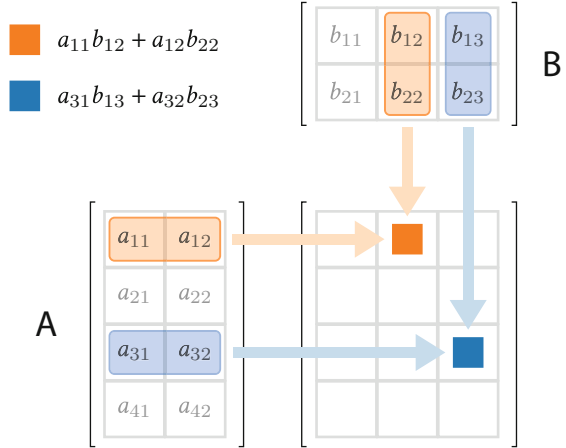
Scalar multiplication works equally for multiplying with a scalar from right or from left. Note that the “scalar multiplication” should not be confused with the “scalar product” which is an operation between two vectors, resulting in a scalar (hence the name).

Python implementation: In Python this type of product is performed as expected, e.g., the statement `2.4 * A` multiplies each element of `A` with 2.4.

A.2.4 Matrix Product

The matrix product is, in a way, a generalization of the scalar product. It is given by the following definition:

Fig. A.2 Multiplication of two matrices: visualization of the involved rows and columns. The “inner” dimensions of the matrices **A** and **B** must agree for the summation to work. Here, the product can be computed as **A** has two columns and **B** has two rows. The product **BA** would not be possible



Definition A.9 (Product of Two Matrices) The multiplication of two matrices requires that the number of columns of the left equals the number of rows of the right matrix. Then, the product **AB** results in a new matrix **C** which has the same number of rows as the left matrix and the same number of columns as the right matrix. The element c_{ij} is given by:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{p=1}^n a_{ip}b_{pj}. \quad (\text{A.24})$$

In other words, c_{ij} is obtained by multiplying all elements of the i th row of the left matrix with the j th column of the right matrix and then summing all results of the products up. A visualization of the operation is shown in Fig. A.2.

Python implementation: numpy has several options for matrix-matrix multiplication as, e.g., the function `dot`, `matmul`, or the operator `@` which is a short form for `matmul`:

```
In [1]: import numpy
A = numpy.array([[1.1, 2.3, -0.1],
                 [0., 0.5, 2.1]])
B = numpy.array([[4.3, 0.2, 0.],
                 [0.5, 1.7, 0.3],
                 [0.5, 1.7, 0.3]])
A @ B # numpy.dot(A, B) and numpy.matmul(A, B) give same results as A @ B

Out [1]: array([[5.83, 3.96, 0.66],
                [1.3 , 4.42, 0.78]])
```

Further Rules for Matrices

Matrix multiplication obeys the following rules (the proofs can be found in most textbooks on linear algebra, and the reader is encouraged to do them as an exercise):

$$\text{Associativity:} \quad (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}) \quad (\text{A.25})$$

$$\text{Left Distributivity:} \quad (\mathbf{A} + \mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC} \quad (\text{A.26})$$

$$\text{Right Distributivity:} \quad \mathbf{C}(\mathbf{A} + \mathbf{B}) = \mathbf{CA} + \mathbf{CB} \quad (\text{A.27})$$

Note that in general $\mathbf{AB} \neq \mathbf{BA}$. As an example, we show numerically that the left distributive law holds for a *particular choice of numbers* (this is not a mathematical proof!). We start by creating the three matrices:

```
In [1]: import numpy
A = numpy.random.rand(2, 3)
B = numpy.random.rand(2, 3)
C = numpy.random.rand(3, 4)
```

and then compute the two terms of the left distributive law:

```
In [2]: (A + B) @ C
```

```
Out [2]: array([[1.52760876, 1.58518409, 1.68986657, 1.62772657],
               [1.14112041, 1.06058372, 1.19346825, 1.38466784]])
```

```
In [3]: A @ C + B @ C
```

```
Out [3]: array([[1.52760876, 1.58518409, 1.68986657, 1.62772657],
               [1.14112041, 1.06058372, 1.19346825, 1.38466784]])
```

Looking at the numbers reveals already that both formulations seem to be equivalent. A better way of comparing *all* elements of the two arrays is to use the function `allclose`, which results in `True` if all elements are identical within a certain absolute and relative tolerance:

```
In [4]: numpy.allclose((A + B) @ C,
                       A @ C + B @ C)
```

```
Out [4]: True
```

A.2.5 Multiplication of a Matrix with a Vector

Multiplication of a matrix with a vector is the special case of multiplication of a matrix with a matrix that has only one column (see Appendix A.2.4). For example, assume that $\mathbf{x}$ is a column vector. Then we have

$$\mathbf{Ax} := x_1 \mathbf{a}_{:,1} + x_2 \mathbf{a}_{:,2} + \cdots + x_n \mathbf{a}_{:,n}, \quad (\text{A.28})$$

where $\mathbf{a}_{:,i}$ is the i th column of $\mathbf{A}$. Note that we do not use a dot between $\mathbf{A}$ and $\mathbf{x}$. In index notation this is written as

$$\mathbf{Ax} := \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix} \quad (\text{A.29})$$

The following python example is also used to remind the reader that for multiplication from the right the vector `b` has to be a column vector:

```
In [1]: import numpy
import numpy
A = numpy.array([[1, 2, 3], [4, 5, 6]])
b = numpy.array([[10], [20], [30]])

# Alternative: b = numpy.array([10, 20, 30]).reshape(-1, 1)
```

Sometimes it is useful to start with defining the elements of a row vector and then convert it into a column vector as shown in the commented outline above. The multiplication can be done using the abbreviation `@` as in

```
In [2]: A @ b

Out [2]: array([[140],
               [320]])
```

or equivalently using `A.dot(b)` (where the function `dot` is a method of the object `A`) or `numpy.dot(A, b)` (where `dot` is a function of the module `numpy`).

A.2.6 Hadamard Product of Matrices

There are a number of situations in machine learning where two matrices need to be multiplied *element-wise*. This is called *Hadamard product*, indicated by the symbol $\odot$. It is defined for matrices $\mathbf{A}$ and $\mathbf{B}$ as:

$$\mathbf{C} = \mathbf{A} \odot \mathbf{B} \quad (\text{A.30})$$

$$c_{ij} = a_{ij}b_{ij}. \quad (\text{A.31})$$

Python implementation: In numpy this type of multiplication simply uses the multiplication symbol and looks as follows:

```
In [1]: A = numpy.array([[1, 2, 3],
                        [4, 5, 6]])
A * A
```

```
Out [1]: array([[ 1,  4,  9],
               [16, 25, 36]])
```

A.2.7 Transposition of a Matrix

The transposition of a matrix $\mathbf{A}$ is obtained by “mirroring” its elements across the diagonal, i.e., effectively interchanging rows with columns.

Definition A.10 (Transposition of a Matrix) If $\mathbf{A}$ is a matrix with m rows and n columns and its elements are given by a_{ij} with $i \in [1, m]$ and $j \in [1, n]$, then the elements of the transposed matrix $\mathbf{A}^T$ are given by a_{ji} .

The transposition operation is indicated by the superscript $(\bullet)^T$ (where the bullet symbol “•” serves as a placeholder), e.g., $\mathbf{A}^T$. In more detail the transposition looks like this:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \Rightarrow \mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}. \quad (\text{A.32})$$

For the transposition operation a number of rules and identities can be proven:

$$\mathbf{AB} = \mathbf{B}^T \mathbf{A}^T \quad (\text{A.33})$$

$$(c\mathbf{A})^T = c(\mathbf{A}^T) \quad (\text{A.34})$$

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T \quad (\text{A.35})$$

$$(\mathbf{A}^T)^T = \mathbf{A}. \quad (\text{A.36})$$

Python implementation: Transposing a matrix $\mathbf{A}$ in numpy is simple: we either use the function `numpy.transpose(A)` or the transposition operator `A.T`.

A.2.8 Neutral Element for Matrix Addition

For matrix addition, the “zero matrix” $\mathbf{0}$ contains only zeros and is the neutral element w.r.t. addition:

$$\mathbf{A} + \mathbf{0} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} = \mathbf{A}. \quad (\text{A.37})$$

This also holds for addition from the left and from the right such that $\mathbf{A} + \mathbf{0} = \mathbf{0} + \mathbf{A} = \mathbf{A}$.

Python implementation: Creating a “zero matrix” in Python is done using the function `numpy.zeros`, as already shown above.

A.2.9 Neutral Element for Matrix Multiplication

For matrix multiplication, the neutral element is the identity matrix $\mathbf{I}$ which contains ones on the main diagonal and zeros everywhere else. It has the same number of rows and columns as the matrix $\mathbf{A}$:

$$\mathbf{I}\mathbf{A} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{1n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \mathbf{A}, \quad (\text{A.38})$$

which can be proven by performing the multiplication element-wise. $\mathbf{I}$ is also the neutral element for multiplication from the right such that $\mathbf{I}\mathbf{A} = \mathbf{A}\mathbf{I} = \mathbf{A}$.

Python implementation: Creating a square identity matrix with 3 rows and 3 columns in python can be done as follows:

```
In [1]: import numpy
A = numpy.eye(3)
A

Out [1]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

A.2.10 Square Matrix

A square matrix has the same number of rows as number of columns. If the number of rows is n , then n is called the *order of the matrix*. Two square matrices that have

the same order can be added or multiplied. Square matrices play an important role in many applications and have a number of interesting properties, in particular when they are additionally symmetric (see Appendix A.2.12).

Python implementation: Testing if a matrix has the correct number of rows and columns can be very helpful. For example, for ensuring that the matrix `A` is in fact a square matrix, one can do the following:

```
In [1]: from numpy import array, shape
A = array([[1, 2, 3],
          [2, 4, 6]])
assert A.shape[0] == A.shape[1], f'A has different number of rows' \
                                     f' and columns: {A.shape}'

Out [1]: ...
AssertionError: Matrix has different number of rows and columns: (2, 3)
```

Note the use of `assert` which gives a (hopefully helpful) error message in case that the assertion conditions resulted in `False`.

A.2.11 Inverse Matrix

Definition A.11 (Inverse Matrix) Assume that $\mathbf{A}$ is a square matrix. Then, $\mathbf{B}$ is termed *inverse matrix of $\mathbf{A}$* if:

$$\mathbf{AB} = \mathbf{I}, \quad (\text{A.39})$$

where $\mathbf{I}$ is the identity matrix. $\mathbf{A}$ is then called *invertible*.

The inverse of a matrix is usually denoted by the superscript $(\bullet)^{-1}$, e.g., the inverse of $\mathbf{A}$ is $\mathbf{A}^{-1}$. Note that matrix inversion is not possible for all kinds of values, i.e., only in cases where the matrix is non-singular. The determinant (see Appendix A.2.16) can be used to determine if a matrix is invertible or not. For invertible matrices the following rules hold:

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \quad (\text{A.40})$$

$$(\mathbf{AB})^{-1} = \mathbf{A}^{-1}\mathbf{B}^{-1} \quad (\text{A.41})$$

There exist various methods and strategies for computing the inverse of a matrix, e.g., *Gaussian elimination* which is also used for solving linear systems of equations and using the so-called *adjoint* of a matrix together with the determinant are two of

such methods. For further details, the reader is referred to the literature on linear algebra.

Python implementation: The `linalg` module of `numpy` has a function for computing the inverse of a matrix:

```
In [1]: from numpy import array, inv
        A = array([[2, 1],
                   [0, 1]])
        A_inv = inv(A)
        A @ A_inv
```

```
Out [1]: array([[1., 0.],
               [0., 1.]])
```

where the resulting identity matrix shows that `A_inv` is indeed the inverse matrix of `A`.

A.2.12 Symmetric Matrix

Symmetric matrices play an important role in linear algebra as they have a number of useful properties. The following is their definition:

Definition A.12 (Symmetric Matrix) A matrix $\mathbf{A}$ is called *symmetric* if transposing it does not change the values of the matrix elements:

$$\mathbf{A} = \mathbf{A}^T. \quad (\text{A.42})$$

This definition implies that a symmetric matrix is also a square matrix. Additionally, it can be shown that symmetric matrices exhibit the following properties:

- All eigenvalues of symmetric matrices are *real-valued*.
- Eigenvectors of $\mathbf{A}$ are *orthogonal*.
- An $n \times n$ matrix is symmetric if and only if there are n eigenvectors that form an orthonormal basis in $\mathbb{R}^n$.
- Symmetric matrices are *diagonalizable*.

More details and derivations are given in Sect. A.3, and in particular the aspect of *diagonalization* is a consequence of the *spectral theorem*, Appendix A.3.4. If a symmetric matrix $\mathbf{A}$ is also invertible, then additionally the following relation holds:

$$(\mathbf{A}^T)^{-1} = (\mathbf{A}^{-1})^T. \quad (\text{A.43})$$

The “chaining” of transposition and inversion operations in arbitrary order is often abbreviated as $(\bullet)^{-T}$ where the bullet symbol “ $\bullet$ ” serves as a placeholder.

A.2.13 Triangular Matrices

An often encountered task in linear algebra is to decompose a matrix into a product of typically simpler matrices. One among such matrix types are triangular matrices:

Definition A.13 (Triangular Matrix) A square matrix $\mathbf{A}$ is called a *triangular matrix* if all elements above or below the main diagonal are zero.

$\mathbf{A}$ is called *lower triangular matrix* if the elements a_{ij} of $\mathbf{A}$ fulfill $a_{ij} = 0$ for $j > i$, i.e., all elements above the main diagonal are zero. The matrix is called *upper triangular matrix* if the elements fulfill $a_{ij} = 0$ for $j < i$, i.e., all elements below the main diagonal are zero.

An example for a lower triangular matrix is the following matrix:

$$\mathbf{A} = \begin{bmatrix} 1.3 & 0 & 0 & 0 \\ 2.7 & 4.0 & 0 & 0 \\ 9.6 & 3.3 & 1.8 & 0 \\ 7.4 & 0.0 & 2.4 & 2.1 \end{bmatrix}. \quad (\text{A.44})$$

The `numpy` function `numpy.tril(A)` returns the lower triangular matrix of $\mathbf{A}$, and `numpy.triu(A)` returns the upper triangular matrix.

A.2.14 Trace of a Matrix

The sum of all entries on the main diagonal is called the trace of a matrix:

Definition A.14 (Trace of a Matrix) The trace of a matrix $\mathbf{A}$ is abbreviated as $\text{tr}(\mathbf{A})$ and defined as:

$$\text{tr}(\mathbf{A}) = a_{11} + a_{22} + \cdots + a_{nn} \quad \text{where } \mathbf{A} \in \mathbb{R}^{n,n}. \quad (\text{A.45})$$

The trace occurs, e.g., in many continuum mechanical problems as well as in the eigenvalue equation Eq. A.50.

Python implementation: Computing the trace of a matrix could be done by element-wise multiplication of the matrix with the identity matrix, followed by summation—or we use the `numpy` function as follows:

```
In [1]: import numpy

A = numpy.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
A.trace()    # alternative: numpy.trace(A)
```

```
Out [1]: 15
```

A.2.15 Rank of a Matrix

Definition A.15 (Matrix Rank) Assume that a matrix $\mathbf{A}$ consists of n columns, the column vectors $\mathbf{a}_i$. The rank of $\mathbf{A}$, $\text{rank}(\mathbf{A})$, is then the maximum number of linearly independent vectors $\mathbf{a}_i$, or alternatively, the dimension of the vectors' space spanned by these vectors.

It can be shown that in this definition we can also equivalently use the number of linearly independent rows of the matrix to determine the rank. In case of a square matrix, a matrix is said to have *full rank* if the number of independent rows or columns is equal to the total number of rows or columns. Only a matrix that has full rank is invertible.

Python implementation: `numpy` provides a numerical function which is based on *singular value decomposition* (SVD). This function is based on the calculation of eigenvalues; it offers a parameter for prescribing a tolerance as a guard against numerical errors. In practice this becomes relevant only for an extreme value range or badly scaled numbers. Here is how to compute the rank of a matrix:

```
In [1]: import numpy

A = numpy.array([[1, 2, 3],
                 [2, 4, 6],
                 [0, 8, 9]], dtype=float)
numpy.linalg.matrix_rank(A)
```

```
Out [1]: 2
```

Obviously, the first two rows are just multiples of each other which explains the rank of 2. In the next example, the matrix is an upper triangular matrix with full rank:

```
In [2]: import numpy
A = numpy.array([[1, 2, 3],
                 [0, 4, 6],
                 [0, 0, 9]], dtype=float)
numpy.linalg.matrix_rank(A)
```

```
Out [2]: 3
```

A.2.16 Determinant

In [ML](#) there are a number of situations where linear systems of equations need to be solved—which in matrix-vector form requires or results in the inversion of a matrix. Linear algebra provides a concept for analysis of such matrices: the *determinant*. Determinants are additionally used in the context of computing eigenvalues and eigenvectors.

Again, we assume that $\mathbf{A}$ is a square matrix. Then, the determinant is a scalar that characterizes certain properties of the matrix. One of such property is that a matrix is invertible if and only if its determinant is nonzero. We write the determinant as $\det(\mathbf{A})$; in the literature also the notation $|\mathbf{A}|$ is commonly used. The following shows without proof how the determinant of a 2×2 matrix looks like (it is therefore not a strict mathematical definition):

Definition A.16 (Determinant of a 2×2 Matrix) If $\mathbf{A}$ is a 2×2 matrix, then the determinant is:

$$\det(\mathbf{A}) = \det \left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \right) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} . \quad (\text{A.46})$$

(A general definition for $n \times n$ matrices can make use of the *Leibniz formula* or the *Laplace expansion*, but for brevity we have to refer the interested reader to textbooks on linear algebra for this.)

In the above case of the 2×2 matrix we can use the determinant to compute the inverse of $\mathbf{A}$ as follows:

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{bmatrix} a_{22} & -a_{12} \\ -a_{21} & a_{11} \end{bmatrix} , \quad (\text{A.47})$$

which is why $\mathbf{A}$ is invertible only if $\det(\mathbf{A}) \neq 0$. Determinants of matrices with more rows (and equal number of columns) can be computed either with specialized rules (e.g., for 3×3 matrices this is the *Sarrus rule*) recursively using the concept of

sub-determinants through the *Laplace expansion* or numerically. We conclude this brief paragraph with listing a few rules for computing with determinants:

- *Product of matrices:* $\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$
- *Invariance w.r.t. transposition:* $\det(\mathbf{A}) = \det(\mathbf{A}^T)$
- *Scalar multiplication:* $\det(\lambda \mathbf{A}) = \lambda^n \det(\mathbf{A})$ where n is the rank of the matrix
- *Invariance w.r.t. change of basis:* The determinant of $\mathbf{A}$ stays unchanged if a matrix is represented in a new basis systems (e.g., after a rotation of the Euclidean coordinate frame)
- *Square matrices:* The determinant of a square matrix is the product of all diagonal elements.

Python implementation: For numerical computation of the determinant, `numpy` provides the function `det` which is contained in the `linalg` module.

A.2.17 Orthogonal Matrix

Definition A.17 (Orthogonal Matrix) An orthogonal matrix $\mathbf{Q}$ is a matrix for which the following relation holds:

$$\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}, \quad (\text{A.48})$$

where $\mathbf{Q}$ is a quadratic matrix and $\mathbf{I}$ is the identity matrix.

As a consequence the inverse of an orthogonal matrix is identical to its transpose:

$$\mathbf{Q}^{-1} = \mathbf{Q}^T \quad (\text{A.49})$$

which is why orthogonal matrices are very useful as computing the inverse is extremely simple. The following additional properties holds:

- The determinant of an orthogonal matrix is either $+1$ or -1 .
- Orthogonality implies that the eigenvectors of the matrix are linearly independent and therefore form a basis of a vector space.
- While matrices in general represent linear transformations, any orthogonal matrix $\mathbf{Q}$ with $\det(\mathbf{Q}) = +1$ represents a rotation.

There are many applications for orthogonal matrices. Among those is the MP3 compression (a normalized version of the discrete cosine transform) which can be

represented by an orthogonal matrix. Further applications are principal component analysis as discussed in Sect. 15.2.

Python implementation: To show that a matrix is orthogonal we only need to show that the product $\mathbf{Q}\mathbf{Q}^T$ results in the identity matrix. We do this by creating a “rotation matrix” from which we already know that it should be orthogonal:

```
In [1]: from numpy import array, sin, cos, allclose, eye, pi

# create a matrix Q
theta = pi / 8
Q = array([[cos(theta), -sin(theta)], [sin(theta), cos(theta)]])

# create a 2x2 identity matrix
I = eye(2)

# show that it's an orthogonal matrix
allclose(Q @ Q.T, I), allclose(Q.T @ Q, I)
```

```
Out [1]: (True, True)
```

Here, we used the numpy function `numpy.allclose` which compares all elements of the two given matrices and returns `True` if and only if all elements are identical (up to roundoff errors).

A.3 Derived Properties, Further Theorems, and Advanced Definitions

A.3.1 Eigenvalues, Eigenvectors, and the Characteristic Polynomial

Mathematical details: Eigenvalues are way to express what happens to a matrix (and in particular to its eigenvectors) when it is subjected to a linear mapping, i.e., a particular kind of change of “coordinate frame.” The German word “eigen” can be translated as *characteristic* or *intrinsic*. This stems from the fact that eigenvectors often contain information of fundamental (physical) properties, e.g., in vibrations of mechanical systems the eigenvalues are the frequencies with which the system vibrates, and the Hartree-Fock theory defines atomic orbitals as the eigenvectors of the so-called Fock operator.

We start with the following definition:

Definition A.18 (Eigenvalue Equation) For the real-valued, square matrix $\mathbf{A}$, the number $\lambda \in \mathbb{R}$ is called an *eigenvalue* of $\mathbf{A}$ if the eigenvalue equation:

$$\mathbf{A}v = \lambda v \quad (\text{A.50})$$

(continued)

holds. There, $\mathbf{v}$ is the corresponding *eigenvector* which is assumed to be a nonzero column vector $\mathbf{v} \in \mathbb{R}^n$.

Eigenvalues and eigenvectors are sometimes also called *characteristic* values and *characteristic* vectors, respectively. To put the above definition in simpler words, eigenvectors have the property that their direction is unchanged by a linear transformation. Such a transformation is represented by a square matrix $\mathbf{A}$ which is applied to a vector $\mathbf{v}$, i.e., the matrix-vector product $\mathbf{A}\mathbf{v}$ gives the transformed vector. If only the length of the vector changes, then all vector components are scaled by the same scalar λ and we again arrive at the eigenvalue equation $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. By bringing both terms to the same side of the equation, we get:

$$(\lambda\mathbf{I} - \mathbf{A})\mathbf{v} = \mathbf{0} \quad (\text{A.51})$$

where $\mathbf{I}$ is the identity matrix which has the same shape as the square matrix $\mathbf{A}$. The situation of $\mathbf{v} = \mathbf{0}$ is not considered an eigenvector, and therefore, it must be a vector with nonzero entries. As a consequence, the other factor of the product, i.e., the term in parentheses, must be singular because otherwise it would be possible to bring it to the right-hand side (by inversion); multiplying it with $\mathbf{0}$, however, results in $\mathbf{v} = \mathbf{0}$ which violates the assumption. Thus, $(\lambda\mathbf{I} - \mathbf{A})$ must be singular, i.e., it cannot be inverted. For singular matrices, the determinant must vanish, which is expressed in the so-called *characteristic equation*:

Definition A.19 (Characteristic Equation) For a real-valued, square matrix $\mathbf{A}$, the characteristic equation is:

$$p_{\mathbf{A}}(\lambda) = \det(\lambda\mathbf{I} - \mathbf{A}) = 0 \quad (\text{A.52})$$

for some scalar λ . Then, $p_{\mathbf{A}}(\lambda)$ is the *characteristic polynomial*.

The *characteristic polynomial* $p_{\mathbf{A}}(\lambda)$ is a polynomial with degree of up to $\text{rank}(\mathbf{A})$. The solutions of the characteristic equation, Eq. A.52, are the eigenvalues λ_i , which are also called the *spectrum* of $\mathbf{A}$.

Surprisingly, the characteristic equation is no longer dependent on $\mathbf{v}$! Note that sometimes this equation is also encountered with changed signs as $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ which, however, does not influence subsequent derivations as long as this is done consistently. For solving this equation, we first state two useful properties: If a quadratic $n \times n$ matrix $\mathbf{A}$ has the eigenvalues $\lambda_1, \dots, \lambda_n$, then the trace of $\mathbf{A}$ is the sum of all eigenvalues:

$$\text{tr}(\mathbf{A}) = \lambda_1 + \cdots + \lambda_n = \sum_{i=1}^n \lambda_i . \quad (\text{A.53})$$

The product of all eigenvalues of $\mathbf{A}$ is given by the determinant of $\mathbf{A}$:

$$\det(\mathbf{A}) = \lambda_1 \times \lambda_2 \times \cdots \times \lambda_n = \prod_{i=1}^n \lambda_i \quad (\text{A.54})$$

where “ $\times$ ” is used to indicate the scalar multiplication. Before we state the general solution of this equation we first write it out for a 2×2 matrix:

$$\det(\lambda \mathbf{I} - \mathbf{A}) = \det \left(\begin{bmatrix} \lambda - a_{11} & a_{12} \\ a_{21} & \lambda - a_{22} \end{bmatrix} \right) \quad (\text{A.55})$$

$$= (\lambda - a_{11})(\lambda - a_{22}) - a_{12}a_{21} \quad (\text{A.56})$$

$$= \lambda^2 - \lambda(a_{11} + a_{22}) + (a_{11}a_{22} - a_{12}a_{21}) = 0 \quad (\text{A.57})$$

Hence, the determinant is a polynomial of degree 2 with λ as the variable. If we additionally make use of the definition of the trace and determinant, we obtain the *characteristic polynomial*:

Characteristic polynomial for a 2×2 matrix:

$$\lambda^2 - \lambda \text{tr}(\mathbf{A}) + \det(\mathbf{A}) = 0 . \quad (\text{A.58})$$

Now, the eigenvalues can be easily computed as:

$$\lambda = \frac{\text{tr}(\mathbf{A})}{2} \pm \sqrt{\frac{\text{tr}^2(\mathbf{A})}{4} - \det(\mathbf{A})} . \quad (\text{A.59})$$

Finally, we can also compute the eigenvectors solving Eq. A.50 by insertion of each of the eigenvalues and solving for the corresponding eigenvector $\mathbf{v}$.

Computing the characteristic polynomial and the resulting eigenvalues and eigenvectors for larger matrices is quickly getting quite involved. For many ML applications and dataset, the involved matrices are rather large, and therefore, the use of numerical algorithms is recommended. However, the main idea still stays the same, and in particular the structure of the equation—a λ^n term in the beginning followed by new terms, the trace, and determinant terms at the end—stays roughly the same.

Python implementation: We conclude this section with a Python example that the reader can also use as an exercise to solve manually. For computing eigenvalues and eigenvectors we use the numpy function `eig` from the module `linalg`:

```
In [1]: import numpy
A = numpy.array([[2, 3],
                 [3, -6]], dtype=float)
eig_values, eig_vectors = numpy.linalg.eig(A)
lambda1, lambda2 = eig_values
lambda1, lambda2
```

```
Out [1]: array([ 3., -7.])
```

As a test we now insert the first eigenvalue and eigenvector into the eigenvalue equation Eq. A.51 and show that evaluating the right-hand side results in a vector with zero entries:

```
In [2]: lambda1, lambda2 = eig_values
v1, v2 = eig_vectors[:, 0], eig_vectors[:, 1] # two column vectors
(lambda1 * numpy.eye(2) - A) @ v1
```

```
Out [2]: array([0.00000000e+00, 2.22044605e-16])
```

The result is a vector where both entries are approximately zero since the value of $\approx 2.2 \cdot 10^{-16}$ can, within the range of the machine accuracy, be considered as zero.

A.3.2 Diagonal Matrices and Diagonalization of Matrices

Diagonal matrices play an important role in many applications of linear algebra such as *diagonalization* and have a special relationship with eigenvalues and eigenvectors: in particular, for diagonal matrices the eigenvectors are already known.

Definition A.20 (Diagonal Matrix) A diagonal matrix is a matrix in which all entries that are not located on the main diagonal are zero. For the elements on the main diagonal there are no restrictions and some of them also can be zero.

To construct a diagonal matrix we use the operator `diag` which is defined as:

$$\text{diag}(\lambda_1, \dots, \lambda_n) := \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_n \end{bmatrix} \quad \text{with } \lambda_i \in \mathbb{R} . \quad (\text{A.60})$$

Some of the diagonal elements also can be zero, e.g., the following matrix is also a diagonal matrix:

$$\mathbf{A} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -2 \end{bmatrix}. \quad (\text{A.61})$$

In this book we only consider square diagonal matrices. Thus, all diagonal matrices are symmetric matrices with $\mathbf{A}^T = \mathbf{A}$.

Diagonalization of a matrix $\mathbf{A}$ is the process of finding an alternative representation of a square matrix $\mathbf{A}$; this representation consists of an invertible matrix $\mathbf{P}$ and a diagonal matrix $\mathbf{D}$. Based on this, we define a *diagonalizable matrix* as follows:

Definition A.21 (Diagonalizable Matrix) A square matrix $\mathbf{A}$ is diagonalizable if there exists a diagonal matrix $\mathbf{D}$ and an invertible matrix $\mathbf{P}$ such that $\mathbf{A}$ has the representation:

$$\mathbf{D} = \mathbf{P}^{-1} \mathbf{A} \mathbf{P}. \quad (\text{A.62})$$

An alternative formulation to Eq. A.62 is the factorization of $\mathbf{A}$ into a diagonal matrix $\mathbf{D}$ and a matrices $\mathbf{P}$ whose columns are the eigenvectors of $\mathbf{A}$ —the so-called *eigendecomposition* $\mathbf{A}$:

$$\mathbf{A} = \mathbf{P} \mathbf{D} \mathbf{P}^{-1}. \quad (\text{A.63})$$

If $\mathbf{A}$ has full rank, then all eigenvectors are orthogonal. If the eigenvectors are additionally scaled to unit length,² $\mathbf{P}$ is an orthogonal matrix, which implies that $\mathbf{P}^{-1} = \mathbf{P}^T$; $\mathbf{P}$ represents different types of transformations. $\mathbf{P}$ is also called *modal matrix*, and $\mathbf{D}$ is also called *spectral matrix* as it has all eigenvalues as diagonal elements.

Diagonalizable matrices act as to anisotropically scale the space, i.e., the coordinate system spanned by the eigenvectors is scaled by the corresponding eigenvalues. The intimate relationship between eigenvectors/eigenvalues and diagonal matrices also shows in the fact that diagonalization is identical to finding eigenvalues and eigenvectors. This is summarized in Algorithm A.1.

A very convenient aspect of diagonalizable matrices is that their inverse can very easily be computed as long as none of their eigenvalues is zero. It is simply:

² Scaling the vectors does not change the result of the eigendecomposition $\mathbf{P} \mathbf{D} \mathbf{P}^{-1}$ as the inversion of $\mathbf{P}$ cancels out the scaling factors from BP.

$$\mathbf{A}^{-1} = \mathbf{P} \mathbf{D}^{-1} \mathbf{P}^T \quad \text{with} \quad \mathbf{D}^{-1} = \text{diag} \left(\frac{1}{\lambda_1}, \dots, \frac{1}{\lambda_n} \right). \quad (\text{A.64})$$

A.3.3 Similarity Transformations

An alternative approach uses the notion of similarity between matrices:

Definition A.22 (Similarity of Matrices) Two square matrices $\mathbf{A}$ and $\mathbf{D}$ are *similar* if there is an invertible matrix $\mathbf{P}$ such that:

$$\mathbf{D} = \mathbf{P}^{-1} \mathbf{A} \mathbf{P}. \quad (\text{A.65})$$

As a consequence, a matrix $\mathbf{A}$ is diagonalizable if it is similar to a diagonal matrix. In case that the two matrices are similar, then they must have the same rank, trace, determinant, and eigenvalues. It is sufficient to show that any of these properties is *not* fulfilled in order to conclude that two matrices are *not* similar. On the other hand, proving that two general matrices are similar is not straightforward. Often, the *Jordan canonical form* of the matrices is used, but this is beyond the scope of this short overview.

A.3.4 The Spectral Theorem

A very useful theorem in linear algebra is the *spectral theorem* which is also sometimes called the *principal axis theorem*. It essentially tells if and how a matrix can be *diagonalized*; compare Algorithm A.1. This is important for many derivation but also for practical purposes as diagonal matrices are easier to handle and the formulations tend to be simpler.

The name of this theorem stems from the fact that the set of all eigenvalues of a matrix is called a *spectrum* (although for some of the readers that only might explain one unusual name with another somewhat unusual name). The theorem is formulated as follows:

Spectral Theorem: Assume that $\mathbf{A}$ is an arbitrary but symmetric matrix. Then $\mathbf{A}$ can be *diagonalized* such that it has only eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_n$ that are mutually orthogonal with:

$$\mathbf{A} \mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \text{with } i = 1, \dots, n \quad (\text{A.66})$$

The $\lambda_i \in \mathbb{R}$ are the eigenvalues.

Algorithm A.1: Summary of the Algorithmic Steps for Matrix Diagonalization

Let us assume for simplicity a 3×3 square matrix $\mathbf{A}$. The general case of a $n \times n$ matrix can be handled analogously. These are the steps required for diagonalization of $\mathbf{A}$:

1. **Determine the eigenvalues λ using the determinant of the eigenvalue equation.** For this, we use Eq. A.52 which reads:

$$\det(\lambda \mathbf{I} - \mathbf{A}) = 0 ,$$

where $\mathbf{I}$ is the identity matrix. The solution of the equation is obtained using the characteristic polynomial and will result in (up to) three eigenvalues λ_i for the 3×3 matrix.

2. **For each eigenvalue λ_i find the corresponding eigenvector $\mathbf{v}_i$.** We use the eigenvalue equation Eq. A.50 for each λ_i :

$$(\lambda_i \mathbf{I} - \mathbf{A})\mathbf{v}_i = \mathbf{0}$$

and use elementary row operations on the term in parentheses. As the system of equations can be an undetermined, we fix one component of $\mathbf{v}_i$ and then determine the others.

3. **“Assemble” the modal matrix.** As a preparation, scale the eigenvectors such that they have unit length, $\mathbf{v}_i := \mathbf{v}_i / \|\mathbf{v}_i\|$. The modal matrix $\mathbf{P}$ consists of the eigenvectors, ordered in columns:

$$\mathbf{P} = [\mathbf{v}_1 \ \mathbf{v}_2 \ \mathbf{v}_3] .$$

$\mathbf{P}$ is now an orthonormal matrix.

4. **Compute the diagonal matrix $\mathbf{D}$.** For this we use Eq. A.62 and the spectral theorem (cf. Appendix A.3.4) to conclude that the eigenvectors are mutually orthogonal, and thus, $\mathbf{P}$ is an orthogonal matrix with $\mathbf{P}^{-1} = \mathbf{P}^T$ which is helpful for computing $\mathbf{D}$:

$$\mathbf{D} = \mathbf{P}^T \mathbf{A} \mathbf{P} .$$

Thus, $\mathbf{D}$ will contain the three eigenvalues $\lambda_1, \lambda_2, \lambda_3$ with the following structure:

$$\mathbf{D} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} ,$$

so that from now on we can directly go from the first step (determining the eigenvalues) to setting up $\mathbf{D}$.

The property of orthogonality implies that the eigenvectors are linearly independent and therefore form a basis of a vector space. From the spectral theorem a number of useful properties and equalities can be derived:

1. For an arbitrary real-valued $m \times n$ matrix $\mathbf{A}$, the $n \times n$ matrix product $\mathbf{A}^T \mathbf{A}$ is symmetric and so is the $m \times m$ product $\mathbf{A} \mathbf{A}^T$. Therefore, the spectral theorem is applicable for both of these matrix products which is an important prerequisite for the following claim.
2. The products $\mathbf{A}^T \mathbf{A}$ and $\mathbf{A} \mathbf{A}^T$ have identical nonzero eigenvalues.
This is extremely helpful in case that $\mathbf{A}$ has a very large number of rows but a considerably smaller number of columns (or vice versa): we are now able to choose whether we use $\mathbf{A} \mathbf{A}^T$ or $\mathbf{A}^T \mathbf{A}$ for computing eigenvectors and eigenvalues!
3. The eigenvalues of $\mathbf{A} \mathbf{A}^T$ and $\mathbf{A}^T \mathbf{A}$ are all larger than or equal to zero.

For brevity, we do not show the proofs here even though they are quite short. The interested reader is referred to any textbook on linear algebra.

As an example for the usefulness of the spectral theorem, imagine a dataset consisting of 10000 records and 3 features. The resulting feature matrix $\mathbf{A}$ has 10000 rows but only 3 columns. If we need to compute the eigenvalues of the product $\mathbf{A} \mathbf{A}^T$, then there is an easy way to do this: compute the 9 eigenvalues of the 3×3 matrix $\mathbf{A}^T \mathbf{A}$. Then all other 9997 eigenvalues of $\mathbf{A} \mathbf{A}^T$ are (by help to the claim in 2.) simply zero!

A.3.5 Eigendecomposition and Singular Value Decomposition

A method closely related to diagonalization which is also used for decomposition of matrices is *singular value decomposition* (**SVD**). There, for a real matrix $\mathbf{A}$ the eigendecomposition has the form of $\mathbf{A} = \mathbf{P} \mathbf{\Sigma} \mathbf{Q}^T$, where $\mathbf{P}$ and $\mathbf{Q}$ are orthogonal matrices that, however, need not be the inverse of each other. The entries of $\mathbf{\Sigma}$ are called the *singular values of $\mathbf{A}$* . Additionally, the diagonal matrix $\mathbf{\Sigma}$ has only real values that are larger or equal to zero, as opposed to $\mathbf{D}$ from the eigendecomposition which may even consist of complex values. Last but not least, the eigendecomposition requires $\mathbf{U}$ to be a square matrix while **SVD** does not have any requirements.

A comment on the use in principal component analysis: Eigendecomposition as well as **SVD** can be used for principal component analysis (compare Sect. 15.2); **SVD** would directly be applied to the centered data matrix while eigendecomposition acts on the covariance matrix. In both cases the resulting eigenvalues can be interpreted as the lengths of the semiaxes of a two-dimensional ellipse or (hyper-) ellipsoid with the eigenvectors defining the directions of the axes. For some implementations, **SVD** can be more memory intensive than the eigendecomposition which is why we focus here on the latter.

A.4 Matrix-Related Operation That Only Exists in Numpy

We conclude this linear algebra overview with a comment on potential pitfalls: There are a few differences between how matrices and vectors are handled in numpy as compared to the mathematical, linear algebra definitions and conventions.

A.4.1 Addition of a Scalar to a Matrix

Adding a scalar to all elements of a matrix can be differently done in numpy, which is mainly related to performance aspects. For example, in numpy the following operation is valid:

```
In [1]: import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], dtype=float)
A + 0.5 # which works equally well as 0.5 + A

Out [1]: np.array([[1.5, 2.5, 3.5],
                  [4.5, 5.5, 6.5],
                  [7.5, 8.5, 9.5]])
```

Mathematically, it is not allowed to sum up two objects with different dimensions. However, in numpy the product is internally “broadcasted” and optimized. In a Jupyter notebook the execution time can be measured using the “magic” function `numpy`

```
In [2]: %timeit A + 0.5

788 ns ± 19.3 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Comparing this with addition of `A` with a matrix with constant values

```
In [3]: %timeit A + np.array([[0.5, 0.5, 0.5], \
                             [0.5, 0.5, 0.5], \
                             [0.5, 0.5, 0.5]])

1.61 µs ± 12.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

shows that this is a much slower operation as now a full matrix needs to be allocated in the memory and then it is element-wise added, whereas in the previous version each element of `A` is simply incremented by 0.5. As a warning, keep in mind that even though Python is very forgiving, you should not get used to writing `A + 0.5` in any mathematical document.

A.4.2 One-Dimensional vs. Two-Dimensional “Vectors”

Furthermore, Python is also quite forgiving concerning the multiplication of vectors and matrices—in some cases, that is. The `numpy.dot(a, b)` performs different types of “multiplications” depending on the dimension of the two inputs. For example, if the second variable is a one-dimensional array, then the function calculates a sum product over the last axis of `a` and `b`. To be on the safe side, it is always advisable not to mix “1D vectors” (an array with dimensions 1) with “2D vectors” ($1 \times n$ or $n \times 1$ matrices), unless you really know what you are doing. Usually, it is advisable to turn all one-dimensional vectors into two-dimensional arrays, e.g., using `numpy.reshape`.

As this is quite a critical subject we repeat the most important steps here. We start by constructing a matrix and then extract a row from that matrix:

```
In [1]: import numpy as np

A = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]], dtype=float)
a = A[1] # second row of A, identical to A[1, :]
a
```

```
Out [1]: array([ 9., 10., 11., 12.])
```

`a` is now a one-dimensional array as can also be seen by taking a look at its shape and dimensions and comparing this to the original matrix `A`:

```
In [2]: print("shape(A) :", A.shape, "    dim(A) :", A.ndim)
        print("shape(a) :", a.shape, "    dim(a) :", a.ndim)

shape(A) : (3, 4)    dim(A) : 2
shape(a) : (4,)     dim(a) : 1
```

Now, we use `reshape` to put the same data into a new shape. This can be imagined as first, “flattening” a matrix, i.e., row by row, to concatenate each row vector until we have a very long vector with all elements of the matrix. Next, these numbers are then—again row by row—used to create a new matrix, this time with possibly different numbers of columns and/or rows. For both steps, we always start at the top left corner of the matrix.

```
In [3]: b = a.reshape(4, 1) # here: identical to a.reshape(-1, 1)
print("shape(b) :", b.shape)
print("dim(b)   :", b.ndim)
print(b)
```

```
shape(b) : (4, 1)
dim(b)    : 2
[[ 9.]
 [10.]
 [11.]
 [12.]]
```

A commonly used way of “reshaping” is to set one dimension to -1 , e.g., `np.reshape(-1, 1)`. This implies that the second dimension is fixed as one, from which the other, remaining dimension (the one with -1) can be inferred. Obviously, the total number of elements cannot change.

Below, a number of chosen proofs and derivations are given, which are not required for the understanding of the main text but which nonetheless can be useful for understanding some relations and theorems¹ in more detail.

B.1 Proofs of the Theorems About Expectation Values

Theorem B.1 (Expected Value of a Product) *Let us assume two independent random variables, X and Y . Then, the expectation of the product of the two random variables is equal to the product of the expectations:*

$$E[XY] = E[X] \cdot E[Y] .$$

Proof To prove this theorem for the case of discrete random variables, we assume that X can take on N values and Y takes on M values. The product XY is again a random variable which can take the values $x_i y_j$ with $i = 1 \dots N$ and $j = 1 \dots M$. If all values are equally likely, then the proof is very simple:

$$E[XY] = \frac{1}{NM} \sum_{i=1}^N \sum_{j=1}^M x_i y_j$$

¹ A *theorem* is a mathematical statement or relation that can be derived or proved.

$$\begin{aligned}
&= \left(\frac{1}{N} \sum_{i=1}^N x_i \right) \left(\frac{1}{M} \sum_{j=1}^M y_j \right) \\
&= E[X] \cdot E[Y] . \quad \square
\end{aligned} \tag{B.1}$$

B.2 Proofs of Some Theorems About Variances

Theorem B.2 (Multiplication with a Scalar) *Assuming that X is a random variable and c a scalar value. Then the following identity holds:*

$$\text{Var}(cX) = c^2 \text{Var}(X)$$

Proof

$$\begin{aligned}
\text{Var}(cX) &= E \left[(cX - cE[X])^2 \right] \\
&= E \left[c^2 (X - E[X])^2 \right]
\end{aligned} \tag{B.2}$$

with the relation $E[cX] = cE[X]$ according to Eq. 7.6 we can take c^2 out of the operator leading to

$$\begin{aligned}
&= c^2 E \left[(X - E[X])^2 \right] \\
&= c^2 \text{Var}(X) . \quad \square
\end{aligned} \tag{B.3}$$

Theorem B.3 (Sum of Two Random Variables) *Assuming that X and Y are two random variables that are independent from each other. Then the variance of their sum is the sum of their variances:*

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$$

Proof

$$\begin{aligned}
\text{Var}(X + Y) &= E \left[((X + Y) - E[X + Y])^2 \right] \\
&= E \left[((X - E[X]) + (Y - E[Y]))^2 \right] \\
&= E \left[(X - E[X])^2 + 2(X - E[X])(Y - E[Y]) + (Y - E[Y])^2 \right]
\end{aligned} \tag{B.4}$$

Because X and Y are independent variables, also $X - E[X]$ and $Y - E[Y]$ are independent variables. Thus, their product is zero. Additionally, the expectation value of a sum is the sum of the expectation values and we get:

$$\begin{aligned}
\text{Var}(X + Y) &= E \left[(X - E[X])^2 \right] + 0 + E \left[(Y - E[Y])^2 \right] \\
&= \text{Var}(X) + \text{Var}(Y) . \quad \square
\end{aligned} \tag{B.5}$$

B.3 Simplification of Pearson Moment Coefficient of Skewness

Pearson moment coefficient of skewness as introduced in Sect. 9.3.6 is given by:

$$\gamma_1 = E \left[\left(\frac{X - \mu}{\sigma} \right)^3 \right] \quad \text{with} \quad \mu = E[X] . \tag{B.6}$$

There, σ is the standard deviation. Expand the terms of the expectation values to arrive at a simpler term that can easily be written down for multiple data x_i (with $i = 1 \dots N$):

$$\gamma_1 = \frac{E[(X - \mu)^3]}{\sigma^3} \tag{B.7}$$

$$= \frac{E[X^3] - 3\mu E[X^2] + 3\mu^2 E[X] - \mu^3}{\sigma^3} \tag{B.8}$$

$$= \frac{E[X^3] - 3\mu (E[X^2] - \mu E[X]) - \mu^3}{\sigma^3} \tag{B.9}$$

$$= \frac{E[X^3] - 3\mu\sigma^2 - \mu^3}{\sigma^3} \tag{B.10}$$

$$= \frac{E[X^3] - 3\mu\sigma^2 - \mu^3}{(\sigma^2)^{\frac{3}{2}}} . \tag{B.11}$$

This can now be easily computed with:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i \quad (\text{B.12})$$

$$\sigma = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2 \quad (\text{B.13})$$

$$\mathbb{E}[X^3] = \frac{1}{N} \sum_{i=1}^N x_i^3. \quad (\text{B.14})$$

B.4 Proofs and Additional Derivations for Distributions

B.4.1 Property of the Binomial Coefficient

We start with the definition of the binomial coefficient which is given as:

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!} \quad (\text{B.15})$$

Next, we introduce one of the most well-known identities that are often used for proofs and derivations in the context of binomial coefficients:

$$k \binom{n}{k} = n \binom{n-1}{k-1}, \quad (\text{B.16})$$

which is a way to link the binomial coefficient of the “n-1/k-1” quantities to the original ones. For our derivation, the point of departure is the expression $\binom{n-1}{k-1}$:

$$\binom{n-1}{k-1} = \frac{(n-1)!}{((n-1)-(k-1))! \cdot (k-1)!} \quad (\text{B.17})$$

$$= \frac{(n-1)!}{(n-k)! \cdot (k-1)!} \quad (\text{B.18})$$

Subsequently, we employ the property of the factorial for which it is $x! = (x-1)! \cdot x$. Insertion of this relation into Eq. B.15 gives:

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!} = \frac{(n-1)! \cdot n}{(n-k)! \cdot (k-1)! \cdot k} \quad (\text{B.19})$$

$$= \frac{n}{k} \cdot \frac{(n-1)!}{(n-k)! \cdot (k-1)!} \quad (\text{B.20})$$

This expression now allows us to rewrite the left-hand side of Eq. B.16 as:

$$k \binom{n}{k} = k \frac{n}{k} \frac{(n-1)!}{(n-k)! \cdot (k-1)!} \quad (\text{B.21})$$

$$= n \frac{(n-1)!}{(n-k)! \cdot (k-1)!} \quad (\text{B.22})$$

As a last step we use:

$$\binom{n-1}{k-1} = \frac{(n-1)!}{(n-k)! \cdot (k-1)!} \quad (\text{B.23})$$

from above and obtain as a final result:

$$k \cdot \binom{n}{k} = n \cdot \binom{n-1}{k-1}. \quad (\text{B.24})$$

B.4.2 Binomial Distribution: Derivation of Mean and Variance

The binomial distribution is, according to Sect. 10.4, given by:

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k}. \quad (\text{B.25})$$

To derive the formulation of the mean we start with the general definition of the (population) mean:

$$\mu = \sum_{i=0}^n x_i p(x_i) \quad (\text{B.26})$$

where the population, for which the mean is calculated, consists of altogether $n+1$ items. We then substitute the *probability mass function* (PMF), Eq. B.25, for $p(x)$:

$$\mu = \sum_{i=0}^n k_i \binom{n}{k_i} p^{k_i} (1-p)^{n-k_i}. \quad (\text{B.27})$$

If we compute the first element of the sum for $i = 0$, we find that it is zero. For all following values it is in general:

$$k \binom{n}{k} = n \binom{n-1}{k-1} \quad (\text{B.28})$$

which then gives:

$$\mu = \sum_{i=1}^n n \binom{n-1}{k_i-1} p^{k_i} (1-p)^{n-k_i} \quad (\text{B.29})$$

$$= np \sum_{i=1}^n \binom{n-1}{k_i-1} p^{k_i-1} (1-p)^{n-k_i} \quad (\text{B.30})$$

where we took np out of the summation as it does not depend on i . The last term $(1-p)^{n-k_i}$ can be replaced by $(1-p)^{n-1} (1-p)^{-(k_i-1)}$. With this, we obtain:

$$\mu = np \sum_{i=1}^n \binom{n-1}{k_i-1} p^{k_i-1} (1-p)^{n-1} (1-p)^{-(k_i-1)}. \quad (\text{B.31})$$

To get rid of all the $n-1$ and $k-1$ terms, we substitute $m := n-1$ and $j_i = k_i-1$:

$$\mu = np \sum_{i=0}^m \binom{m}{j_i} p^{j_i} (1-p)^{m-j_i}. \quad (\text{B.32})$$

where the summands have the same structure as the binomial PMF (except for the variable names). Thus, summing this PMF-expression must result in 1, as it is the probability of all outcomes:

$$\sum_{i=0}^m \binom{m}{j_i} p^{j_i} (1-p)^{m-j_i} = 1. \quad (\text{B.33})$$

and therefore we get for the mean:

$$\mu = np. \quad (\text{B.34})$$

B.4.3 Isocontour Lines of a Bivariate Normal Distribution

Assume a bivariate normal distribution $\mathcal{N}(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2)$ whose *probability density function* (PDF) has the following form:

$$f(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{(x_1-\mu_1)^2}{2\sigma_1^2} - \frac{(x_2-\mu_2)^2}{2\sigma_2^2}\right) \quad (\text{B.35})$$

Then the level set of all points (x_1, x_2) for given shape parameters μ_1, μ_2, σ_1 , and σ_2 that have the constant value $c \in \mathbb{R}$ is given by:

$$f(x_1, x_2; \mu_1, \mu_2, \sigma_1, \sigma_2) \doteq c . \quad (\text{B.36})$$

The question is now, what the geometrical shape of the isocontour lines is. Our intuition points us into the direction of an ellipse, which is in general given in terms of its major and minor semiaxes, a and b , by:

$$\frac{x_1^2}{a^2} + \frac{x_2^2}{b^2} = 1 \quad (\text{B.37})$$

This equation assumes that the ellipse is centered at the origin of the coordinate system. To prove that the contour lines of the bivariate normal distribution have elliptical shape, we start by determining all points (x_1, x_2) that fulfill Eq. B.36 and will bring it into the governing equation for the ellipse, Eq. B.36:

$$c = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{(x_1 - \mu_1)^2}{2\sigma_1^2} - \frac{(x_2 - \mu_2)^2}{2\sigma_2^2}\right) . \quad (\text{B.38})$$

Multiplying both sides with $2\pi\sigma_1\sigma_2$ and applying the $\ln$ function results in:

$$-\frac{(x_1 - \mu_1)^2}{2\sigma_1^2} - \frac{(x_2 - \mu_2)^2}{2\sigma_2^2} = \ln(2\pi c\sigma_1\sigma_2) . \quad (\text{B.39})$$

This can now be simplified by multiplying with -1 and utilizing that $-\ln(x) = \ln(1/x)$:

$$\frac{(x_1 - \mu_1)^2}{2\sigma_1^2} + \frac{(x_2 - \mu_2)^2}{2\sigma_2^2} = \ln\left(\frac{1}{2\pi c\sigma_1\sigma_2}\right) . \quad (\text{B.40})$$

Furthermore, dividing both sides of the equation by the logarithm term gives:

$$\frac{(x_1 - \mu_1)^2}{2\sigma_1^2 \ln\left(\frac{1}{2\pi c\sigma_1\sigma_2}\right)} + \frac{(x_2 - \mu_2)^2}{2\sigma_2^2 \ln\left(\frac{1}{2\pi c\sigma_1\sigma_2}\right)} = 1 . \quad (\text{B.41})$$

This is already quite close to Eq. B.37, and with the definitions of the two semiaxes as:

$$a = \sqrt{2\sigma_1^2 \ln\left(\frac{1}{2\pi c\sigma_1\sigma_2}\right)} \quad \text{and} \quad b = \sqrt{2\sigma_2^2 \ln\left(\frac{1}{2\pi c\sigma_1\sigma_2}\right)} , \quad (\text{B.42})$$

where we assumed that a is the major semiaxis and b is the minor semiaxis (we simply switch them if $a < b$). Insertion of these abbreviations yield:

$$\frac{(x_1 - \mu_1)^2}{a^2} + \frac{(x_2 - \mu_2)^2}{b^2} = 1 , \quad (\text{B.43})$$

which is the equation for an ellipse that is shifted along the two directions by μ_1 and μ_2 . In terms of the distribution, this equation describes the line that indicates a value of the PDF of c .

References

1. C. Hill. *Learning Scientific Programming with Python*, 2nd edition. Cambridge University Press, 2016. ISBN 978-1108745918.
2. Q. Kong, T. Siau, and A. M. Bayen. *Python Programming and Numerical Methods*. Elsevier, 2021. ISBN 978-0-12-819549-9. DOI <https://doi.org/10.1016/c2018-0-04165-1>.

List of Python Listings

9.1	A Python function for reading the file <code>iris.csv</code> . The function returns the feature matrix and the target vector. Note the specification of the <code>dtype</code> when the two <code>np.array</code> s are created. Due to the mixed data type (numbers and strings) in the file <code>iris.csv</code> , we import all of it as strings and then create <code>x</code> as a <code>float</code> and <code>y</code> as a <code>string</code> . Section 4.7 provides more information on how to obtained the file	181
10.1	A minimal example for plotting the PDF and CDF of a normal distribution	219
12.1	Python function for performing the train-test split. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact	286
13.1	A Python function that returns piecewise polynomial approximations at equidistant points. The main difference to regression codes in the previous section is that we use <code>idx</code> to “filter out” all points that are not in the current interval. The resulting data for each interval are stored as a list of tuple of the <code>x</code> and <code>y</code> coordinates	345
13.2	Kernel regression with radial basis functions: the Python function for the RBF and for creating the feature matrix (to be used with the code snippets in the text)	350
14.1	An implementation of the kNN algorithm. In case that there is a tie during voting (here implemented using the <code>argmax</code> function, which returns the index of the element with the maximum value), then only the first class with the highest count contained in <code>count</code> will be used	378
16.1	An implementation of the train-test split	445
17.1	Implementation of the Rosenblatt perceptron for making a prediction. Here, the prediction will be a value of either 0 or 1. However, it is also possible to use, e.g., -1 and $+1$ (see Sect. 17.2.5 for an explanation)	465

17.2	Implementation of the perceptron learning rule. The code reuses the function <code>predict</code> from Python Listing 17.1. Here, the training is stopped if the error is zero. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact	473
17.3	An implementation of the network of two neurons. Even though it is a very simplistic implementation, it is able to handle a number of input records at the same time as can be seen by the fact that <code>x1</code> and <code>x2</code> are both numpy <code>arrays</code>	486
17.4	General implementation of the vectorized equations for a network with one hidden layer. Note that <code>np.where</code> also works with the row vector <code>theta1</code> : it automatically applies it to each row of <code>X1 @ W1</code> so that creating <code>THETA1</code> is not required	492
18.1	Implementation of a network with two input nodes, two nodes in a hidden, layer and a single output node. See the text for more information. Note that contrary to all best practices in programming, we omitted docstrings, comments, type hints, and “sanity checks” to ensure that the code stays compact	507

Index

A

Abaqus, 3
Abstract base class, 517
Accuracy, 258
Activation functions, 459, 498
 linear, 498
 logistic, 500
 ReLU, 500
 sigmoid, 500
 tanh, 501
Activation layer, 512
ADALINE model, 476
Adam, 543
Adaptive data generation, 547
Additive principle, 70
Aether, 6
Aggregation function, 459
AI winter, 481, 497
AlexNet, 541
Anaximenes, 6
Anscombe's quartet, 196
Antecedent, 361
Aquinas, T., 6, 8
Aristotle, 6
Array, 32
Artificial intelligence, 16
Ashby, M., 19
Ashby map, 19
Atomistic simulations, 20
Attribute, 257
Autoencoder, 545, 548
Average, 87, 133
Average pooling, 538

B

Babylonia, 4
Backpropagation, 469, 501, 510, 513
Backward pass, 501, 512
Bandwidth, 351

Bar diagram, 199
Baseline model, 452
Basis for polynomials, 330
Basis functions, 329, 330
Basis vector, 397
Batch normalization, 524
Batch size, 524
Bayes, T., 10
Bayes' theorem, 99
Bayesian information criterion (BIC), 429
Bellman, R.E., 26
Bernoulli trial, 210
Bessel's correction, 190
Bias, 153, 258
Bias-variance tradeoff, 320
Bimodal, 189
Binary classification, 359
Binary cross-entropy loss, 552
Binary segmentation, 262, 546
Bishop, C., 237
Bivariate distributions, 110
Black box model, 42, 239
Boolean decision problem, 461
Bootstrap, 158, 450
Bottleneck blocks, 543
Box-Cox transformation, 251
Box plot, 200
Box-whisker plot, 200
Brackets, 32
Broyden, Fletcher, Goldfarb, and Shanno (BFGS), 543

C

Cahn-Hilliard equation, 59
Candidate model, 441
Categorical encoding, 438
Centered data matrix, 168
Centering, 251
Central limit theorem, 163, 218

- Central tendency, 183, 187
 - Chain rule, 504, 513
 - Characteristic equation, 590
 - Characteristic polynomial, 589–591
 - Cherry picking, 10, 163
 - Class IDs, 367
 - Classification, 238, 253
 - binary, 359, 470
 - Class imbalance, 373, 449
 - Clustering, 253, 424
 - Cnidus, E., 6
 - Code, 548
 - Coefficient of determination, 529
 - Coefficient of momentum, 544
 - Coefficient of variation, 261
 - Collinearity, 334
 - Column vector, 38, 570
 - Combinations, 81, 84
 - without repetition, 81
 - with replacement, 84
 - Combinatorics, 69
 - Concentric half-spheres, 6
 - Conditional, 555
 - Conditional probability, 99, 125
 - Condition number, 334
 - Confounding variable, 175
 - Confusion matrix, 263
 - Consequent, 361
 - Constrained optimization, 402
 - Contingency table, 186
 - Continuous probability distribution, 113
 - Control points, 346
 - Convolution, 534
 - Convolutional layer, 534
 - Convolutional neural network, 533
 - Copernican Revolution, 7
 - Copernicus, N., 7
 - Correlation, 172
 - Correlation coefficient, 10, 529
 - Correlation matrix, 175
 - Correlation of variables, 184
 - Cost function, 293, 298, 502
 - Covariance, 165
 - matrix, 167
 - Cross-entropy, 267
 - Cross tab, 186
 - Cross validation, 441
 - k-fold, 447
 - leave-one-out, 446
 - repeated k-fold, 448
 - stratified k-fold, 449
 - Cumulative distribution function, 115
 - Cumulative frequencies, 112
 - Curse of dimensionality, 9, 26, 256, 321, 390, 429
- D**
- Data, 23
 - augmentation, 544
 - bins, 198
 - categorical, 89
 - cleaning, 245
 - collection, 244
 - compression, 394
 - covariance matrix, 223
 - matrix, 45
 - preparation, 245
 - preprocessing, 245
 - record, 45
 - space, 290
 - structured, 43
 - tabular, 45
 - transformation, 247
 - visualization, 196, 256
 - Datalogy, 11
 - Data matrix, 47, 49
 - Data mining, 11, 21
 - Data science, 15, 18
 - Dataset, 44
 - Cahn-Hilliard, 58
 - element properties, 60
 - handwritten digits, 63
 - iris flower, 62
 - Ising, 57
 - MNIST, 63
 - nanoindentation, 60
 - tensile test, 53
 - DBSCAN, 430
 - de Brahe, T., 8
 - Decision boundary, 254, 359, 363, 367, 468
 - Decoder, 548
 - Deep CNN autoencoders, 550
 - Deep Galerkin method, 563
 - Deep learning, 497
 - Deep neural network, 17
 - DeepONet, 563
 - Denoising, 418, 550
 - Descriptive statistics, 152
 - Design matrix, 49, 325
 - Design of experiment, 28
 - Deviation
 - mean squared, 259
 - Diagonalization, 593

- Dice coefficient, 265
- Digital twin, 26
- DIKW pyramid, 25, 237
- Dilated convolutions, 539
- Dilation rate, 539
- Dimensionality reduction, 253, 391, 408, 548
- Discovery of new materials, 394
- Discriminator, 550
- Disjoint, 71
- Dislocation, 21
- Dispersion, 184, 190
- Distribution
 - Bernoulli, 210
 - bimodal, 202
 - binomial, 211
 - bivariate normal, 127, 219
 - categorical, 109
 - discrete uniform, 208
 - exponential, 229
 - Gaussian, 217, 425
 - geometric, 213
 - logistic, 230
 - lognormal, 227
 - multivariate, 110
 - multivariate normal, 222
 - normal, 217
 - Poisson, 215
 - univariate, 110
- Distribution shape, 184
- Domain knowledge, 17, 438
- Dot product, 571
- Dropout, 527
- E**
- Early exaggeration, 422
- Early stopping, 544
- Eigendecomposition, 408, 593
- Eigenvalue, 589
- Eigenvalue equation, 403
- Eigenvector, 589, 593
- Embedding, 410
- Encoder, 548
- Ensemble methods, 420
- Epsm, 156
- Error, 258
 - extrapolation, 278
 - mean absolute, 259
 - mean squared, 259
 - root mean squared, 260
 - root mean squared logarithmic, 261
 - testing, 284
 - training, 275, 284
- Error measure, 257
- Euclidean distance, 369, 573
- Euclidean norm, 572
- Event, 69, 92, 94
 - impossible, 94
 - independent, 96
- Evidence, 100
- Example, 45
- Expectation operator, 132
- Expectation step, 427
- Expectation value, 132
 - homogeneity, 134
 - linearity, 134
 - scaling, 134
 - translating, 135
- Experiment, 69
- Explanatory variable, 257, 283
- Exploratory data analysis (EDA), 179
- Extended feature matrix, 324
- Extrapolation error, 278
- Extreme outlier, 201
- Extrinsic dimensionality, 390
- F**
- Factorial, 73
- False negative, 262, 360
- False positive, 262, 360
- Feature, 257, 283
 - creation, 438
 - engineering, 19, 411, 437
 - selection, 440
 - transformation, 437
- Feature matrix, 48
 - extended, 336
- Features, 45
- Feed forward step, 469
- Fence, 201
 - inner, 201
 - outer, 201
- Final model, 441
- Fine-tuning, 547
- First central moment, 144
- First principal component, 404
- First raw moment, 142
- Forward pass, 501, 510
- Fourier basis functions, 341, 353
- Fourier series, 353
- Fourth moment, 195
- Frequency distribution table, 110
- Frequency table, 184
- F1 score, 266
- Full rank, 586
- Fully connected layer, 488, 510, 514
- Fully connected network, 487

G

Galton, F., 10, 271
Gauss, C.F., 10
Gaussian mixture model, 425
Gaussian Naive Bayes, 378
Generator, 550
Global minimum, 298
Glorot, X., 525
Goodfellow, I., 550
Gradient descent, 299, 512
 mini-batch, 543
 stochastic, 543
Gram matrix, 327
Greedy maximization, 402
Grid search, 294

H

Hamming distance, 370
Hand-labeling, 546
He, K., 526
Hidden layer, 488
Hinge loss, 385
Histogram, 198
Hoff, T., 476
Holdout method, 283
Hyperparameter, 243, 318, 428
Hyperparameter optimization, 320, 374
Hyperparameter study, 441
Hypothesis, 279, 280
Hypothesis class, 366
Hypothesis space, 280, 289

I

Image reconstruction, 417
Independent events, 97, 122
Independent variable, 42
Independent variables, 283
Inference, 152, 237, 253, 281
Inferential statistics, 152
Information, 23
Input data matrix, 48
Input layer, 488
Input variable, 41, 257
Instance, 43, 45
Instance-based learning, 374
Instance space, 290
Interquartile range, 200
Intersection over union, 264
Interval, 36
Intrinsic dimensionality, 390
Inverted dropout, 528
Ising, E., 57

J

Jaccard similarity, 263
Joint CDF, 123
Joint distribution, 122
Joint PMF, 123
Joint probability, 99, 121
Joint probability density function, 124, 126
Joint probability distribution, 122

K

KDE plot, 202
Kepler, J., 8
Kernel density estimation, 352
Kernel projection, 387
Kernel regression, 348
Kernel smoothing, 349
Kernel trick, 387, 420
k-medians, 424
kNN regression, 355
Knots, 344
Knowledge, 23, 25
Knowledge discovery, 20
Kolmogorov, A., 97
Kolmogorov axioms, 97
Kramer, M.A., 548
Kurtosis, 195

L

Label, 252, 257, 283
Label encoding, 438
Langrangian multiplier, 402
Laplace, P.S., 10
Lasso regression, 354
Latent space, 548
Law of truly large numbers, 163
Lazy learning, 374
Leaky ReLU, 500
Learning
 instance-based, 240
 lazy, 242
 memory-based, 240
 unsupervised, 253
Learning bias, 562
Learning rate, 299
Least square (LSQ) method, 10, 274, 293
Lecun, Y., 540
Legendre, A.-M., 10
Lenet, 540
Libration, 9
Likelihood, 87, 100
Linear function, 317
Linearly separable, 359

- Linear model, 288
- Linear sampling, 156
- List comprehension, 161
- L1 loss, 259
- L2 loss, 259
- L_2 norm, 573
- Local minimum, 298
- Logistic function, 231
- Lookup table, 42
- Loss function, 292, 502
- Loss landscape, 307

- M**
- Machine epsilon, 299
- Machine learning definition, 235
- Macrostates, 74
- Mahalanobis distance, 225
- Majority class, 158
- Making inference, 152
- Manhattan distance, 370
- Manifold learning, 421
- Margin, 385
- Marginalization, 141
- Marginal probability, 99
- Material property charts, 18
- Materials data science, 18
- Materials Genome Initiative, 20
- Matrix, 37, 575
 - addition, 576
 - covariance, 223
 - determinant, 587
 - diagonal, 592
 - diagonalization, 592
 - Hadamard product, 580
 - inverse, 583
 - lower triangular, 585
 - modal, 593
 - neural element, 582
 - orthogonal, 588
 - product, 577
 - rank, 586
 - scalar multiplication, 577
 - spectral, 593
 - square, 582
 - symmetric, 584
 - trace, 585
 - transposition, 581
 - triangular, 585
 - upper triangular, 585
 - vector multiplication, 579
- Maximization step, 427
- Maximum likelihood estimation, 426
- Maximum margin estimator, 385
- Max pooling, 538
- Mayer, J.T., 9
- McCulloch, W., 459
- Mcculloch-Pitts model, 459
- Mean, 133, 142
 - arithmetic, 187
 - geometric, 187
 - harmonic, 187
- Mean error, 259
- Median skewness, 194
- Metadata, 3
- Method of least squares, 10
- Metric, 323
- Microscopy, 21
- Microstates, 74
- Mild outlier, 201
- Min-batch, 524
- Minimization, 298
- Minkowski distance, 370
- Min-max GAN loss, 552
- Min/max scaler, 251
- Minority class, 158
- Mitchell, T.T., 237
- MNIST database, 63
- Modal value, 189
- Mode, 189
- Mode collapse, 554, 562
- Model, 5, 238, 239, 254
- Model class, 275
- Model parameter, 254, 275
- Model selection, 428, 429
- Moment
 - central, 142
 - first raw, 142
 - standardized, 145
 - zero-th raw, 141
- Momentum, 544
- Monomials, 329
- Most general hypothesis, 368
- Most specific hypothesis, 368
- Moving averages, 348
- Multicollinearity, 354
- Müller, A.C., 237
- Multiclass classification, 360
- Multicollinearity, 334
- Multidimensional scaling, 421
- Multilayer perceptron, 487, 501
- Multimodal, 189
- Multiple linear regression, 323, 335
- Multiple linear regression models, 310
- Multiple polynomial regression, 339
- Multiplication principle, 71, 79
- Multivariate distribution, 185
- Mutually exclusive, 94

N

Naive Bayes methods, 378
 Nash equilibrium, 551
 Natural splines, 346
 Naur, P., 11
 Negative example, 363
 Network
 architecture, 510
 Neuron
 biological, 457
 model, 457
 multipolar, 458
 Non-linear function, 317
 Nonlinear scaling, 250
 Non-parametric dimensionality reduction, 421
 Non-parametric methods, 342
 Normal distribution, 425
 Normalization, 250, 261
 Normalized Xavier initialization, 525
 Normal matrix, 327

O

Object detection, 264
 Objective function, 294, 298
 Observation, 89
 Observational bias, 562
 Occam, W., 7, 8
 Occam's razor, 7, 8, 429
 One-hot encoding, 251
 Operator learning, 563
 Optimal hyperplane, 385
 Ordinary least squares, 327
 Orthogonal projection, 573
 Orthogonal vector projection, 573
 Orthogonal vectors, 572
 Outcome, 92, 93
 Outlier, 188, 201, 246
 global, 246
 point, 246
 Output data matrix, 49
 Output layer, 488
 Output variable, 257
 Overfitting, 6, 282, 320, 351
 Oversampling, 158

P

Padding, 537
 Parameterized ReLU, 500
 Parameter space, 280, 289
 Parametric models, 322
 Parametric regression, 342
 Parentheses, 32
 Pattern, 237

Pearson, K., 10, 271
 Pearson's measure of kurtosis, 195
 Pearson's mode skewness, 194
 Pearson's 2nd coefficient of skewness, 194
 Percentile, 188, 200
 Perceptron, 463
 Perceptron learning rule, 469
 Permutation, 74
 Permutation feature importance, 439
 Permutations, 79, 80
 Perplexity, 421
 Physical units, 3
 Physics-informed neural network (PINN), 563
 Physics informed machine learning, 562
 Piecewise polynomial function, 344
 Piecewise regression splines, 346
 Piecewise step function, 345
 Pitts, W., 459
 p-norm, 370
 Poisson point process, 216
 Poisson process, 216
 Polynomial interpolation, 277
 Polynomial regression, 318, 336
 Pooling, 538
 Population, 151
 Position vector, 570
 Positive example, 363
 Posterior probability, 100
 Precision, 258, 266
 Prediction, 253, 281
 Predictors, 283
 Preprocessing, 245
 Principal
 direction, 404
 stress, 392
 stress components, 393
 Principal component analysis (PCA), 391
 application, 430
 kernel, 420
 randomized, 420
 Principle of parsimony, 8
 Prior probability, 100
 Probabilistic model, 378
 Probability, 86
 density function, 108, 109
 function, 108
 joint, 97
 mass function, 108, 109
 table, 111
 Probability theory, 87
 Problem of vanishing gradients, 542, 555
 Product rule, 99
 ProGAN, 555
 Ptolemy, C., 6

Q

Quartile, 200

R

Radial basis function, 349

Random experiment, 88

Random noise, 278

Random variable, 86, 103

 continuous, 106

 discrete, 105

 image, 106

 image of a, 105

 scaling, 134

Range, 184

Range scaling, 251

Rare events, 163, 188, 246

Raw moment, 140

Real numbers, 32

Recall, 266

Regressand, 257

Regression, 238, 253, 271

 multiple linear, 287

 simple linear, 287

Regression plane, 311

Regression splines, 346

Regression to the mean, 162

Regressor, 257

Regularization, 354

Reinforcement learning, 253

Relative cumulative frequency, 112

Research data management, 20

Residual, 258

 block, 542

 connection, 542

 error, 291

Resnet, 541

Response variable, 257

Retrograde motion, 4, 5

Ridge regression, 354

Rosenblatt perceptron, 463

Row matrix, 325

Row vector, 38

Rummelhart, D.E., 497

Runge, C.D., 346

S

Same padding, 538

Sample, 151

 representative, 153

Sample mean, 187

Sample space, 92, 93

Sampling, 69, 151, 153, 154

 over, 544

 simple random, 155

 stratified, 159

 systematic, 156

 take from top, 158

 weighted, 158

Samuel, A., 236

Scalar field, 575

Scalar product, 571

Scatter plot, 197

Scatter plot matrix, 203

Scientific machine learning, 562

Scientific Python Stack, 182

Score value, 404

Scree plot, 415, 430

Seaborn, 203

Second principal component, 407

Segmentation, 264

Selection bias, 153

Semantic segmentation, 545

Semi-parametric method, 343

Set, 32

 difference, 34

 disjoint, 34

 empty, 34

 intersection, 34

 operation, 32, 33

 union, 34

Shallow neural network, 17

Shrinkage penalty, 354

Simple linear regression, 335

Singular value decomposition, 596

Skewness, 193

Skip connection, 542

Spectral theorem, 594

Spectrum, 590, 594

Spinodal decomposition, 58

Spurious correlation, 174

Squashing functions, 498

Standard deviation, 10, 139

Standardization, 250

Statistical

 error, 291

 inference, 152, 278

Statistical sampling, 440

Statistics, 149

Steepest descent, 299

Step function, 498

Stirling's approximation, 73

Stochastic regularization, 527

Strata, 159

Stratification, 159

Stratified k-fold cross-validation, 449

Stratified sampling, 449

Stride, 538

Structured dataset, 43

StyleGAN, 555

Summary statistics, 183

Supervised learning, 22, 252

Support vector, 385

Support vector regression, 355

Systematic sampling, 156

T

Take from top, 158

Tanzi, E., 459

Target, 283

Target variable, 252, 257

Testing cost, 294

Testing dataset, 441

t-distributed stochastic neighbor embedding
(t-SNE), 421, 430

Threshold logical unit (TLU), 459

Total covariance, 409

Training cost function, 294

Training data, 239

Train/test split, 446

Transfer learning, 544

Transformers, 556

True function, 278

True negative, 262

True positives, 262

Tukey, J.W., 11

Tuple, 32, 35

U

Uncertainty, 87

Underfitting, 282, 319

Undersampling, 158

U-Net, 545

Uniform, 189

Uniform manifold approximation and
projection (UMAP), 21

Unimodal, 189

Unit vector, 573

Univariate distribution, 184

Unsupervised learning, 389

V

Validation dataset, 441

Valid padding, 538

Vandermonde matrix, 337, 341

Vanishing gradients, 542

Variability, 258

Variable, 257

Variance, 137

Variance explained, 416

Vector, 37, 570

 component, 571

 field, 574

 product, 571

 space, 570

 transposition, 571

Vectorization, 302

Venn diagram, 34

Visualization, 196

Voronoi tessellation, 372

W

Waldeyer, H.W., 457, 459

Wasserstein distance, 555

Wasserstein GAN, 555

Wavelet regression, 354

Weighted kNN, 373

Weight initialization, 525

Weight space, 289

Weight update, 501

Werbos, P.J., 497

Whisker, 201

White box model, 240

Widrow, B., 476

Wisdom, 25

Wu, C.F.J., 11

X

Xavier initialization, 525

XOR problem, 462, 467, 468, 481, 484

Z

Zahavi, J., 11