DATA SCIENCE SERIES

MODELS DEMYSTIFIED

A Practical Guide from Linear Regression to Deep Learning

Michael Clark and Seth Berry

A **Chapman & Hall** Book

CRC Press
Taylor & Francis Group

Models Demystified

Unlock the Power of Data Science and Machine Learning

In this comprehensive guide, we delve into the world of data science, machine learning, and AI modeling, providing readers with a robust foundation and practical skills to tackle real-world problems. From basic modeling techniques to advanced machine learning algorithms, this book covers a wide range of topics, ensuring that readers at all levels can benefit from its content. Each chapter is meticulously crafted to offer clear explanations, hands-on examples, and code snippets in both Python and R, making complex concepts accessible and actionable. Additional focus is placed on model interpretation and estimation, common data issues, modeling pitfalls to avoid, and best practices for modeling in general.

Michael Clark is a senior machine learning scientist for OneSix, and in prior stints, was a data science consultant at the University of Michigan and Notre Dame. His models have been used in production across a variety of industries, and can be seen in dozens of publications across several academic disciplines. He has a passion for helping people of all skill levels learn difficult stuff.

Seth Berry is the Academic Co-Director of the Master of Science in Business Analytics (MSBA) Residential Program, and Associate Teaching Professor at the University of Notre Dame for the IT, Analytics, and Operations Department. He has a PhD in Applied Experimental Psychology, and has been teaching and consulting in data science for over a decade.

CHAPMAN & HALL/CRC DATA SCIENCE SERIES

Reflecting the interdisciplinary nature of the field, this book series brings together researchers, practitioners, and instructors from statistics, computer science, machine learning, and analytics. The series will publish cutting-edge research, industry applications, and textbooks in data science.

The inclusion of concrete examples, applications, and methods is highly encouraged. The scope of the series includes titles in the areas of machine learning, pattern recognition, predictive analytics, business analytics, Big Data, visualization, programming, software, learning analytics, data wrangling, interactive graphics, and reproducible research.

Recently Published Titles

Data Science for Sensory and Consumer Scientists
Thierry Worch, Julien Delarue, Vanessa Rios De Souza and John Ennis

Data Science in Practice
Tom Alby

Introduction to NFL Analytics with R
Bradley J. Congelio

Soccer Analytics
An Introduction Using R
Clive Beggs

Spatial Statistics for Data Science
Theory and Practice with R
Paula Moraga

Research Software Engineering
A Guide to the Open Source Ecosystem
Matthias Bannert

The Data Preparation Journey
Finding Your Way With R
Martin Hugh Monkman

Getting (more out of) Graphics
Practice and Principles of Data Visualisation
Antony Unwin

Introduction to Data Science
Data Wrangling and Visualization with R Second Edition
Rafael A. Irizarry

Data Science
A First Introduction with Python
Tiffany Timbers, Trevor Campbell, Melissa Lee, Joel Ostblom and Lindsey Heagy

Mathematical Engineering of Deep Learning
Benoit Liquet, Sarat Moka, and Yoni Nazarathy

Introduction to Classifier Performance Analysis with R
Sutaip L.C. Saw

For more information about this series, please visit: https://www.routledge.com/Chapman--Hall-CRC-Data-Science-Series/book-series/CHDSS

Models Demystified
A Practical Guide from Linear Regression to Deep Learning

Michael Clark and Seth Berry

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Cover designed by Freepik

First edition published 2026
by CRC Press
2385 NW Executive Center Drive, Suite 320, Boca Raton FL 33431

and by CRC Press
4 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

CRC Press is an imprint of Taylor & Francis Group, LLC

Library of Congress Cataloging-in-Publication Data
Names: Clark, Michael (Machine learning scientist) author | Berry, Seth author
Title: Models demystified : a practical guide from linear regression to deep learning / Michael Clark and Seth Berry.
Description: First edition. | Boca Raton, FL : CRC Press, 2026. | Series: Chapman & Hall/CRC data science series | Includes bibliographical references. | Summary: “In this comprehensive guide, we delve into the world of data science, machine learning, and AI modeling, providing readers with a robust foundation and practical skills to tackle real-world problems. From basic modeling techniques to advanced machine learning algorithms, this book covers a wide range of topics, ensuring that readers at all levels can benefit from its content. Each chapter is meticulously crafted to offer clear explanations, hands-on examples, and code snippets in both Python and R, making complex concepts accessible and actionable. Additional focus is placed on model interpretation and estimation, common data issues, modeling pitfalls to avoid, and best practices for modeling in general”-- Provided by publisher.
Identifiers: LCCN 2025005941 (print) | LCCN 2025005942 (ebook) | ISBN 9781032582580 hardback | ISBN 9781032587882 paperback | ISBN 9781003451501 ebook
Subjects: LCSH: Data mining--Statistical methods | Electronic data processing--Structured techniques | Statistics--Data processing
Classification: LCC QA76.9.D343 C4735 2026 (print) | LCC QA76.9.D343 (ebook) | DDC 006.3/12--dc23/eng/20250404
LC record available at https://lccn.loc.gov/2025005941
LC ebook record available at https://lccn.loc.gov/2025005942

ISBN: 978-1-032-58258-0 (hbk)
ISBN: 978-1-032-58788-2 (pbk)
ISBN: 978-1-003-45150-1 (ebk)

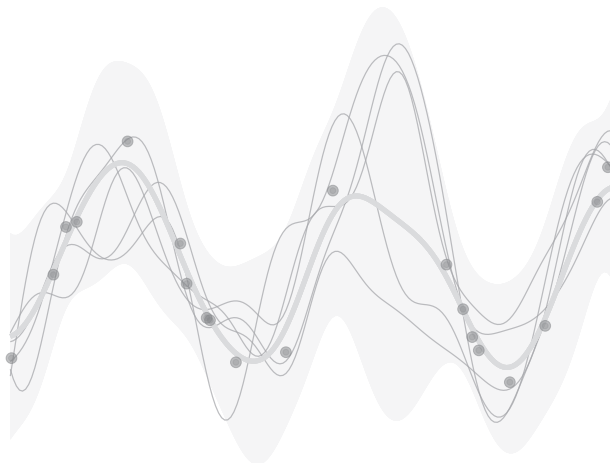
DOI: 10.1201/9781003451501

Typeset in Nimbus Roman font
by KnowledgeWorks Global Ltd.

Table of Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface



Hello and welcome! This book is your companion to exploring the realm of modeling in data science. It is designed to provide you with something useful whether you're a beginner looking to learn some fundamentals, or an experienced practitioner seeking a fresh perspective. Our goal is to equip you with a better understanding of how models work and how to use them, including both basic and more advanced techniques, where we touch on everything from linear regression to deep learning. We'll also show how different models relate to one another to better empower you to successfully apply them in your own data-driven projects. We aim to provide an overview on how to use both machine learning and traditional statistical modeling in a practical fashion, with a balanced emphasis on interpretability and predictive power. Join us on this exciting journey as we explore the world of models in data science!

What Will You Get Out of This Book?

We're hoping for a couple things for you as you read through this book. In particular, if you're starting your journey into data science, we hope you'll leave with:

- A firm understanding of modeling basics from a practical perspective
- A toolset of models and related ideas that you can instantly apply for competent modeling
- A balanced treatment of statistical and machine learning approaches

If you're already familiar with modeling, we hope you'll leave with:

- Additional context for the models you already know
- Some introduction to models you don't know
- Additional understanding of how to choose the right model for the job and what to focus on

For anyone reading this book, we especially hope you get a sense of the commonalities between different models and a good sense of how they work. If you happen to be reading this book in print, you can find the book in web form at https://m-clark.github.io/book-of-models. There you'll also find all the code, figures, and other content that you can interact with more easily, as well as the most up-to-date content, fixes, etc. The web version will be updated with some regularity and have additional content as well.

Brief Prerequisites

You'll definitely want to have some familiarity with R or Python (both are used for examples), and some very basic knowledge of statistics will be helpful. We'll try to explain things as we go, but we won't be able to cover everything. If you're looking for a good introduction to R, we recommend R for Data Science or the Python for Data Analysis book for Python. Beyond that, we'll try to provide the context you need so that you can be comfortable trying things out.

Data and Code

All the data and code used in this book is available on the book's GitHub repository. See the data descriptions in the data section for more information on each of the datasets used. In addition, notebooks with chapter code are also available there (if applicable). For contributions, please see the contributing page for more information. Thanks for reading!

About the Authors

Michael Clark is a Senior Machine Learning Scientist for OneSix. Prior to industry he honed his chops in academia, earning a PhD in Experimental Psychology before turning to data science full-time as a consultant. His models have been used in production across a variety of industries, and can be seen in dozens of publications across several academic disciplines. He has a passion for helping others learn difficult stuff, and he has taught a variety of data science courses and workshops for people of all skill levels in many different contexts.

He also maintains a blog covering many aspects of statistical and machine learning modeling, and has several posts and long-form documents on a variety of data science topics there. He lives in Ann Arbor, Michigan with his wife and his dog, where they all enjoy long walks around the neighborhood. During the course of writing this book, he became a father to Juni, and he is now learning the joys of sleep deprivation.

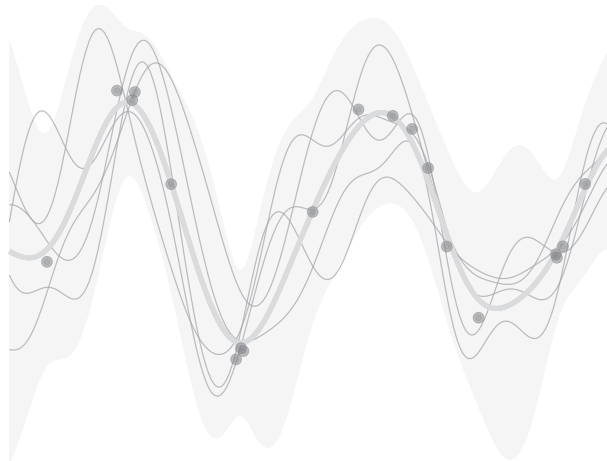
Seth Berry is the Academic Co-Director of the Master of Science in Business Analytics (MSBA) and Associate Teaching Professor at the University of Notre Dame for the IT, Analytics, and Operations Department. He likewise has a PhD in Applied Experimental Psychology and has been teaching and consulting in data science for over a decade. He is an excellent instructor of several data science courses at the undergraduate and graduate level.

He lives in the South Bend area of Indiana with his wife and three kids, and he spends his free time lifting more weights than he should, playing guitar, and chopping wood.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

1

Introduction



We are constantly inundated with data, regardless of our background and whether we're conscious of it or not. It's inescapable, from our first attempts to understand the world around us, to our most recent efforts to explain why we still don't get it. Even now, our most complicated and successful models are almost uninterpretable even to those who created them. But that doesn't mean that even in difficult circumstances we can't understand the essence of how models work, and make practical decisions from their results. And if you're reading this, you are probably the type of person who wants to keep trying anyway! So for seasoned professionals or perhaps just the data curious, we want to help you learn more about how to use data to answer the questions you have.

1.1 What Is This Book?

This book aims to demystify the complex world of data science modeling. It serves as a practical resource and is something you can refer to for a quick

overview of a specific modeling technique, a reminder of some modeling-related topic you've seen before, or perhaps a sneak peak into some modeling details.

The text is focused on a few statistical and machine learning concepts that are ubiquitous, and modeling approaches that are widely employed, and especially those which form the basis for most other models in use in a variety of domains. Believe it or not, whether a lowly t-test or a complex neural network, there is a tie that binds, and you don't have to know every detail to get a solid model that works well enough. We hope to help you understand some of the core modeling principles, and how the simpler models can be extended and applied to a wide variety of data scenarios. We also touch on some topics related to the modeling process, such as common data issues and causal inference.

Our approach is first and foremost a practical one - models are just tools to help us reach a goal, and if a model doesn't work in the world, it's not very useful. But modeling is often a delicate balance of interpretation and prediction, and each data situation is unique in some way, almost always requiring a bespoke approach. What works well in one setting may be poor in another, and what may be the state of the art may only be marginally better than a simpler approach that is more easily interpreted. In addition, complexities arise even in an otherwise deceptively simple application. However, if you have a core understanding of the techniques that lie at the heart of many models, you'll automatically have many more tools at your disposal to tackle the problems you face, and be more comfortable with choosing the best for your needs.

This book also strives to find the balance between statistical texts that don't speak to predictive power or machine learning techniques, and machine learning treatments that consider the job done after calling the predict method. We aim to provide a solid treatment of both, and show how both are necessary perspectives of data modeling. The right modeling tool for your job may come from anywhere, and we hope you'll get a good sense of what's out there, and how to use it.

1.1.1 What we hope you take away

Here are a few things we hope you'll take away from this book:

- A sense of the common thread that runs through the modeling landscape, from simple linear models to complex neural networks
- A small set of modeling tools that will nonetheless be applicable to many common data problems you'll encounter
- Enough understanding to be able to confidently apply these tools to your own data

While we recommend working through the chapters in order if you're starting out, we hope that this book can serve as a "choose your own adventure"

reference. Whether you want a surface-level understanding or a deeper dive, we think you will find value in this book.

1.1.2 What you can expect

For each topic that we cover in a chapter, you will generally see the same type of content structure. We start with an overview and provide some key ideas to keep in mind as we go through the chapter. You'll also be given a sense of the context required. This should help you choose any topic you feel comfortable with, and skip over those you don't.

Models are implemented with code using standard approaches, though results are usually shown in a more digestible format with tables and visualizations. To further demystify the modeling process, at various points we take a DIY approach to show *how* a model or some aspect of it comes about by estimating the results by hand for comparison. We'll also provide some concluding thoughts, connections to other techniques and topics, and suggestions on what to explore next. For some chapters, we'll also provide suggestions for things to try on your own.

Some topics may get a bit more into the weeds than you want, and that's okay! We hope that you can take away the big ideas and come back to the details when you're ready. Just having an awareness of what's possible is often the first step to understanding how to apply it to your own data. In general though, we'll touch a little bit on a lot of things, but hopefully not in an overwhelming way.

1.1.3 What you can't expect

This book will not teach you programming, but you really only need a very basic understanding of R or Python. We also won't be teaching you basic statistics, so we won't be delving into hypothesis testing or the intricacies of statistical theory. The text is more focused on applied modeling, prediction, and performance than a normal stats book, and it is more focused on interpretation and uncertainty in the modeling process than a typical machine learning book. It's not an academic treatment of the topics, so when it comes to references, you'll be more likely to find a nice blog post or YouTube video that clearly demonstrates a concept, rather than a dense academic paper. That said, you should have a great idea of where to go and what to search to go further for deeper content.

1.2 Who Should Use This Book?

This book is intended for every type of *data dabbler*, no matter what part of the data world you call home. If you consider yourself a data scientist, a machine learning engineer, a business analyst, or a deep learning hobbyist, you already know that the best part of a good dive into data is the modeling. But whatever your data persuasion, models give us the possibility to answer questions, make predictions, and understand what we're interested in a little bit better. And no matter who you are, it isn't always easy to understand *how the models work*. Even when you do get a good grasp of a modeling approach, things can still get complicated, and there are a lot of details to keep track of. In other cases, maybe you just have other things going on in your life and have forgotten a few things. In that case, we find that it's always good to remind yourself of the basics! So if you're just interested in data and hoping to understand it a little better, then it's likely you'll find something useful.

1.3 Which Language?

You've probably noticed most data science books, blogs, and courses choose R or Python. While many individuals often have a strong opinion toward teaching and using one over the other, we eschew dogmatic approaches and language flame wars. R and Python are both great languages for modeling and both flawed in unique ways. Even if you specialize in one, it's good to have awareness of the other, as they are the most popular languages for statistical modeling and machine learning, and both excel in at least some areas the other does not. We use both extensively in our own work for teaching, personal use, and production level code, and either may be useful to whatever task you have in mind.

Throughout this book, we will be presenting demonstrations in both R and Python, and you can use both or take your pick, but we want to leave that choice up to you. Our goal is to use them as a tool to help understand some big model ideas. We do present the initial code in R for statistical models, and Python for machine learning approaches and beyond, as we feel their relative strengths are in those areas, and for a balanced focus. But either language can be used well for any modeling task in this book.

In the end, this book can be a resource for the R user who could use a little help translating their R knowledge to Python. We'd also like it to be a

resource for the Python user who sees the value in R's statistical modeling abilities and more. You'll find that our coding style/presentation bends more toward legibility, clarity and consistency, which is not necessarily the same as a standard like PEP8 or the tidyverse style guide[1]. We hope that you can take the code we provide and make it your own, and that you can use it to help you understand the models we're discussing.

1.4 Moving Toward an Excellent Adventure

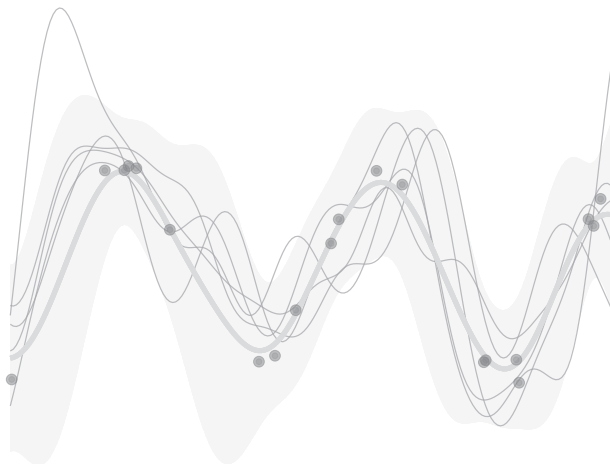
Remember the point we made about "choosing your own adventure"? Modeling and programming in data science is an adventure, even if you never leave your desk! Every situation calls for choices to be made, and every choice you make will lead you down a different path. You will run into errors, dead-ends, and you might even find that you've spent considerable time to conclude that nothing interesting is happening in your data. This, no doubt, is actually part of the fun, and all of those struggles will make your ultimate success that much sweeter. Like every adventure, things might not be immediately clear, and you might find yourself in perilous situations! If you find that something isn't making sense upon your first read, that's fine! Your humble authors have spent considerable time mulling over models and foggy ideas during our assorted (mis)adventures, and nobody should expect to master complex concepts on a single read through! In any arena where you strive to develop skills, distributed practice and repetition are essential. When concepts get tough, step away from the book, and come back with a fresh mind. We have great faith you will get where you want to go, and we're here to help you along the way!

[1]The commonly used coding styles for both R and Python aren't actually scientifically derived or tested, and only recently has research been conducted in this area (see Ivanova et al. (2020) for an example). The guidelines are generally good but mostly reflect the preferences of the person(s) who wrote them. Our focus here is not on programming though.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

2

Thinking About Models



Before we get into the details of models and how they work, let's think more about what we mean when talking about them. As we'll see, there are different ways we can express models and ultimately use them, so let's start by understanding what a model is and what it can do for us.

2.1 What Is a Model?

At its core, a model is just an **idea**. It's a way of thinking about the world, about how things work, how things change over time, how they are different from each other, and how they are similar. The underlying thread is that **a model expresses relationships** about various aspects of the world around us. One can also think of a **model as a tool**, one that allows us to take in information, derive some meaning from it, and act on it in some way. Just like other ideas and tools, models have consequences in the real world, and they can be used wisely or foolishly.

2.2 What Goes into a Model? What Comes Out?

In the context of a model, how we specify the nature of the relationship between various entities depends on the context. In the interest of generality, we'll refer to the **target** as what we want to explain, and **features** as those aspects of the data we will use to explain it. Because people come at data from a variety of contexts, they often use different terminology to mean the same or similar things. The next table shows some of the common terms used to refer to features and targets. Note that they can be mixed and matched, for example, someone might refer to covariates and a response, or inputs and a label.

Table 2.1: Common Terms for Features and Targets

Feature	Target
independent variable	dependent variable
predictor variable	response
explanatory variable	outcome
covariate	label
x	y
input	output
right-hand side	left-hand side

Some of these terms actually suggest a particular type of relationship (e.g., a causal relationship, an experimental setting), but here we'll typically avoid those terms if we can, since those connotations may not apply to most situations. In the end though, you may find us using any of these words to describe the relationships of interest so that you are comfortable with the terminology, but typically we'll stick with features and targets for the most part. In our opinion, these terms have the least hidden assumptions/implications and just imply 'features of the data' and the 'target' we're trying to explain or predict[1].

2.3 Expressing Relationships

As noted, a model is a way of expressing a relationship between a set of features and a target, and one way of thinking about this is in terms of **inputs** and

[1]Just a side note, some refer to the observed target as the 'true' values. All data is measured with error, or simply just varies, so you won't be dealing with 'true' values, but merely *observed* values.

outputs. A model takes in inputs and spits out an output that we hope is similar to the target. But how can we go from input to output?

Well, first off, we assume that the features and target are **correlated**, that there is some relationship between the feature `x` and target `y`. The output of a model will correspond to the target if they are correlated, and more closely match it with stronger correlation. If so, then we can ultimately use the features to **predict** the target. In the simplest setting, a correlation implies a relationship where x and y typically move up and down together (positive correlation) or they move in opposite directions where x goes up and y goes down (negative correlation). But it can also get more complicated than that (Figure 2.1, bottom-right).

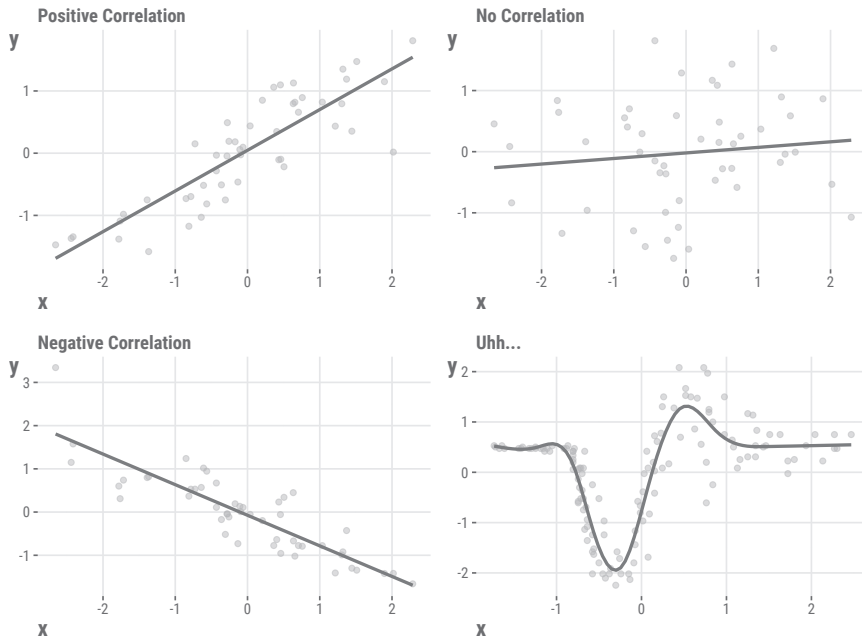


Figure 2.1: Correlation.

Even with multiple features, or nonlinear feature-target relationships, where things are more difficult to interpret, we can stick to this general notion of correlation, or simply **association**, to help us understand how the features account for the target's variability, or why it behaves the way it does.

2.3.1 Mathematical expression of an idea

Models are expressed through a particular language, math, but don't let that worry you if you're not so inclined. As a model is still just an idea at its core, the idea is the most important thing to understand about it. The **math is just a formal way of expressing the idea** in a manner that can be communicated and understood by others in a standard way, and math can help make the idea precise. Here is a generic model formula expressed in math:

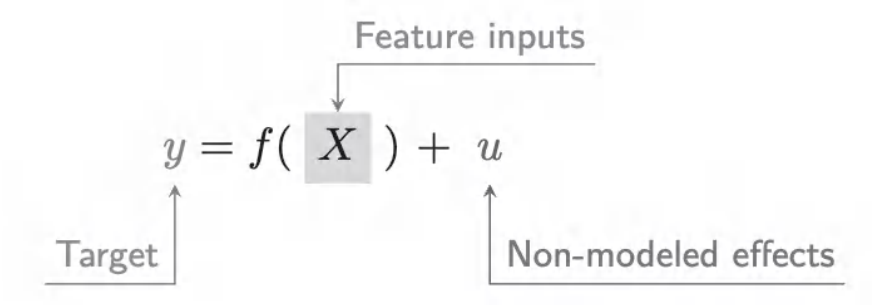


Figure 2.2: Generic model.

In words, this equation says we are trying to explain something y, as a function $f()$ of other things X. The output of our model is $f(X)$, but there is typically some aspect we can't explain u that is also at play. This depiction is the basic form of a model used in data science, and it's essentially the same for linear regression, logistic regression, and even random forests and neural networks.

But in simpler terms, we're just trying to understand everyday things, like how the amount of sleep relates to cognitive functioning, how the weather affects the number of people who visit a park, how much money to spend on advertising to increase sales, how to detect fraud, and so on. Any of these could form the basis of a model, as they stem from scientifically testable ideas, and they all express relationships between things we are interested in, possibly even with an implication of causal relations.

2.3.2 Expressing models visually

Often it is useful to express models visually, as it can help us understand the relationships more easily. For example, we already showed how to express the relationship between a single feature and target in Figure 2.1. A more formal way is with a graphical model, and the following is a generic representation of a **linear model**.

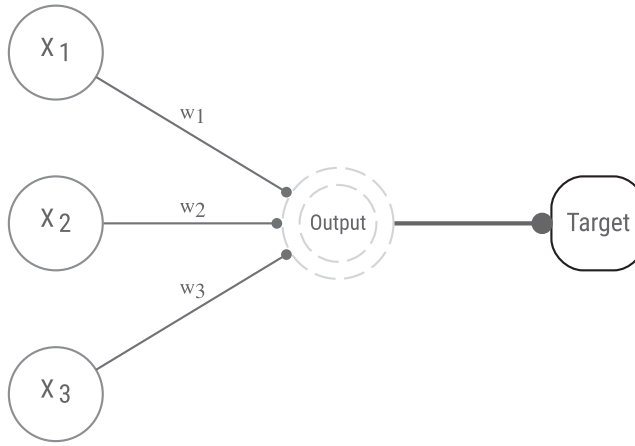


Figure 2.3: Linear model.

This makes clear there is an output from the model that is created from the inputs (X). The 'w' values are weights, which can be different for each input, and the output is the combination of these weighted inputs. As we'll see later, we'll want to find a way to create the best correspondence between the outputs of the model and the target, which is the essence of **model fitting**.

2.3.3 Expressing models in code

Applying models to data can be simple. For example, if you wanted to create a linear model to understand the relationship between sleep and cognitive functioning, you might express it in code as follows.

R

```
lm(cognitive_functioning ~ sleep, data = df)
```

Python

```
from statsmodels.formula.api import ols

model = ols('cognitive_functioning ~ sleep', data = df).fit()
```

The first part with the ~ is the model formula, which is how math comes into play to help us express relationships. Beyond that we just specify where, for example, the observed values for cognitive functioning and the amount of sleep are to be located. In this case, they are found in the same dataframe called `df`,

which may have been imported from a spreadsheet somewhere. Very easy isn't it? But that's all it takes to express a straightforward idea. More conceptually, we're saying that cognitive functioning is a linear function of sleep. You can probably already guess why R's function is `lm`, and you'll eventually also learn why statsmodels function is `ols`, but for now just know that both are doing the same thing.

2.3.4 Models as implementations

In practice, models are implemented in a variety of ways, and the previous code is just one way to express a model. For example, the linear model can be expressed in a variety of ways depending on the tool used, such as a simple linear regression, a penalized regression, or a mixed model. When we think of models as a specific implementation, we are thinking of something like `glm` or `lmer` in R, or `LinearRegression` or `XGBoostClassifier` in Python, or the architecture of a deep neural network. In our examples, we use functions where we will specify the formula that expresses the feature target relationships, or we will specify the input features and target in some fashion, e.g., as separate data objects called `X` and `y`. Afterward, or in conjunction with this specification, we will fit the model to the data, which is the process of finding the best way to map the feature inputs to the target.

2.4 Components of Modeling

It might help to also think about models, or the process of modeling, as having different aspects or parts. We can break our thinking about models into the following components.

Task

The task can be thought of as the goal of our model, which might be defined as regression, classification, ranking, or next word prediction. It is closely tied to the objective (loss) function, which is a measure of correspondence between the model output and the target we're trying to understand. The objective function provides the model a goal - minimize target-output discrepancy or maximize similarity. As an example, if our target is numeric and our task is 'regression', we can use mean squared error as an objective function, which provides a measure of the prediction-target discrepancy.

Model

In data science, a model generally refers to a unique (mathematical) implementation we're using to answer our questions. It specifies the **architecture** of the model, and as we will see, this might be a simple linear component, a series

of trees, or a neural network. In addition, the model specifies the **functional form**, the $f()$ in our equation, that translates inputs to outputs, and the **parameters** required to make that transformation. In code, the model is implemented with functions such as `lm` in R, or in Python, an `XGBoostClassifier` or PyTorch `nn.Model` class.

Algorithm

Various algorithms allow us to estimate the parameters of the model, typically in an iterative fashion, moving from one guess to a hopefully better one. We can think of general approaches, like maximum likelihood, Bayesian estimation, or stochastic gradient descent. Or we can focus on a specific implementation of these, such as penalized likelihood, Hamilton Monte Carlo, or backpropagation.

So when we think about models, we start with an idea, but in the end it needs to be expressed in a form that suggests an architecture. That architecture specifies how we take in data and make outputs in the form of predictions, or something that can be transformed to them. With that in place, we need an algorithm to search the parameter space of the model, and a way to evaluate how well the model is doing. While this is enough to produce results, it only gets us the bare minimum.

We will see demonstrations of all of these components throughout the book, and how they work together to produce results. Beyond these components, there are many more things we have to do to prepare the data for modeling, help us interpret those results, understand the model's performance, and get a sense of its limitations.

2.5 Some Clarifications

You will sometimes see models referred to as a specific statistic, a particular aspect of the model, or an algorithm. This is often a source of confusion for those who are early on in their data science journey, because the terms don't really refer to what the model represents. For example, a t-test is a statistical result, not a model in and of itself. Similarly, some refer to 'logit model' or 'probit model', but these are *link functions* used in fitting what is in fact the same model, which we'll cover in detail later. A 'classifier' tells you the *task* of the model, but not what the model is. Ordinary Least Squares (OLS) is an estimation technique used for many types of models, not just another name for linear regression. Machine learning can potentially be used to fit *any* model and is not a specific collection of models.

All this is to say that it's good to be clear about the model, and to try to keep it distinguished from specific aspects or implementations of it. Sometimes the

nomenclature can't help getting a little fuzzy, and that's okay. Again though, at the core of a model is the idea that specifies the relationship between the features and target.

2.6 Key Steps in Modeling

When it comes to modeling, there are a few key steps that you should always keep in mind. These are not necessarily exhaustive, but we feel they're a good way to think about how to approach modeling in data science.

Define the problem

Start by clearly defining the problem you want to solve. It is often easy to express in very general terms, but it is more challenging to precisely pin down the problem statement in a way that can actually help you solve it. What are you trying to predict? What data do you have to work with? What are the constraints on your data and model? What are the consequences of the results, whatever they may be? Why do you even care about any of this? These are all questions you should try to answer before diving into modeling.

Know your data well

During our time consulting in industry and academia, we've seen many cases where the available data is simply not suited to answer the question at hand[2]. This leads to wasted time, money, and other resources. You can't possibly answer your question if the data doesn't have the appropriate content to do so.

In addition, if your data is fraught with issues due to inadequate exploration, cleaning, or transformation, then you're going to have a hard time getting valuable results. It is very common to be dealing with data that has issues that even those who collected it are unaware of, so always look out for ways to improve it.

Have multiple models at your disposal

Go into a modeling project with a couple models in mind that you think might be useful. This could even be as simple as increasing complexity within a single model approach – you don't have to get too fancy! You should have a few models that you're comfortable with and that you know how to use, and for

[2]This is a common problem where data is often collected for one purpose and then used for another, as with general purpose surveys or administrative data. Sometimes it can be that the available data is simply not enough to say anything without a lot of uncertainty, as in the case of demographic data regarding minority groups, for which there may be few instances of a particular sample. Deep learning approaches like zero/Few-shot learning isn't applicable here, because there isn't a model pretrained on millions or billions of similar examples to transfer knowledge from.

which you know the strengths and weaknesses. Whenever possible, make time to explore more complex or less familiar approaches that you also think may be suitable to the problem. As we'll demonstrate, model comparison can help you have more confidence in the results of the model that's finally chosen. Just like in a lot of other situations, you don't want to 'put all your eggs in one basket', and you'll always have more to talk about and consider if you have multiple models to work with.

Communicate your results

If you don't know the model and underlying data well enough to explain the results to others, you're not going to be able to use them effectively in the first place. Conversely, you also may know the technical side very well, but if you're unable to communicate the results in simpler terms that others can understand, you're going to have a hard time convincing others of the value of your work. Communication is an essential component of the modeling process, and it's something that you should be thinking about from the very beginning.

2.7 The Hard Part

Modeling is just one aspect of the data science process, and the hard part of that process is often not so much the model itself, but everything else that goes into it and what you do with it after. It can be difficult to come up with the original idea for a model, and even harder to get it to work in practice.

The Data

Model performance is largely going to come from the quality of the data and how you've prepared it, from ensuring its integrity to feature engineering. Some models will usually work better than others in certain situations, but there are no guarantees, and often the practical difference in performance is minimal. But you can potentially improve performance by understanding your data better, and by understanding the limitations of your model. Having more domain knowledge can help reduce noise and irrelevant information that you might have otherwise retained, and it can provide insights for feature engineering. Thorough data exploration can reveal bugs and issues to be fixed and will help you understand the relationships between your features and your target.

The Interpretation

Once you have a model, you need to understand what it's telling you. This can be as simple as looking at the coefficients of a linear regression, or as complex as trying to understand the output of a hidden layer in a neural network. Once you get past a linear regression though, you need to *expect* model interpretation

to get hard. But whatever model you use, you need to be able to explain what the model is doing, and how you're ultimately coming to your conclusions. This can be difficult and often requires a lot of work. Even if you've used a model often, it may still be difficult to understand in a new data environment. Model interpretation can take a lot of effort, but it's important to do what's necessary to trust your model results, and help others trust them as well.

What You Do With It

Once you have the model and you (think you) understand it, you need to be able to use it effectively. If you've gone to this sort of trouble, you must have had a good reason for undertaking what can be a very difficult task. We use models to make business decisions, inform policy, understand the world around us, and make our lives better. However, using a model effectively means understanding its limitations, as well as the practical, ethical, scientific, and other *consequences* of the decisions you make based on it. It's at this point that the true value of your model is realized.

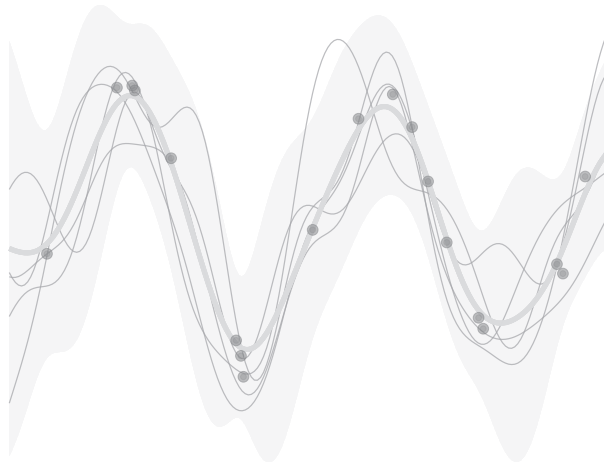
In the end, models are a tool to *help* you solve a problem. They do not solve the problem for you, and they do not absolve you of the responsibility of understanding the problem and the consequences of your decisions.

2.8 Getting Ready for More

The goal of this book is to help you understand models in a practical way that makes clear the relationships we're trying to understand with them, and also how models produce those results we're so interested in. We'll be using a variety of models to help you understand the relationships between features and targets, and how to use models to make predictions, and how to interpret the results. We'll also show you how the models are estimated, how to evaluate them, and how to choose the right one for the job. We hope you'll come away with a better understanding of how models work, and how to use them in your own projects. So let's get started!

3

The Foundation



Now it's time to dive into some modeling! We'll start things off by covering the building block of all modeling, and a solid understanding here will provide you the basis for just about anything that comes after, no matter how complex it gets. The **linear model** is our starting point. At first glance, it may seem like a very simple model, and it is, relatively speaking. But it's also quite powerful and flexible, able to take in different types of inputs, handle nonlinear relationships, temporal and spatial relations, clustering, and more. Linear models have a long history, with even the formal and scientific idea behind correlation and linear regression being well over a century old[1]! And in that time, the linear model is far and away the most used model out there. But before we start talking about the *linear* model, we need to talk about what a **model** is in general.

[1]Regression in general is typically attributed to Galton, and correlation to Pearson, whose coefficient bearing his name is still the most widely used measure of association. Peirce & Bowditch were actually ahead of both (Rovine and Anderson 2004), but Bravais beat all of them.

3.1 Key Ideas

To get us started, we can provide a few concepts key to understanding linear models. We'll cover each of these as we go along.

- The **linear model** is a foundation on which one can build an understanding for practically any other model.
- It combines the strength of its inputs - features - to predict a target.
- Prediction is fundamental to assessing and using a model.
- There are many ways to interpret a model, at the feature level and as a whole.
- It is rather easy to start building upon and adding complexity to a linear model.
- All models come with assumptions, and it's good to be aware of these.
- The things you learn here will be useful in many other contexts - such as other types of models, for classification tasks, and more!

As we go along, be sure that you feel you have the 'gist' of what we're talking about. Almost everything that goes beyond linear models builds on what's introduced here, so it's important to have a firm grasp before climbing to new heights.

3.1.1 Why this matters

The basic linear model and how it comes about underpins so many other models, from the simplest t-test to the most complex neural network. It provides a powerful foundation, and it is a model that you'll see in many different contexts. It's also a model that is relatively easy to understand, so it's a great place to start!

3.1.2 Helpful context

We're just starting out here, but we're kind of assuming you've had some exposure to the idea of statistical or other models, even if only from an interpretation standpoint or visualizations of various relationships. We assume you have an understanding of basic stats like central tendency (e.g., a mean or median), variance, correlation, and stuff like that. And if you intend to get into the code examples, you'll need a basic familiarity with Python or R.

3.2 The Linear Model

The linear model is perhaps the simplest *functional* model we can use to express a relationship between the features that serve as inputs, and the targets we hope to explain with them. And because of that, it's possibly still the most common model used in practice, and it is the basis for many types of other models. So why don't we do one now?

The following dataset has individual movie reviews containing the movie rating (1-5 stars scale), along with features pertaining to the review (e.g., word count), those that regard the reviewer (e.g., age) and features about the movie (e.g., genre, release year).

For our first linear model, we'll keep things simple. Let's predict the rating based on the word count of the review using a specific type of linear model called **linear regression** which, historically speaking, is probably the most common model ever used! We'll use the `lm()` function in R and the `ols()` function in Python[2] to fit the model. Both functions take a formula as the first argument, which, as we noted elsewhere (Section 2.3.3), is just a way of expressing the relationship between the features and target. The formula is displayed as `y ~ x1 + x2 + ...`, where `y` is the target name and `x*` are the feature names. We also need to specify what the data object is, typically a dataframe, where the features and target are found.

R

```
# all data found on github repo
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

model_lr_rating = lm(rating ~ word_count, data = df_reviews)

summary(model_lr_rating)
```

```
Call:
lm(formula = rating ~ word_count, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max
-2.0648 -0.3502  0.0206  0.3352  1.8498

Coefficients:
```

[2]We use the `smf.ols` approach because it is modeled on the R approach.

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.49164    0.04236    82.4   <2e-16 ***
word_count  -0.04268    0.00369   -11.6   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.591 on 998 degrees of freedom
Multiple R-squared:  0.118, Adjusted R-squared:  0.118
F-statistic:  134 on 1 and 998 DF,  p-value: <2e-16
```

Python

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

# all data found on github repo
df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

model_lr_rating = smf.ols('rating ~ word_count', data = df_reviews).fit()

model_lr_rating.summary(slim = True)
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                 rating   R-squared:                       0.118
Model:                            OLS   Adj. R-squared:                  0.118
No. Observations:                1000   F-statistic:                     134.1
Covariance Type:            nonrobust   Prob (F-statistic):           3.47e-29
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      3.4916      0.042     82.431      0.000       3.409       3.575
word_count    -0.0427      0.004    -11.580      0.000      -0.050      -0.035
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
"""
```

For such a simple model as this linear regression is, we certainly have a lot to unpack here! Don't worry, you'll eventually come to know what it all means. But it's nice to know how easy it is to get the results! For now we can just say that there's a *negative* relationship between the word count and the rating (the -0.043), which means that we expect lower ratings with longer reviews. The

output also tells us that the value regarding the relationship is statistically significant (`P(>|t|)` value is $< .05$).

Getting more into the details, we'll start with the fact that the linear model posits a **linear combination** of the features. This is an important concept to understand, but really, a linear combination is just a sum of the features, each of which has been multiplied by some specific value. That value is often called a **coefficient**, or possibly **weight**, depending on the context, and will allow different features to have different contributions to the result. Those contributions reflect the amount and direction of the feature-target relationship. The linear model is expressed as (math incoming!):

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n \tag{3.1}$$

- where y is the target.
- $x_1, x_2, ...x_n$ are the features.
- b_0 is the intercept, which is kind of like a baseline value or offset. If we had no features at all, it would just be the mean of the target.
- $b_1, b_2, ...b_n$ are the coefficients or weights for each feature.

But let's start with something simpler. Let's say you want to take a sum of several features. In math you would write it as:

$$x_1 + x_2 + ... + x_n$$

In this equation, x is the feature and n is the number identifier for the features, so x_1 is the first feature (e.g., word count), x_2 the second (e.g., movie release year), and so on. x is an arbitrary designation - you could use any letter, symbol you want, or even better would be the actual feature name. Now look at the linear model.

$$y = x_1 + x_2 + ... + x_n$$

In this case, the function is *just a sum*, something so simple we do it all the time. In the linear model sense though, we're actually saying a bit more. Another way to understand that equation is that *y is a function of x*. We don't show any coefficients here, i.e., the bs in our initial equation (Equation 3.1), but technically it's as if each coefficient was equal to a value of 1. In other words, for this simple linear *model*, we're saying that each feature contributes in an identical fashion to the target.

In practice, features will never contribute in the same ways, because they correlate with the target differently or are on different scales. So if we want to relate some features, x_1 and x_2, to target y, we probably would not assume that they both contribute in the same way. For instance, we might assign more weight to x_1 than x_2, for whatever reason. In the linear model, this is expressed

by multiplying each feature by a different coefficient or weight. So the linear model's primary component is really just a sum of the features multiplied by their coefficients, i.e., a *weighted sum*. Each feature's contribution to explaining or accounting for the target is proportional to its coefficient. So if we have a feature x_1 and a coefficient b_1, then the contribution of x_1 to the target is $b_1 \cdot x_1$. If we have a feature x_2 and a coefficient b_2, then the contribution of x_2 to the target is $b_2 \cdot x_2$. And so on. So the linear model is really just a sum of the features multiplied by their respective weights.

For our specific model, here is the mathematical representation:

$$\text{rating} = b_0 + b_1 \cdot \text{word\_count}$$

And with the actual results of our model:

$$\text{rating} = 3.49 + -0.04 \cdot \text{word\_count}$$

Not too complicated, we hope! But let's make sure we see what's going on here just a little bit more.

- Our *idea* is that the length of the review in words is in some way related to the eventual rating given to the movie.
- Our *target* is the movie's rating by a reviewer, and the *feature* is the word count.
- We *map the feature to the target* via the linear model, which provides an initial understanding of how the feature is related to the target. In this case, we start with a baseline of 3.49. This value makes sense only in the case of a rating with no review, and it is what we would guess if the word count was 0. But we know there are reviews for every observation, so it's not very meaningful as is. We'll talk about ways to get a more meaningful intercept later, but for now, that is our starting point. Moving on, if we add a single word to the review, we expect the rating to decrease by -0.04 stars. So if we had a review that was 10 words long, i.e., the mean word count, we would predict a rating of 3.49 + 10*-0.04 = 3.1 stars.

3.2.1 The linear model visualized

Given our single feature model, we can easily plot the relationship between the word count and the rating. Now we can visually see the negative or downward-sloping relationship.

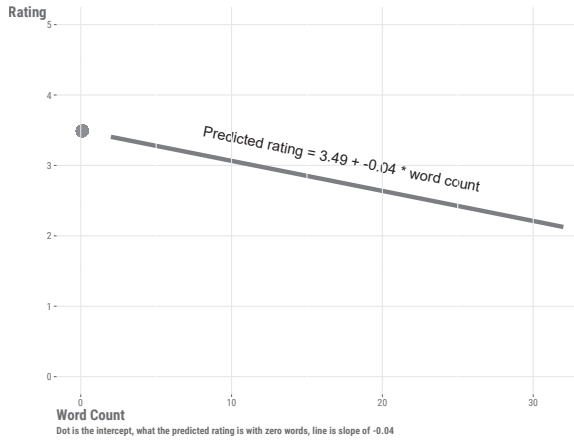


Figure 3.1: Linear regression of rating vs. word count

We can also express the linear model as a *graph* or *graphical model*, which can be a very useful way to think about models in a visual fashion. As we come across other models, a visualization like this can help us see both how different models relate to one another and are actually very similar to one another. In the following example, we have three features predicting a single target, so we have three 'nodes' for the features, and a single node for the target. The feature nodes are combined into a linear combination to produce the **output** of the model. In the context of linear models, the initial combination is often called the **linear predictor**. Each 'edge' signifies the connection of a feature to the combined result and is labeled with the coefficient or weight. The connection between the linear predictor and final model output is direct, without any additional change from the linear predictor stage. This output will ultimately be compared to the observed target value to assess model performance. We'll return to this depiction a little bit later in this chapter (Section 3.7), and other parts as well. But for our standard linear model, we're all set.

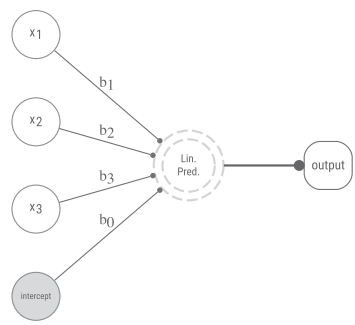


Figure 3.2: Linear regression as a graphical model.

So at this point you have the basics of what a linear model is and how it works, and a couple ways to think about it, whether through programming, math, or just visually. But there is a lot more to it than that. Just getting the model is easy enough, but we need to be able to use it and understand the details better, so we'll get into that now!

3.3 What Do We Do with a Model?

Once we have a working model, there are two primary ways we can use it. One way to use a model is to help us understand the relationships between the features and our outcome of interest. In this way, the focus can be said to be on **explanation**, or interpreting the model results. The other way to use a model is to make estimates about the target for specific observations, often ones we haven't seen in our data. In this case, the focus is on **prediction**. In practice, we often do both, but the focus is usually on one or the other. We'll cover both in detail eventually, but let's start with prediction.

3.3.1 Prediction

A model's utility often lies in its ability to make predictions about the world around us, and this depends fundamentally on the model's ability to predict the target. Once our model has been *fit* to the data, we can obtain predictions by plugging in values for the features that we are interested in, and, using the corresponding weights and other parameters that have been estimated, come to a guess about a specific observation. Let's go back to our results, shown in the following table.

Table 3.1: Linear Model Output

feature	estimate	std_error	statistic	p_value	conf_low	conf_high
intercept	3.49	0.04	82.43	0.00	3.41	3.57
word_count	−0.04	0.00	−11.58	0.00	−0.05	−0.04

Table 3.1 shows the **coefficient** for the feature and the intercept, which is our starting point. In this case, the coefficient for word count is -0.04, which means that for every additional word in the review, the rating goes down by -0.04 stars. So if we had a review that was 10 words long, we would *predict* a rating of 3.49 + **10*-0.04** = 3.1 stars.

When we're talking about the predictions (or outputs) for a linear model, we usually will see this as the following mathematically:

$$\hat{y} = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n \tag{3.2}$$

What is $\hat{y}$? The 'hat' over the y just means that it's a predicted, or 'expected', or estimated value of the model, i.e., the output. This distinguishes it from the target value we actually observe in the data. Our first equations that just used y implicitly suggested that we would get a perfect rating value given the model, but that's not the case. We can only get an estimate. The $\hat{y}$ in a linear regression is also the linear predictor in our graphical version (Figure 3.2), which makes clear it is not the actual target, but the output produced by a combination of the features related to the target.

To make our first equation (Equation 3.1) accurately reflect the relationship between the target and our features, we need to add what is usually referred to as an **error term**, ϵ, to account for the fact that our predictions will not be perfect[3]. So the full linear (regression) model is:

$$y = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n + \epsilon \tag{3.3}$$

The error term is a random variable that represents the difference between the actual value and the predicted value, which comes from the weighted combination of features. We can't know what the error term is, but we can estimate its values, often called **residuals** or just prediction errors, as well as parameters associated with it, just like we can the coefficients. We'll talk more about that in the chapter on estimation (Chapter 6).

Another way to write the model formally is:

$$y|X, \beta, \sigma = \mathrm{N}(\mu, \sigma^2)$$

$$\mu = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n \tag{3.4}$$

or

$$y|X, \beta, \sigma = b_0 + b_1 x_1 + b_2 x_2 + ... + b_n x_n + \epsilon$$

$$\epsilon \sim \mathrm{N}(0, \sigma^2) \tag{3.5}$$

[3]In most circumstances, if you ever have perfect prediction, or even near-perfect prediction, the usual issues are that you have either asked a rather obvious/easy question of your data (e.g., predicting whether an image is of a human or a car), or have accidentally included the target in your features (or a combination of them) in some way.

This makes explicit that the target is assumed to be **conditionally** normally distributed with a mean corresponding to the linear combination of the features, and a variance of σ^2. What do we mean by *conditionally*? This means that, given the features and the estimated model parameters, the target follows a normal distribution (N()). This is the standard assumption for linear regression, and it's a good one to start with, but it's not our only option. We'll talk more about this later in this chapter (Section 3.6), and see what we might do differently in Chapter 8. We will also see that we can estimate the model parameters without any explicit reference to a probability distribution in Chapter 6.

> **i** Predictions by Any Other Name…
>
> You'll often see predictions referred to as **fitted values**, but these imply we are only talking about the observed data features the model was trained on or 'fit' to. Predictions can also be referred to as **expected values**, **estimates**, **outputs**, or **forecasts**, the latter is especially common in time series analysis. Within generalized linear models and others where there may ultimately be a transformation of the output, you may see it referred to as a **linear predictor**.

3.3.2 What kinds of predictions can we get?

What predictions we can get depends on the type of model we are using. For the linear model we have at present, we can get predictions for the target, which is a **continuous variable**. Very commonly, we also can get predictions for a **categorical target**, such as whether the rating is 'good' or 'bad'. This simple breakdown pretty much covers everything, as we typically would be predicting a continuous numeric variable or a categorical variable, or more of them, like multiple continuous variables, or a target with multiple categories, or sequences of categories (e.g., words).

In our case, we can get predictions for the rating, which is a number between 1 and 5. Had our target been a binary good vs. bad rating, our predictions would still be numeric for most models, and usually expressed as a probability between 0 and 1, say, for the 'good' category, or in an initial form that is then transformed to a probability. For example, in the context of predicting a good rating, higher probabilities would mean we'd more likely predict the movie is good, and lower probabilities would mean we'd more likely predict the movie is bad. We then would convert that probability to a class of good or bad depending on a chosen probability cutoff. We'll talk about how to get predictions for categorical targets later[4].

[4]Some models, such as the tree approaches outlined in Section 11.6, can directly predict categorical targets, but we still like, and often prefer using a probability.

We previously saw a prediction for a single observation where the word count was 10 words, but we can also get predictions for multiple observations at once. In fact, we can get predictions for all observations in our data. Besides that, we can also get predictions for observations that we don't even have data for, which is a very neat thing to be able to do! The following shows how we can get predictions for all data, and for a single observation with a word count of 5[5].

R

```
all_predictions = predict(model_lr_rating)

df_prediction = tibble(word_count = 5)
single_prediction = predict(model_lr_rating, newdata = df_prediction)
```

Python

```
all_predictions = model_lr_rating.predict()

df_prediction = pd.DataFrame({'word_count': [5]})
single_prediction = model_lr_rating.predict(df_prediction)
```

Here is a plot of our predictions for the observed data versus the actual ratings[6]. The reference line is where the points would fall if we had perfect prediction. We can see that the predictions are definitely not perfect, but we don't expect this. They are not completely off-base either, in that generally higher predicted scores are associated with higher observed values. We'll talk about how to assess the quality of our predictions later, but we can at least get a sense that we have a correspondence between our predictions and target, which is definitely better than not having a relationship at all!

[5]Some not as familiar with R should be aware that tibbles are a type of dataframe. The name distinguishes them from the standard dataframe, and they have some additional features that make them more user-friendly.

[6]Word count is **discrete**, which means it can only take whole numbers like 3 or 20, and it is our only feature. Because of this, we can only make very limited predicted rating values, while the observed rating can take on many other values. Because of this, the raw plot would show a more banded result with many points overlapping, so we use a technique called **jittering** to move the points around a little bit so we can see them all. The points are still roughly in the same place, but they are moved around a little bit so we can see them all.

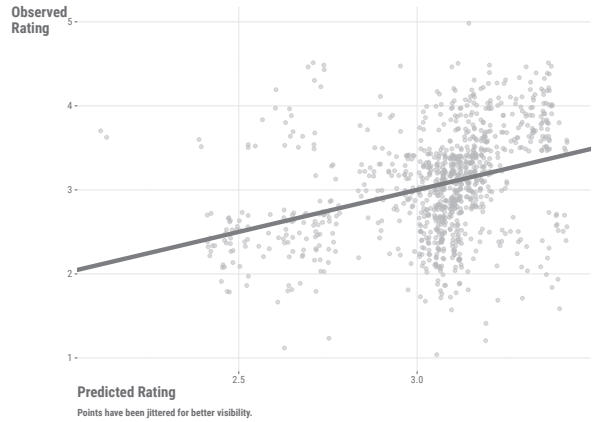


Figure 3.3: Predicted vs. observed ratings.

We saw what our prediction looks like for a single observation, and now we'll add in a few more: one for a review of 10 words, and one for 50 words, which is beyond the length of any review in this dataset, and one for 12.3 words, which isn't even possible for this data, since words are only counted as whole values. To get these values, we just use the same prediction approach as before, and we specify the word count value we want to predict for.

Table 3.2: Predictions for Specific Observations

Word Count	Predicted Rating
5.0	3.3
10.0	3.1
12.3	3.0
50.0	1.4

The values reflect the negative coefficient from our model, showing decreasing ratings with increasing word counts. Furthermore, we see the power of the model's ability to make predictions for what we don't see in the data. Maybe we limited our data review size, but we know there are reviews with 50 or more words out there, and we can still make a guess as to what the rating would be for such a review. Maybe in another case, we know a group of people who have on average reviews of 12.3 words, and we can make a guess as to what the predicted rating would be for that group. Our model doesn't literally know anything about the context of the data, but we can use our knowledge to make predictions that are meaningful to us. This is a very powerful capability, and it's one of the main reasons we use models in the first place.

3.3.3 Prediction error

As we have seen, predictions are not perfect, and an essential part of the modeling endeavor is to better understand these errors and why they occur. In addition, error assessment is the fundamental way in which we assess a model's performance, and, by extension, compare that performance to other models. In general, prediction error is the difference between the actual value and the predicted value or some function of it. In statistical models, it is also often called the **residual**. We can look at these individually, or we can look at them in aggregate with a single metric.

Let's start with looking at the residuals visually. Often the modeling package you use will have this as a default plotting method when doing a standard linear regression, so it's wise to take advantage of it. We plot both the distribution of raw error scores and the cumulative distribution of absolute prediction error. Here we see a couple of things. First, the distribution appears roughly normal, which is a good thing, since statistical linear regression assumes our error is normally distributed, and the prediction error serves as an estimate of that. Second, we see that the mean of the errors is zero, which is a property of linear regression, and the reason we look at other metrics besides a simple 'average error' when assessing model performance. We can also see that our average *absolute* error is around 0.5, most of our predictions (>90%) are within ±1 star rating.

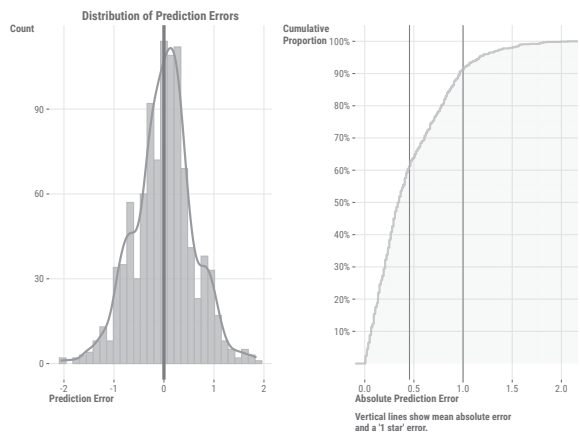


Figure 3.4: Distribution of prediction errors.

Of more practical concern is that we don't see extreme values or clustering, which might indicate a failure on the part of the model to pick up certain segments of the data. It can still be a good idea to look at the extremes just in case we can pick up on some aspect of the data that we could potentially incorporate into the model. So looking at our worst prediction in absolute

terms, we see the observation has a typical word count, and so our simple model will just predict a fairly typical rating. But the actual rating is 1, which is 2.1 away from our prediction, a very noticeable difference. Further data inspection may be required to figure out why this came about, and this is a process you should always be prepared to do when you're working with models.

Table 3.3: Worst Prediction

rating	prediction	word_count
1.0	3.1	10

3.3.4 Prediction uncertainty

We can also look at the uncertainty of our predictions, which is a measure of how much we expect our predictions to vary. Not only are they off from the observed value, but also our predictions themselves are just a guess based on the data we have, and we'd like to know how much we can trust them.

This trust is often expressed as an interval range of values that we expect our prediction to fall within, with a certain level of confidence. But! There are actually two types of intervals we can get. One is really about the mean prediction, or *expected value* we would get from the model at that observation. This is usually called a **confidence interval**. The other type of interval is based on the model's ability to predict new data, and it is typically called a **prediction interval**. This interval is about the actual prediction we would get from the model for any value, whether it was data we had seen before or not. Because of this, the prediction interval is always wider than the confidence interval, and it's the one we usually want to use when we're making predictions about new data.

Here is how we can obtain these from our model.

R

```
prediction_CI = predict(
    model_lr_rating,
    newdata = df_prediction,
    se.fit = TRUE,    # standard error of the fit
    interval = 'confidence'
)

prediction_PI = predict(
    model_lr_rating,
    newdata = df_prediction,
    se.fit = TRUE,
```

```
    interval = 'prediction'
)

pred_intervals = bind_rows(
    as_tibble(prediction_CI$fit),
    as_tibble(prediction_PI$fit),
) |> mutate(
    interval = c('confidence', 'prediction'),
    type = c('mean', 'observation')
)

pred_intervals
```

Python

```
# contains both confidence ('mean_') and prediction ('obs_') intervals
pred_intervals = (
    model_lr_rating
    .get_prediction(df_prediction)
    .summary_frame(alpha = 0.05)
)

pred_intervals
```

Table 3.4: Prediction Intervals for Specific Observations

interval	type	fit	lwr	upr
confidence	mean	3.28	3.23	3.33
prediction	observation	3.28	2.12	4.44

As expected, our prediction interval is wider than our confidence interval, and we can see that the prediction interval is quite wide: from a rating of 2.1 to 4.4. This is a consequence of the fact that we have a lot of uncertainty in our predictions for new observations, and we can't expect to get a very precise prediction from our model with only one feature. This is a common issue with many models, and one that having a better model can help remedy. We can also plot these intervals across a range of values to get a better sense of what they look like. Let's do so for all observed word counts.

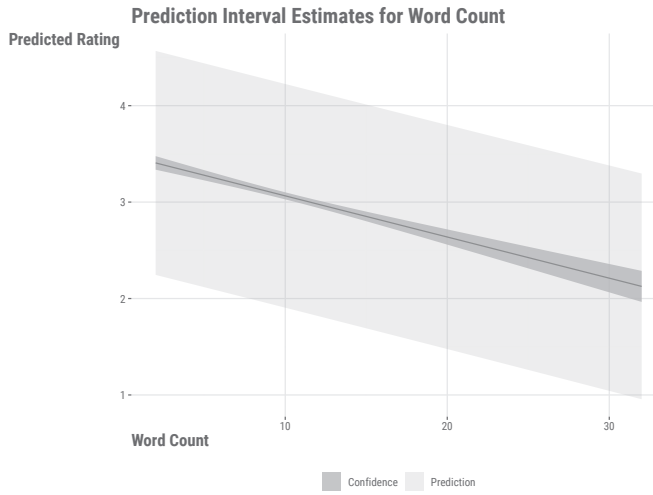


Figure 3.5: Prediction and confidence intervals compared.

Once you move past simpler linear models and generalized linear models, obtaining uncertainty estimates for predictions is difficult, and tools to do so can be scarce. This is especially the case for models used in machine learning contexts, and relatively rare for deep learning approaches. In practice, you can use bootstrapping (Section 7.5) to get a sense of the uncertainty, but this is often not a good estimate in many data scenarios and can be computationally expensive. Bayesian approaches (Section 7.6) can also provide estimates of uncertainty, but likewise are computationally expensive, and require a good deal of expertise to implement for more complex settings. Quantile regression (Section 9.5) can sometimes be appropriate to estimate predictions at different quantiles that can serve as a proxy for prediction intervals, but tools to do so for various models are uncommon. On the other hand, **conformal prediction** tools are becoming more popular, and they can provide a more reliable estimate of prediction uncertainty for any type of model. Yet they too are computationally expensive for more accurate estimates, and good tools are only recently becoming available[7].

So at this point you have the gist of prediction, prediction error, and uncertainty in a prediction, but there is still more to modeling! We'll come back to global assessments of model error very shortly, and even more detail can be found in Chapter 4 where we dive deeper into our models and how they work, and Chapter 6, where we see how to estimate the parameters of our model by picking those that will reduce the prediction error the most. For now though,

[7]For a good intro to conformal prediction, see Angelopoulos and Bates (2022). The mapie package is a good tool for Python, and the tidymodels family has recently added this functionality via the probably package. Michael Clark has a blog post on this as well.

let's move on to the other main use of models, **explanation**, where the focus is on understanding the relationships between the features and the target.

3.4 How Do We Interpret the Model?

When it comes to interpreting the results of our model, there are a lot of tools at our disposal, though many of the tools we can ultimately use will depend on the specifics of the model we have employed. In general though, we can group our approach to understanding results at the **feature level** and the **model level**. A feature-level understanding regards the relationship between a single feature and the target. Beyond that, we also attempt comparisons of feature contributions to prediction, i.e., relative importance. Model-level interpretation is focused on assessments of how well the model 'fits' the data, or more generally, predictive performance. We'll start with the feature level, and then move on to the model level.

3.4.1 Feature-level interpretation

As mentioned, at the feature level, we are primarily concerned with the relationship between a single feature and the target, for whatever features are of interest. More specifically, we are interested in the direction and magnitude of the relationship, but in general, it all boils down to how a feature induces change in the target. For numeric features, we are curious about the change in the target given some amount of change in the feature values. It's conceptually the same for categorical features, but often we like to express the change in terms of group mean differences or something similar, since the order of categories is not usually meaningful. An important aspect of feature-level interpretation is the specific predictions we can get by holding the data at key feature values.

Let's start with the basics by looking again at our coefficient table from the model output.

Table 3.5: Linear Regression Coefficients

feature	estimate	std_error	statistic	p_value	conf_low	conf_high
intercept	3.49	0.04	82.43	0.00	3.41	3.57
word_count	−0.04	0.00	−11.58	0.00	−0.05	−0.04

Here, the main thing to look at is both the actual feature coefficient values and the direction of their relationship, positive or negative. The coefficient for word count is -0.04, and this means that for every additional word in the

review, the rating goes down by -0.04. This interpretation gives us directional information, but how can we interpret the magnitude of the coefficient?

Let's try and use some context to help us. While a drop of -0.04 might not mean much to us in terms of ratings, we might not be as sure about a change in one word for a review. However, we do know the standard deviation of the rating score, i.e., how much it moves around naturally on its own, is 0.63. So the coefficient is about 6% of the standard deviation of the target. In other words, the addition of a single word to a review results in an expected decrease of 6% of what the review would normally bounce around in value. We might not consider this large, but also, a single word change isn't much. What would be a significant change in word count? Let's consider the standard deviation of the feature. In this case, it's 5.1 for word count. So if we increase the word count by one standard deviation, we expect the rating to decrease by -0.04 * 5.1 = -0.2. That decrease then translates to a change of -0.2/0.63 = -0.32 standard deviation units of the target. Without additional context, many would think that's a significant change[8], or at the very least, that the coefficient is not negligible, and that the feature is indeed related to the target. But we can also see that the coefficient is not so large that it's not believable in this context.

> **i** Standardized Coefficients
>
> The calculation we just did results in what's often called a **standardized** or 'normalized' coefficient. In the case of the simplest model with only one feature like this, it is identical to the Pearson r correlation metric, which we invite you to check and confirm on your own. In the case of multiple features, it represents a (partial) correlation between the target and the feature, after adjusting for the other features. But before you start thinking of it as a measure of *importance*, it is not. It provides some measure of the feature-target linear relationship, but that does not entail *practical* importance, nor is it useful in the presence of nonlinear relationships, interactions, and a host of other interesting things that are typical to data and models.

After assessing the coefficients, next up in our table is the **standard error**. The standard error is a measure of how much the coefficient varies from sample to sample. If we collected the data multiple times, even under practically identical circumstances, we wouldn't get the same value each time. The value would bounce around a bit, and the standard error is an estimate of how much it would bounce around. In other words, the standard error is a measure of

[8]Historically, people cite Cohen (2009) for effect size guidelines for simple models, but such guidelines are notoriously problematic. Rely on your own knowledge of the data, provide reasons for your conclusions, and let others draw their own. If you cannot tell what would constitute a notable change in your outcome of interest, you probably don't know the target well enough to interpret the model regarding it, and you need to do some more research.

uncertainty, and along with the coefficients, it's used to calculate everything else in the table.

The statistic, here a t-statistic from the Student t-distribution[9], is the ratio of the coefficient to the standard error. This gives us a sense of the effect relative to its variability, but the statistic's primary use is to calculate the **p-value** related to its distribution[10], which is the probability of seeing a coefficient as large (or larger) as the one we have, *if* we assume from the outset that the true value of the coefficient is zero, *and* the model assumptions are true as well. In this case, the p-value is 3.47e-29, which is extremely small. We can conclude that the coefficient is statistically different from zero, and that the feature is related to the target, at least statistically speaking. However, the interpretation we used regarding the coefficient previously is far more useful than the p-value, as the p-value can be affected by many things not necessarily related to the feature-target relationship, such as sample size.

Aside from the coefficients, the most important output is the **confidence interval** (CI). The CI is a range of values that encapsulates the uncertainty we have in our guess about the coefficients. While our best guess for the effect of word count on rating is -0.04, we know it's not *exactly* that, and the CI gives us a range of reasonable values we might expect the effect to be based on the data at hand and the model we've employed.

In this case, the default is a 95% confidence interval, and we can think of this particular confidence interval like throwing horseshoes. If we kept collecting data and running models, 95% of our CIs would capture the true value, and this is one of the many possible CIs we could have gotten. That's the technical definition, which is a bit abstract[11], but we can also think of it more simply as a range of values that are good guesses for the true value, whatever it may be. In this case, the CI is -0.05 to -0.035 with 95% confidence. We can also see that the CI is relatively narrow, which is also nice to see, as it implies that we have a good idea of what the coefficient is. If it was very wide, we would

[9]Most statistical tables of this sort will use a t (Student t-distribution), Z (normal distribution), or F (F-distribution) statistic. It doesn't really matter for your purposes which is used by default, but the distribution is used to provide the p-value of interest and claim statistical significance (or not).

[10]You can calculate this as `pt(stat, df = model degrees of freedom, lower=FALSE)*2` in R, or use `stats.t.cdf` in Python. The model degrees of freedom provided in the summary output (a.k.a. residual degrees of freedom) are used when obtaining the two-sided p-value, which is what we want in this case. When it comes to t and Z statistics, anything over 2 is statistically significant by the common standard of a p-value of .05 or less. Note that even though output will round it to zero, the true p-value can never be zero.

[11]The interpretation regarding the CI is even more nuanced than this, but we'll leave that for another time. For now, we'll just say that the CI is a range of values that are good guesses for the true value. Your authors have used frequentist and Bayesian statistics for many years, and we are fine with both of them, because they both work well enough in the real world. Despite where this ranged estimate comes from, the vast majority use CIs in the same way, and they are a useful tool for understanding the uncertainty in our estimates.

have a lot of uncertainty about the coefficient, and we may not want to base important decisions regarding it.

Keep in mind that your chosen model has a great influence on what you'll be able to say at the feature level. As an example, as we get into machine learning models, you won't have as easy a time with coefficients and their confidence intervals, but you still may be able to say something about how your features relate to the target, and we'll continue to return to the topic. But first, let's take a look at interpreting things in another way.

> **i** Hypothesis Testing
>
> The confidence interval and p-value will for coefficients in typical statistical linear models will coincide with one another in that, for a given alpha significance level, if the 1-alpha% CI includes zero, then your p-value will be greater than alpha, and vice versa. This is because the same standard error is used to calculate both. However, the framework of using a CI vs. using the p-value for claiming statistical significance actually came from individuals that were philosophically opposed. Modern-day usage of both is a bit of a mess that would upset both Fisher (p-value guy) and Neyman (CI guy), but your authors find that this incorrect practical usage doesn't make much practical difference in the end.

3.4.2 Model-level interpretation

So far, we've focused on interpretation at the feature level. But knowing the interpretation of a feature doesn't do you much good if the model itself is poor! In that case, we also need to assess the model as a whole, and as with the feature level, we can go about this in a few ways. Before getting too carried away with asking whether your model is any good or not, you always need to ask yourself *relative to what*? Many models claim top performance under various circumstances, but which are statistically indistinguishable from many other models. So we need to be careful about how we assess our model, and what we compare it to.

Predictions vs. observed

When we looked at the models previously in Figure 3.3, we examined how well the predictions and target line up, and that gave us an initial feel for how well the model fits the data. Most model-level interpretation involves assessing and comparing model fit and variations on this theme. Here we show how easy it is to obtain such a plot.

R

```
predictions = predict(model_lr_rating)
y = df_reviews$rating

ggplot(
    data = data.frame(y = y, predictions = predictions),
    aes(x = y, y = predictions)
) +
    geom_point() +
    labs(x = 'Predicted', y = 'Observed')
```

Python

```
import matplotlib.pyplot as plt

predictions = model_lr_rating.predict()
y = df_reviews.rating

plt.scatter(y, predictions)
```

Model metrics

We can also get an overall assessment of the prediction error from a single metric. In the case of the linear model we've been looking at, we can express this as the sum or mean of our squared errors, the latter of which is a very commonly used modeling metric: **MSE** or **mean squared error** . Its square root, **RMSE** or **root mean squared error**[12], is also very commonly used. We'll talk more about this and similar metrics elsewhere (Section 4.2), but we can take a look at the RMSE for our model now.

If we look back at our results, we can see this expressed as the part of the output or as an attribute of the model[13]. The RMSE is more interpretable, as it gives us a sense that our typical errors bounce around by about 0.59. Given that the rating is on a 1-5 scale, this maybe isn't bad, but we could definitely hope to do better than get within roughly half a point on this scale. We'll talk about ways to improve this later.

[12]Any time we're talking about MSE for performance, we're also talking about RMSE, as it's just the square root of MSE, so which one you choose is mostly arbitrary. Taking the square root makes the metric more interpretable, as it's in the same units as the target, but it's not necessary to claim one model performs better than another.

[13]The actual divisor for linear regression output depends on the complexity of the model, and in this case the sum of the squared errors is divided by N-2 (due to estimating the intercept and coefficient) instead of N. This is a technical detail that would only matter for data too small to generalize beyond anyway, and not important for our purposes here.

R

```
# summary(model_lr_rating) # 'Residual standard error' is approx RMSE
summary(model_lr_rating)$sigma   # We can extract it directly
```

```
[1] 0.5907
```

Python

```
np.sqrt(model_lr_rating.scale)   # RMSE
```

```
0.590728780660127
```

Another metric we can use to assess **model fit** in this particular situation is the **mean absolute error**(MAE). MAE is similar to the mean squared error, but instead of squaring the errors, we just take the absolute value. Conceptually it attempts to get at the same idea, how much our predictions miss the target on average, and here the value is 0.46, which we actually showed in our residual plot (Figure 3.4). With either metric, the closer to zero the better, since as we get closer, we are reducing prediction error.

We can also look at the **R-squared** (R^2) value of the model. R^2 is possibly the most popular measure of model performance with linear regression and linear models in general. Before squaring, it's just the correlation of the predicted versus observed values that we saw in the previous plot (Figure 3.3). When we square it, we can interpret it as a measure of how much of the variance in the target is explained by the model. In this case, our model shows the R^2 is 0.12, which is not bad for a single feature model in this type of setting. We interpret the value as 12% of the target variance is explained by our model, and more specifically by the features in the model. In addition, we can also interpret R^2 as 1 - the proportion of error variance in the target, which we can calculate as $1 - \frac{\text{MSE}}{var(y)}$. In other words the complement of R^2 is the proportion of the variance in the target that is not explained by the model. Either way, since 88% is not explained by the model, our result suggests there is plenty of work left to do!

Note also, that with R^2 we get a sense of the variance shared between *all* features in the model and the target, however complex the model gets. As long as we use it descriptively as a simple correspondence assessment of our predictions and target, it's a fine metric. For various reasons, it's not a great metric for comparing models to each other, but again, as long as you don't get carried away, it's okay to use.

3.4.3 Prediction vs. explanation

In your humble authors' views, one can't stress enough the importance of a model's ability to predict the target. It can be a poor model, maybe because the data is not great, or perhaps we're exploring a new area of research, but we'll always be interested in how well a model **fits** the observed data. In situations where we're focused on how things will work out in the future, we're just as much or even more interested in how well a model predicts *new* data.

In many settings, **statistical significance** is focused on a great deal, and much is made about models that may actually have little predictive power. As strange as it may sound to some, you can read results in journal articles, news features, and business reports in many fields with hardly any mention of a model's predictive capability. In these cases, the focus is almost entirely on the **explanation** of the model, and usually the statistical significance of the features with regard to their relationship to the target.

In those settings, statistical significance is often used as a proxy for importance, though this is rarely ever justified. As we've noted elsewhere, statistical significance is affected by other things besides the size of the coefficient. And without an understanding of the context of the features, in this case, like how long typical reviews are, what their range is, what variability of ratings is, etc., the information it provides is extremely limited, and many would argue, not very useful.

If we are very interested in the coefficient or weight value specifically, it is better to focus on the range of possible values. This is provided by the confidence interval, along with the predictions that come about based on that coefficient's value, which will likewise have interval estimates. Like statistical significance, a confidence interval is also a 'loaded' description of a feature's relationship to the target, not without issues. However, we can use it in a very practical way as a range of possible values for that feature's weight, and more importantly, *think of possibilities rather than certainties.*

Suffice it to say at this point, that how much one focuses on prediction versus explanation depends on the context and goals of the data endeavor. There are cases where predictive capability is of utmost importance, and we care less about explanatory details, but not to the point of ignoring it. For example, even with deep learning models for image classification, where the inputs are just RGB values from an image, we'd still like to know what the (notably complex) model is picking up on. Otherwise, we may be classifying images based on something like image backgrounds (e.g., outdoors vs. indoors) instead of the objects of actual interest (dogs vs. cats). In some business or other organizational settings, we are very, or even mostly, interested in the coefficients/weights, which might indicate how to allocate monetary resources

in some fashion. But if those weights come from a model with no predictive power, placing much importance on them may be a fruitless endeavor.

In the end we'll need to balance our efforts to suit the task at hand. Prediction and explanation are both fundamental to the modeling endeavor. We return to this topic again in Chapter 13 (Section 13.2).

3.5 Adding Complexity

We've seen how to fit a model with a single feature and interpret the results, and that helps us to get oriented to the general process when using a linear model. However, we'll always have more than one feature for a model except under some very specific circumstances, such as exploratory data analysis. So let's see how we can implement a model with more features and that makes more practical sense.

3.5.1 Multiple features

We can add more features to our model very simply. Using the standard functions we've already demonstrated, we just add them to the formula as follows[14].

```
'y ~ feature_1 + feature_2 + feature_3'
```

In other cases where we use matrix inputs, additional features will just be the additional input columns, and nothing conceptually about the model actually changes.

```
# X are features, y is the target
GenericModel(X, y)
```

We might have a lot of features, and even for relatively simple linear models this could be dozens in some scenarios. A compact depiction of our model uses matrix representation, which we'll show in the next callout, but you can find more detail in the matrix overview Appendix B. For our purposes with a standard linear model, all you really need to know is that this formula:

$$y = X\beta + \epsilon \qquad \text{or} \qquad y = \alpha + X\beta + \epsilon \tag{3.6}$$

[14]This is the case for both R and statsmodels.

is the same as this:

$$y = \alpha + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \cdots + \epsilon$$

where y is the target, X is a 2-d matrix of features[15], where the rows are observations/instances and columns features, and β is a vector of coefficients or weights corresponding to the number of columns in X. Matrix multiplication provides us an efficient way to get our expected value/prediction, and depicting the model in this way is a common practice that makes it more succinct.

Matrix Representation of a Linear Model

Here we'll show the matrix representation form of the linear model in more detail. In the following, y is a vector of all target observations, and X is a matrix of features. The β vector is the vector of coefficients. The column of 1s serves as a means to incorporate the intercept, as it's just multiplied by whatever the estimated intercept value is. Matrix multiplication form can be seen as an efficient way to get the sum of the features multiplied by their coefficients.
Here is y as a vector of observations, $n \times 1$.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{3.7}$$

Here is the $n \times p$ matrix of features, including the intercept:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \dots & x_{1p} \\ 1 & x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \dots & x_{np} \end{bmatrix} \tag{3.8}$$

And finally, here is the $p \times 1$, vector of coefficients:

$$= \begin{bmatrix} \mathrm{b}_0 \\ \mathrm{b}_1 \\ \vdots \\ \mathrm{b}_\mathrm{p} \end{bmatrix} \tag{3.9}$$

[15] In the first depiction without α, there is an additional column at the beginning of the matrix that is all 1s, which is a way to incorporate the intercept into the model. However, most models that use a matrix as input will not have the intercept column, as it's either not part of the model estimation, or is automatically added behind the scenes, and may be estimated separately.

Putting it all together, along with the error term, we get the linear model in matrix form:

$$\mathbf{y} = \mathbf{X} + \tag{3.10}$$

You will also see it depicted in a transposed fashion, such that $y = \beta^\top X$, or $f(x) = w^\top X + b$, with the latter formula typically seen in the context of machine learning. This is just a matter of preference, except that it may assume the data is formatted in a different way, or possibly an author is talking about matrix/vector operations for a single observation. You'll want to pay close attention to what the dimensions are.
For the models considered here and almost all 'tabular data' scenarios, the data is stored in the fashion we've represented in this text, but you should be aware that other data settings will force you to think of multi-dimensional arrays[16] instead of 2-d matrices, for example, with image processing. So it's good to be flexible.

With that in mind, let's get to our model! In what follows, we keep the word count, but now we add some aspects of the reviewer, such as age and the number of children in the household, and features related to the movie, like the release year, the length of the movie in minutes, and the total reviews received. We'll use the same approach as before, and literally just add them as we depicted in our linear model formula (Equation 3.3).

R

```
model_lr_rating_extra = lm(
    rating ~
        word_count
        + age
        + review_year
        + release_year
        + length_minutes
        + children_in_home
        + total_reviews,
    data = df_reviews
)

summary(model_lr_rating_extra)
```

```
Call:
```

[16] In deep learning, model arrays are referred to as the more abstract representation of **tensors**, but for practical purposes the distinction doesn't really matter for modeling, as the tensors are always some n-dimensional array.

```
lm(formula = rating ~ word_count + age + review_year + release_year +
    length_minutes + children_in_home + total_reviews, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max
-1.8231 -0.3399  0.0107  0.3566  1.5144

Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)      -4.56e+01   7.46e+00   -6.11  1.5e-09 ***
word_count       -3.03e-02   3.33e-03   -9.10  < 2e-16 ***
age              -1.69e-03   9.24e-04   -1.83   0.0683 .
review_year       9.88e-03   3.23e-03    3.05   0.0023 **
release_year      1.33e-02   1.79e-03    7.43  2.3e-13 ***
length_minutes    1.67e-02   1.53e-03   10.90  < 2e-16 ***
children_in_home  1.03e-01   2.54e-02    4.05  5.5e-05 ***
total_reviews     7.62e-05   6.16e-06   12.36  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.52 on 992 degrees of freedom
Multiple R-squared:  0.321, Adjusted R-squared:  0.316
F-statistic:   67 on 7 and 992 DF,  p-value: <2e-16
```

Python

```
model_lr_rating_extra = smf.ols(
    formula = 'rating ~ word_count \
        + age \
        + review_year \
        + release_year \
        + length_minutes \
        + children_in_home \
        + total_reviews',
    data = df_reviews
).fit()

model_lr_rating_extra.summary(slim = True)
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                           OLS Regression Results
==============================================================================
Dep. Variable:                 rating   R-squared:                       0.321
Model:                            OLS   Adj. R-squared:                  0.316
```

```
No. Observations:                1000   F-statistic:                    67.02
Covariance Type:            nonrobust   Prob (F-statistic):           3.73e-79
================================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
Intercept          -45.5688      7.463     -6.106      0.000     -60.215     -30.923
word_count          -0.0303      0.003     -9.102      0.000      -0.037      -0.024
age                 -0.0017      0.001     -1.825      0.068      -0.004       0.000
review_year          0.0099      0.003      3.055      0.002       0.004       0.016
release_year         0.0133      0.002      7.434      0.000       0.010       0.017
length_minutes       0.0167      0.002     10.897      0.000       0.014       0.020
children_in_home     0.1028      0.025      4.051      0.000       0.053       0.153
total_reviews     7.616e-05   6.16e-06     12.362      0.000    6.41e-05    8.83e-05
================================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.82e+06. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

There is definitely more to unpack here than our simpler model, but it's important to note that it's just *more* stuff, not *different* stuff. The model-level components are the same in that we still see R^2, etc., although they are all 'better' (higher R^2, lower error) because we have a model that more accurately predicts the observed target.

Our coefficients have the same output, and though they are on different scales we'd interpret them in the same way. Starting with word count, we see that it's still statistically significant, but it has been reduced just slightly from our previous model where it was the only feature (-0.04 vs. -0.03). Why? This suggests that word count has some non-zero correlation, sometimes called **collinearity**, with other features that are also explaining the target to some extent. Our linear model shows the effect of each feature *controlling for other features*, or, *holding other features constant*, or *adjusted for other features*[17]. Conceptually this means that the effect of word count is the effect of word count *after* we've accounted for the other features in the model. In this case, an increase of a single word results in a -0.03 drop, even after adjusting for the effect of other features. Looking at another feature, the addition of a child to the home is associated with 0.1 increase in rating, again, accounting for the other features.

[17] A lot of statisticians and causal modeling folks get very hung up on the terminology here, but we'll leave that to them, as we'd like to get on with things. Don't get us wrong, the distinctions are useful. But for our purposes, we'll just say that we're interested in the relationship of a feature with the target *after* we've accounted for the other features in the model.

> **Features Scales Again**
>
> The scales of the features are quite different, so we can't directly compare the coefficients. For example, the word count coefficient represents a movement of 1 word, and coefficient for release year represents a movement of 1 year. One way to get a better comparison is to standardize the features as we talked about previously (Section 3.4.1), and which we'll talk about more in the data chapter (Section 14.2) and elsewhere.

Thinking about prediction, how would we get a prediction for a movie rating with a review that is 12 words long, written in 2020, by a 30-year-old with one child, for a movie that is 100 minutes long, released in 2015, with 10,000 total reviews? Exactly the same as we did before (Section 3.3.2)! We just create a dataframe with the values we want, and we predict accordingly.

R

```
predict_observation = tibble(
    word_count = 12,
    age = 30,
    children_in_home = 1,
    review_year = 2020,
    release_year = 2015,
    length_minutes = 100,
    total_reviews = 10000
)

predict(
    model_lr_rating_extra,
    newdata = predict_observation
)
```

```
   1
3.26
```

Python

```
predict_observation = pd.DataFrame(
    {
        'word_count': 12,
        'age': 30,
        'children_in_home': 1,
        'review_year': 2020,
        'release_year': 2015,
```

```
        'length_minutes': 100,
        'total_reviews': 10000
    },
    index = ['new_observation']
)

model_lr_rating_extra.predict(predict_observation)
```

```
new_observation   3.260
dtype: float64
```

In our example we're just getting a single prediction, but don't let that hold you back! As we did before, you can predict an entire dataset if you want and use any values for the features you want. Feel free to try a different prediction of your choosing!

3.5.2 Categorical features

Categorical features can be added to a model just like any other feature. The main issue is that they have to be represented numerically, because models only work on numerically coded features and targets. The simplest and most common encoding is called a **one-hot encoding** scheme, which creates a new feature column for each category, and assigns a 1 if the observation has that category label, and a 0 otherwise. This is also called **dummy coding** when used for statistical models. Here is an example of what the coding looks like for the season feature. This is really all there is to it.

Table 3.6: One-Hot Encoding of the Season Feature

rating	season	Fall	Summer	Winter	Spring
2.70	Fall	1	0	0	0
4.20	Fall	1	0	0	0
3.70	Fall	1	0	0	0
2.70	Fall	1	0	0	0
2.40	Summer	0	1	0	0
4.00	Summer	0	1	0	0
1.80	Fall	1	0	0	0
2.40	Summer	0	1	0	0
2.50	Winter	0	0	1	0
4.30	Summer	0	1	0	0

When using statistical models we don't have to do this ourselves. Even other tools for machine learning models will typically have a way to identify and appropriately handle categorical features, even in very complex ways when it

comes to deep learning models. What is important is to be aware that they require special handling, even if this is done behind the scenes. Now let's do a quick example using a categorical feature with our data, and we'll keep a numeric feature as well just for consistency.

R

```
model_lr_cat = lm(
    rating ~ word_count + season,
    data = df_reviews
)

summary(model_lr_cat)
```

```
Call:
lm(formula = rating ~ word_count + season, data = df_reviews)

Residuals:
    Min      1Q  Median      3Q     Max
-1.9184 -0.3622  0.0133  0.3589  1.8372

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    3.3429     0.0530   63.11  < 2e-16 ***
word_count    -0.0394     0.0036  -10.96  < 2e-16 ***
seasonSpring  -0.0301     0.0622   -0.48     0.63
seasonSummer   0.2743     0.0445    6.17  9.8e-10 ***
seasonWinter  -0.0700     0.0595   -1.18     0.24
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.572 on 995 degrees of freedom
Multiple R-squared:  0.176, Adjusted R-squared:  0.173
F-statistic: 53.1 on 4 and 995 DF,  p-value: <2e-16
```

Python

```
model_lr_cat = smf.ols(
    formula = 'rating ~ word_count + season',
    data = df_reviews
).fit()

model_lr_cat.summary(slim = True)
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                 rating   R-squared:                       0.176
Model:                            OLS   Adj. R-squared:                  0.173
No. Observations:                1000   F-statistic:                     53.09
Covariance Type:            nonrobust   Prob (F-statistic):           1.41e-40
==============================================================================
                     coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept            3.3429      0.053     63.109      0.000       3.239       3.447
season[T.Spring]    -0.0301      0.062     -0.483      0.629      -0.152       0.092
season[T.Summer]     0.2743      0.044      6.171      0.000       0.187       0.362
season[T.Winter]    -0.0700      0.059     -1.177      0.239      -0.187       0.047
word_count          -0.0394      0.004    -10.963      0.000      -0.047      -0.032
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
"""
```

We now see the usual output. There is word count again, with its slightly negative association with rating. And we have an effect for each season as well… except, wait a second, where is the fall effect? The coefficients are interpreted the same way. As we move one unit on x, we see a corresponding change in y. But moving from one category to another requires starting at some category in the first place: a reference point. So one category is chosen arbitrarily, but you would have control over this. In our model, 'fall' is chosen just because it is first alphabetically. So if we look at, for example, the effect of summer, we see an increase in the rating of 0.27 relative to fall. The same goes for the other seasons, as they all represent a change relative to fall.

Recall also that an interpretation of the intercept is the expected value of the target when all features are zero. In this case, it's the expected value of the target when the word count is zero and the season is fall.

Summarizing categorical features

When we have a lot of categories, it's often not practical to look at the coefficients for each one, and even when there aren't that many, we often prefer to get a sense of the total effect of the feature. For standard linear models, we can break down the target variance explained by the model into the variance explained by each feature, and this is called the **ANOVA**, or analysis

of variance[18]. It is not without its issues, but it's one way to get a sense of whether a categorical (or other) feature as a whole is statistically significant.

R

```
anova(model_lr_cat)
```

Python

```
import statsmodels.api as sm

sm.stats.anova_lm(model_lr_cat)
```

Table 3.7: ANOVA Table for Categorical Feature

Feature	DF	sumsq	meansq	F-stat.	p.value
word_count	1.00	46.80	46.80	143.02	< 0.001
season	3.00	22.69	7.56	23.12	< 0.001
Residuals	995.00	325.57	0.33		

A primary reason to use ANOVA is to make these sorts of summary claims of statistical significance. In this case, we can say that the relationship of season to rating is statistically significant. From Table 3.7, the DF (degrees of freedom) represents the number of categories minus 1, and the F-statistic is a measure of the mean squared variance explained by the feature relative to the (mean squared) variance not explained by the feature (F = mean square value divided by mean square error, or residual variance). The p-value is the probability of observing an F-statistic as extreme as the one observed, given that the null hypothesis is true. In this case, the null hypothesis is that the feature has no effect on the target. The p-value is less than 0.001, so we reject the null hypothesis and conclude that the observed feature-target relationship is statistically different from an assumption of no relationship. Note that nothing here is different from what we saw in our previous regression models, and we can run an `anova` function on those too[19]. As a final note, it's good to be

[18]There are actually different types of ANOVA in this context, and different ways to calculate the variance values. One may notice the Python ANOVA result is different, even though the season coefficients and initial model are identical. R defaults with what is called Type II sums of squares, while the Python default uses Type I sums of squares. We won't bore you with the details of their differences, and the astute modeler will not come to different conclusions because of this sort of thing, and you now have enough detail to look it up.

[19]For those interested, for those features with one degree of freedom, all else being equal, the F-statistic here would just be the square of the t-statistic for the coefficients, and the p-value would be the same.

reminded that statistical significance is not the same as practical significance. Whether these group differences are meaningful is still left to be decided by the modeler given the context of the data.

Group predictions

A better approach to understanding categorical features for standard linear models is through what are called **marginal effects**, which can provide a kind of average prediction for each category while accounting for the other features in the model. Better still is to visualize these. It's actually tricky to define 'average' when there are multiple features and interactions involved, so be more cautious in those contexts. In this case, we expect the highest ratings for summer releases. We'll return more to this concept in Section 5.5.

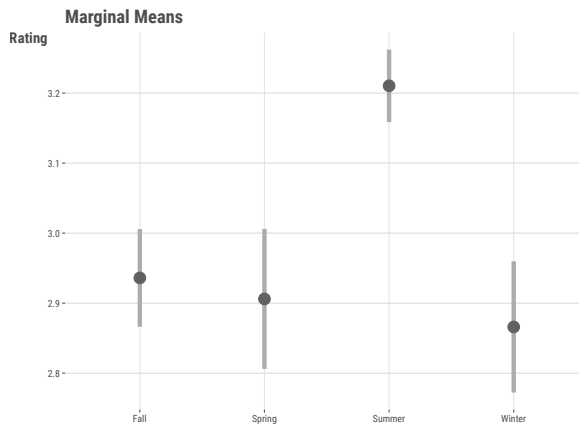


Figure 3.6: Marginal effects of season on rating

3.5.3 Other model complexities

There are a lot more fun things we can do while still employing a linear model. We can add **interactions** between features, account for non-linear relationships, and enhance the linear model we've seen to improve predictions. We'll talk more about these types of techniques throughout the rest of the book.

3.6 Assumptions and More

Every model you use has underlying assumptions which, if not met, could potentially result in incorrect inferences about the effects, performance, or

predictive capabilities of the model. These are assumptions about the data generating process, the stability of the data, the correctness of the data, the appropriateness of the model, and so on.

The standard linear regression model we've come to know is no different, and it has a number of assumptions that must be met for it to be *statistically* valid. Briefly, they are:

- The model is not grossly misspecified (e.g., you've included the right features and not left out important ones)
- The data that you're modeling reflects the population you want to make generalizations about
- The model is linear in the parameters (i.e., no e^{β} or $\beta_1 \cdot beta_2 \cdot X$ type stuff)
- The features are not correlated with the error (prediction errors, unobserved causes)
- Your data observations are independent of each other
- The prediction errors are homoscedastic (e.g., some predictions aren't associated with very large errors relative to others)
- Normality of the errors (i.e., your prediction errors). Another way to put it is that your target variable is normally distributed *conditional* on the features.

A linear regression model does not assume that:

- The features are normally distributed
  - For example, using categorical features is fine
- The target is normally distributed
  - The assumed target distribution is *conditional* on the features, the target (so-called *marginal*) distribution can be whatever it is
- The relationship between the features and target is linear
  - Interactions, polynomial terms, etc. are all fine
- The features are not correlated with each other
  - They usually are

If you do meet these assumptions, it doesn't mean that:

- You have large effects
- You have a well-performing model
- You have causal effects
- You (necessarily) have less uncertainty about your coefficients or predictions than other methods

If you don't meet these assumptions, it doesn't mean that:

- Your model will have poor predictions
- Your conclusions will necessarily be incorrect or even different

And finally, most of the time you can use a different type of linear model to meet these assumptions.

So basically, whether or not you meet the assumptions of your model doesn't actually say much about whether the model is great or terrible. For the linear regression model, if you do meet those assumptions, your coefficient estimates are unbiased[20], and in general, your statistical inferences are valid ones. If you don't meet the assumptions, there are alternative versions of the linear model you could use that would potentially address the issues.

For example, data that runs over a sequence of time (**time series** data) violates the independence assumption, since observations closer in time are more likely to be similar than those farther apart. Violation of this assumption will result in problems with the standard errors of the coefficients, and thus the p-values and confidence intervals. But we could use a **time series** or similar model instead to account for this. If normality is difficult to meet, you could assume a different data generating distribution. We'll discuss some of these approaches explicitly in later chapters (e.g., Chapter 8), but it's also important to note that not meeting the assumptions for the model may only mean you'll prefer a different type of linear or other model to use in order to meet them.

3.6.1 Assumptions with more complex models

Let's say you're running some XGBoost or a Deep Linear Model and getting outstanding predictions. 'Assumptions shmumptions' you say! And you might even be right! But if you want to talk confidently about feature contributions, or know something about the uncertainty in the predictions (which you're assessing, right?), well, maybe you might want to know if you're meeting your assumptions. Some of them are:

- You have enough data to make the model generalizable
- Your data isn't biased (e.g., you don't have 90% of your data from one particular region when you want to talk about a much wider area)
- You adequately sampled the hyperparameter space (e.g., you didn't just use the defaults (Section 15.2.2) or a small grid search)
- Your observations are independent or at least exchangeable and don't have data leakage (Section 15.3.5), or you are explicitly modeling observation dependence
- Your parameter settings you've chosen are correct or at least viable (e.g.,

[20]This means they are correct *on average*, not that they are the *true* value. And if they were biased, this refers to **statistical bias**, and has nothing to do with the moral or ethical implications of the data, or whether the features themselves are biased in measurement. Culturally-biased data is a different problem than statistical/prediction bias or measurement error, though they are not mutually exclusive. Statistical bias can more readily be tested, while other types of bias are more difficult to assess. Even statistical unbiasedness is not necessarily a goal, as we will see later in Section 6.8.

you let the model run for a long enough set of iterations, your batch size was adequate, you had enough hidden layers, etc.)

And if you want to talk about specific feature contributions, you are assuming:

- The features are largely uncorrelated
- The features largely do not interact[21], or that your understanding of feature contribution deals with the interactions

The take-home message is that using models in more complex settings like machine/deep learning doesn't mean you don't have to worry about theoretical and model assumptions. You still have much to consider!

3.7 Classification

Up to this point we've been using a continuous, numeric target. But what about a categorical target? For example, what if we just had a binary target of whether a movie was good or bad? We will dive much more into classification models in our upcoming chapters, but it turns out that we can still formulate it as a linear model, the most common one being a **logistic regression**. The main difference is that we use a transformation of our linear combination of features, using what is sometimes called a **link function**, and we'll need to use a different **objective function** rather than least squares, such as the binomial likelihood, to deal with the binary target. This also means we'll move away from R^2 as a measure of model fit, and focus on something else, like accuracy.

Graphically we can see it in the following way which, when compared with our linear model (Figure 3.2), doesn't look much different. In what follows, we create our linear combination of features exactly as we did for the linear regression setting. Then we put it through the sigmoid function, which is a common link function for binary targets[22]. The result is a probability, which we can then use to classify the observation as good or bad based on a chosen threshold. For example, we might say that any instance associated with a probability greater than 0.5 is classified as 'good', and less than that is classified as 'bad'.

[21] But then why would you be using a complex model that is inherently interacting the features?

[22] The sigmoid function in this case is the inverse logistic function, and the resulting statistical model is called logistic regression. In other contexts the model would not be a logistic regression, but this is still a commonly used **activation function**. But others could potentially be used as well. For example, using a (cumulative) normal instead of logistic distribution to create a probability results in the so-called **probit model**, which is more common in econometrics and other fields.

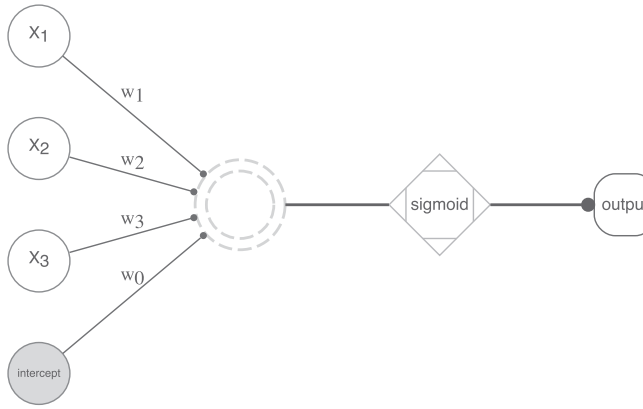


Figure 3.7: Linear model with transformation can be a logistic regression

As soon as we move away from the standard linear model and use transformations of our linear predictor, simple coefficient interpretation becomes difficult, sometimes exceedingly so. We will explore more of these types of models and how to interpret them in later chapters (e.g., Chapter 8).

3.8 More Linear Models

Before we leave our humble linear model, let's look at some others. Here is a brief overview of some of the more common 'linear' models you might encounter but maybe didn't realize they were still just a linear model not too far removed from linear regression.

Generalized Linear Models and Related:

- True generalized linear models (GLM) e.g., logistic, poisson
- Other distributions: beta regression, tweedie, t (so-called robust), truncated
- Penalized regression: ridge, lasso, elastic net
- Censored outcomes: survival models, tobit

Multivariate/multiclass/multipart:

- Multivariate regression (multiple targets)
- Multinomial/Categorical/Ordinal regression (>2 classes)
- Zero (or some number) -inflated/hurdle/altered
- Mixture models and cluster analysis

Random Effects:

- Mixed effects models (random intercepts/coefficients)
- Generalized additive models (GAMs)
- Spatial models (CAR)
- Time series models (ARIMA)
- Factor analysis

Latent Linear Models:

- PCA, Factor Analysis
- Mixture models
- Structural Equation Modeling, graphical models generally

All of these are explicitly linear models or can be framed as such, and may only require only a tweak or two from what you've already seen. For example, they may have a different distributional assumption, a different link function, penalizing the coefficients, etc. In other cases, we can bounce from one to another and even get similar results. For instance, we can reshape our multivariate outcome to be amenable to a mixed model approach and get the exact same results. We can potentially add a random effect to any model, and that random effect can be based on time, spatial or other considerations. Additionally, the same type of linear combination of features used in linear regression can be used in many types of models, even deep learning models!

The important thing to know is that the linear model is a very flexible tool that expands easily and allows you to model most of the types of outcomes we are interested in. As such, it's a very powerful approach to modeling.

3.9 Wrapping Up

Linear models, such as the linear regression demonstrated in this chapter, are a very popular tool for data analysis, and for a good reason. They are relatively easy to implement and very flexible. They can be used for prediction, explanation, and inference, and they can be used across a wide variety of data types. There are also many tools at our disposal to help us use and explore them. But the simpler demos we've seen here are not without their limitations, and you'll want to have more in your toolbox than just the approach we've seen so far.

3.9.1 The common thread

In most of the chapters we want to highlight the connections between models you'll encounter. Linear models are *the* starting point for modeling, and they

can be used for a wide variety of data types and tasks. The linear regression with a single feature is identical to a simple correlation if the feature is numeric, a t-test if it is binary, and an ANOVA if it is categorical. We explored a more complex model with multiple features, and we saw how to interpret the coefficients and make predictions. The creation of a combination of features to predict a target is the basis of all models, and as such, the linear regression model we've just seen is the real starting point on your data science journey.

3.9.2 Choose your own adventure

Now that you've got the basics, where do you want to go?

- If you want to know more about how to understand linear and other models: Chapter 4 and Chapter 5
- If you want a deeper dive into how we get the results from our model: Chapter 6
- If you want to do some more modeling: Chapter 8, Chapter 9, or Chapter 10
- Got more data questions? Chapter 14

3.9.3 Additional resources

If you are interested in a deeper dive into the theory and assumptions behind linear models, you can check out more traditional statistical/econometric treatments such as:

- Gelman, Hill, and Vehtari (2020)
- Gelman (2013)
- Harrell (2015)
- Fahrmeir et al. (2021)
- J. Faraway (2014)
- Wooldridge (2012)
- Greene (2017)

For more applied treatments, consider:

- Navarro (2018)
- Weed and Navarro (2021)
- Kuhn and Silge (2023)

But there are many, many books on statistical analysis, linear models, and linear regression specifically. Texts tend to get more mathy and theoretical as you go back in time, to the mostly applied and code-based treatments today. You will likely need to do a bit of exploration to find one you like best. We also recommend you check out the many statistics and modeling based courses like those on Coursera, EdX, and similar ones, and the many tutorials and blog posts on the internet. Great demonstrations of specific topics can be found on

YouTube, blog posts, and other places. Just start searching and you'll find a lot of great resources!

3.10 Guided Exploration

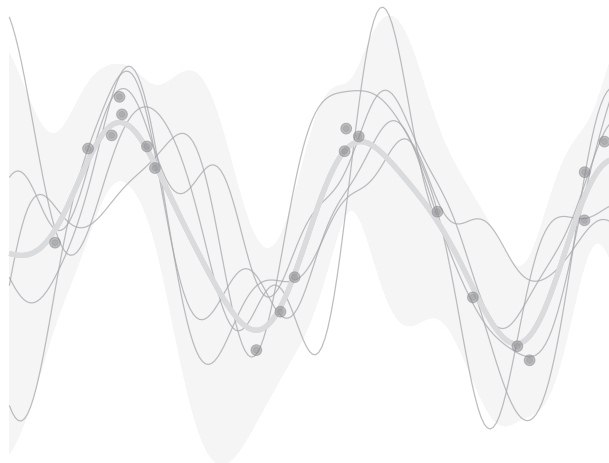
For this exercise let's switch to the world happiness 2018 dataset. You can find details about it in the appendix, Section C.2, and you can download it from the github repo.

- Fit a linear regression model, maybe keep it to three features or less:
  - Predict 'happiness' (`happiness_score`)
  - Suggestion for features: GDP per capita, Social support, Healthy life expectancy
- Summarize the model, and interpret the coefficients. What do you find?
- Assess the model fit with RMSE and R^2.
- Try to get a prediction of at least one new observation of interest, e.g., log GDP per capita of 10, life expectancy of 70, social support of 0.8, which would represent a decently well-off country. Contrast that prediction with a less well-off country, with values less than the median for each feature. What do you find?

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

4

Understanding the Model



In addition to giving the world one of the greatest television show theme songs – Quincy Jones' *The Streetbeater – Sanford & Son* gave us an insightful quote for offering criticism: "You big dummy." While we don't advocate for swearing at or denigrating your model, how do you know if your model is performing up to your expectations? It is easy to look at your coefficients, t-values, and an R^2, and say, "Wow! Look at this great model!" Your friends will be envious of such terrific p-values, and all of the strangers that you see at social functions will be impressed. What happens if that model falls apart on new data, though? What if a stakeholder wants to know exactly how a prediction was made for a specific business decision? Sadly, all of the stars that you gleefully pointed towards in your console will not offer you any real answers.

Instead of falling in immediate love with your model, you should ask some hard questions of it. How does it perform on different slices of data? Do predictions make sense? Is your classification cut-point appropriate? In other words, you should criticize your model before you decide it can be used for its intended purposes. Remember that it is *data modeling*, not *data truthing*. In other words, you should always be prepared to call your model a "big dummy".

4.1 Key Ideas

- Metrics can help you assess how well your model is performing, and they can also help you compare different models.
- Different metrics can be used depending on the goals of your model.
- Visualizations can further help us understand model performance in a variety of ways.

4.1.1 Why this matters

It's never good enough to simply get model results. You need to know how well your model is performing and how it is making predictions. You also should be comparing your model to other alternatives. Doing so provides more confidence in your model and helps you to understand how it is working, and just as importantly, where it fails. This is actionable knowledge.

4.1.2 Helpful context

This takes some of the things we see in other chapters on linear models and machine learning, so we'd suggest having the linear model basics down pretty well.

4.2 Model Metrics

A first step in understanding our model can be done with summary statistics, typically called **metrics**. Regression and classification have different metrics for assessing model performance. We want to give you a sample of some of the more common ones, but we also want to acknowledge that there are many more that you can use, and any might be useful. We would always recommend looking at a few different metrics for any given model to get a better sense of how your model is performing.

Table 4.1 illustrates some of the most commonly used performance metrics. Just because these are popular or applicable for your situation doesn't mean they are the only ones you can or even should use. Nothing keeps you from using more than one metric for assessment, and in fact, it is often a good idea to do so. In general though, you should have a working knowledge of these, as you will likely come across them even if you don't use them all the time.

Table 4.1: Commonly Used Performance Metrics in Machine Learning

Metric	Description	Other Names/Notes
Regression		
RMSE	Root mean squared error	MSE (before square root)
MAE	Mean absolute error	
MAPE	Mean absolute percentage error	
RMSLE	Root mean squared log error	
R-squared	Amount of variance shared by predictions and target	Coefficient of determination
Deviance/AIC	Generalization of sum of squared error	Also 'deviance explained' for similar R2 interpretation
Classification		
Accuracy	Proportion correct	Error rate is 1 - Accuracy
Precision	Proportion of positive predictions that are correct	Positive Predictive Value
Recall	Proportion of positive samples that are predicted correctly	Sensitivity, True Positive Rate
Specificity	Proportion of negative samples that are predicted correctly	True Negative Rate
Negative Predictive Value	Proportion of negative predictions that are correct	
F1	Harmonic mean of precision and recall	F-Beta[1]
AUC	Area under the ROC curve	
False Positive Rate	Proportion of negative samples that are predicted incorrectly	Type I Error, alpha
False Negative Rate	Proportion of positive samples that are predicted incorrectly	Type II Error, beta, Power is 1 - beta
Phi	Correlation between predicted and actual	Matthews Correlation
Log loss	Negative log-likelihood of the predicted probabilities	

[1]Beta = 1 for F1

4.2.1 Regression metrics

The primary goal of our endeavor is to come up with a predictive model. The closer our model predictions are to the observed target values, the better our model is performing. As we saw in Table 4.1, when we have a numeric target, there are quite a few metrics that help us understand prediction-target correspondence, so let's look at some of those.

But before we create a model to get us started, we are going to read in our data and then create two different splits within our data: a **training** set and a **test** set. In other words, we are going to **partition** our data so that we can train a model and then see how well that model performs with data it hasn't seen[1]. For more on this process and the reasons why we do it, see Section 10.4 and Section 10.6. For now, we just need to know that assessing prediction error on the test set will give us a better estimate of our metric of choice.

> **Splitting Data**
>
> This basic split is the foundation of **cross-validation**. Cross-validation is a method for partitioning data into training and non-training sets in a way that allows you to better understand the model's performance. You'll find more explicit demonstration of how to do this in the machine learning chapter Chapter 11.

R

```
# all data found on github repo
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

set.seed(123) # ensure reproducibility

initial_split = sample(
    x = 1:nrow(df_reviews),
    size = nrow(df_reviews) * .75,
    replace = FALSE
)

df_train = df_reviews[initial_split, ]

df_test = df_reviews[-initial_split, ]
```

[1] For anyone comparing Python to R results, the data splits are not the same, so outputs likewise will not be identical, though they should be very similar. We could have forced them to use the same data, but we feel you should get used to the randomness of the process.

Python

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split

# all data found on github repo
df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

df_train, df_test = train_test_split(
    df_reviews,
    test_size = 0.25,
    random_state = 123
)
```

You'll notice that we created training data with 75% of our data, and we will use the other 25% to test our model. This is an arbitrary but common split. With training data in hand, let's produce a model to predict review rating. We'll use the standardized (scaled `_sc`) versions of several features and use the 'year' features starting at year 0, which represents the earliest year observed in our data[2]. Finally, we also include the genre of the movie as a categorical feature.

R

```
model_lr_train = lm(
    rating ~
        review_year_0
        + release_year_0
        + age_sc
        + length_minutes_sc
        + total_reviews_sc
        + word_count_sc
        + genre
        ,
        df_train
)
```

[2]See Section 14.6.2 for more on why we like to start year features at 0.

Python

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

# we'll use 'features' later also
features = [
    "review_year_0",
    "release_year_0",
    "age_sc",
    "length_minutes_sc",
    "total_reviews_sc",
    "word_count_sc",
    "genre",
]

model =  'rating ~ ' + " + ".join(features)

model_lr_train = smf.ols(formula = model, data = df_train).fit()
```

Now that we have a model from our training data, we can use it to make predictions on our test data:

R

```
predictions = predict(model_lr_train, newdata = df_test)
```

Python

```
predictions = model_lr_train.predict(df_test)
```

The goal now is to find out how close our predictions match reality. Let's look at them first:

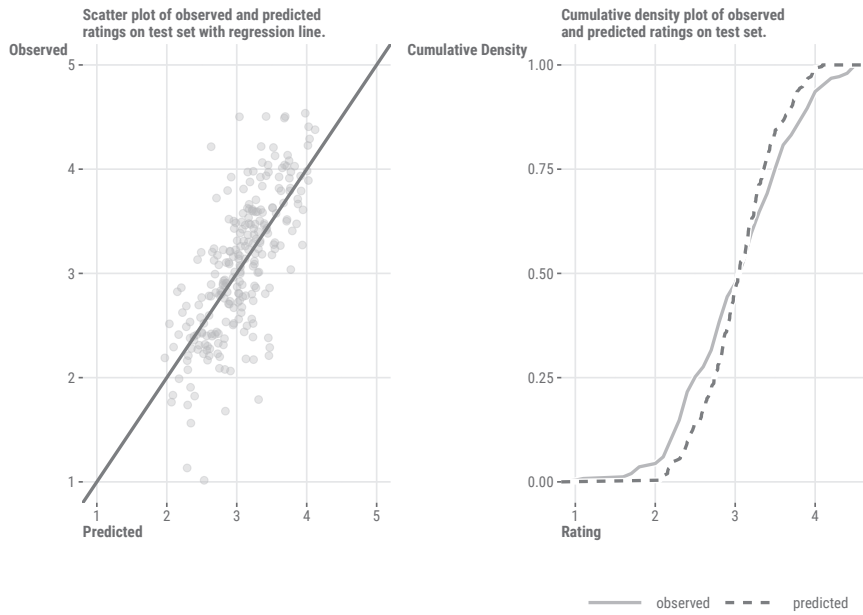


Figure 4.1: Observed vs. predicted ratings.

Obviously, our points do not make a perfect line on the left, which would indicate perfect prediction. Also, the distribution of our values suggests we're over-predicting on the lower end, as none of our predictions even go below 2. We're also under-predicting on the higher end of the target's range. More generally, we're not capturing the range of the observed values very well. But we'd like to determine how far off we are in a general sense. There are a number of metrics that can be used to measure this. We'll go through a few of them here. In each case, we'll demystify the calculations to make sure we understand what's going on.

R-squared

Anyone who has done linear regression has come across the R^2 value. It is a measure of how well the model explains the variance in the target. One way to calculate it is as follows:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{4.1}$$

where y_i is the observed value, $\hat{y}_i$ is the predicted value, and $\bar{y}$ is the mean of the observed values. The R^2 value is basically a proportion of how much

variance in the target (denominator) is attributable to the model features in the form of the predictions (numerator). When applied to the training or observed data, it is a value between 0 and 1, with 1 indicating that the model explains all of the variance in the target.

Alternatively, R^2 can be calculated as the squared correlation between the target and predicted values, which may be more conceptually intuitive. In that sense it can almost always be useful as a *descriptive* measure, just like we use means and standard deviations in exploratory data analysis. However, it is not so great at telling us about predictive quality. Why? Take the predictions from our rating model, and add 10 to them, or make them all negative. In both cases, if you calculate it as the squared correlation of your predictions and target, even though your predictions would be ridiculous, your R^2 will be the same. If you use the formula shown, you could even get negative values! Another problem is that for training data, R^2 will always increase as you add more features to your model, whether they are useful or pure noise! This is why we use other metrics to assess predictive quality.

R

```
residual_ss = sum((df_test$rating - predictions)^2)
total_ss = sum((df_test$rating - mean(df_test$rating))^2)

1 - residual_ss / total_ss

yardstick::rsq_trad_vec(df_test$rating, predictions)

# conceptually identical, but slight difference due
# to how internal calculations are done (not shown)
# cor(df_test$rating, predictions)^2
# yardstick::rsq_vec(df_test$rating, predictions)

# exercise
# cor(df_test$rating, predictions)^2
# cor(df_test$rating, predictions + 1)^2  # same
# yardstick::rsq_trad_vec(df_test$rating, predictions + 1) # negative!
```

```
[1] 0.5193
[1] 0.5193
```

Python

```
from sklearn.metrics import r2_score

residual_ss = np.sum((df_test.rating - predictions)**2)
total_ss = np.sum((df_test.rating - np.mean(df_test.rating))**2)

1 - residual_ss / total_ss

r2_score(df_test.rating, predictions)

# conceptually identical, but slight difference due to
# how calculations are done (not shown)
# np.corrcoef(df_test.rating, predictions)[0, 1]**2

# exercise
# np.corrcoef(df_test.rating, predictions)[0, 1]**2
# np.corrcoef(df_test.rating, predictions + 1)[0, 1]**2  # same
# r2_score(df_test.rating, predictions + 1)  # negative!
```

```
0.508431158347433
0.508431158347433
```

> **i** R-squared Variants
>
> There are different versions of R-squared. 'Adjusted' R-squared is a common one, and it penalizes the model for adding features that don't really explain the target variance. This is a nice sentiment, but its difference versus the standard R-squared would only be noticeable for very small datasets. Some have also attempted to come up with R-squared values that are more appropriate for GLMs for count, binary and other models. Unfortunately, these 'pseudo-R-squared' values are not as interpretable as the original R-squared, and they generally have several issues.

Mean squared error

One of the most common *performance* metrics for numeric targets is the mean squared error (MSE) and its square root, root mean squared error (RMSE). The MSE is the average of the squared differences between the predicted and actual values. It is calculated as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.2}$$

MSE penalizes large errors more. Since errors are squared, the larger the error, the larger the penalty. As mentioned previously, RMSE is just the square root of the MSE. We also saw that it has a similar metric reported in typical linear regression output (Section 3.4.2). Like MSE, RMSE also penalizes large errors, but if you want a metric that is in the same units as the original target data, RMSE is the metric for you. It is calculated as follows:

$$\text{RMSE} = \sqrt{\text{MSE}} \tag{4.3}$$

R

```
mse = mean((df_test$rating - predictions)^2)

mse

yardstick::rmse_vec(df_test$rating, predictions)^2
```

```
[1] 0.2133
[1] 0.2133
```

```
sqrt(mse)

yardstick::rmse_vec(df_test$rating, predictions)
```

```
[1] 0.4619
[1] 0.4619
```

Python

```
from sklearn.metrics import mean_squared_error, root_mean_squared_error

mse = np.mean((df_test.rating - predictions)**2)

mse

mean_squared_error(df_test.rating, predictions)
```

```
0.20798285555421575
0.20798285555421575
```

```
np.sqrt(mse)

root_mean_squared_error(df_test.rating, predictions)
```

```
0.4560513738102493
0.4560513738102493
```

Mean absolute error

The mean absolute error (MAE) is the average of the absolute differences between the predicted and observed values. It is calculated as follows:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.4}$$

MAE is a great metric when all you really want to know is how far off your predictions typically are from the observed values. It is not as sensitive to large errors as the MSE.

R

```
mean(abs(df_test$rating - predictions))

yardstick::mae_vec(df_test$rating, predictions)
```

```
[1] 0.352
[1] 0.352
```

Python

```
from sklearn.metrics import mean_absolute_error

np.mean(abs(df_test.rating - predictions))

mean_absolute_error(df_test.rating, predictions)
```

```
0.3704072983307527
0.3704072983307527
```

Mean absolute percentage error

The mean absolute percentage error (MAPE) is the average of the absolute differences between the predicted and observed values, expressed as a percentage of the observed values. It is calculated as follows:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i - \hat{y}_i|}{y_i} \tag{4.5}$$

R

```
mean(
    abs(df_test$rating - predictions) /
    df_test$rating
) * 100

yardstick::mape_vec(df_test$rating, predictions)
```

```
[1] 12.86
[1] 12.86
```

Python

```
from sklearn.metrics import mean_absolute_percentage_error

np.mean(
    abs(df_test.rating - predictions) /
    df_test.rating
) * 100

mean_absolute_percentage_error(df_test.rating, predictions) * 100
```

```
13.464399850975898
13.464399850975898
```

Which regression metric should I use?

In the end, it won't hurt to look at a few of these metrics to get a better idea of how well your model is performing. You will *always* be using at least one of these metrics to compare different models, so use a few of them to get a better sense of how well your models are performing relative to one another. Does adding a feature help drive down RMSE, indicating that the feature helps to reduce large errors? In other words, does adding complexity to your model provide a big reduction in error? If adding features doesn't help reduce error, do you really need to include them in your model?

4.2.2 Classification metrics

Whenever we are classifying outcomes in a binary case, we don't have the same ability to compare a predicted score to an observed score. Instead, we typically use the predicted probability of an outcome, establish a cut-point for that probability, convert everything below that cut-point to 0, and then

convert everything at or above that cut-point to 1. We can then compare a table predicted versus target **classes**, typically called a **confusion matrix**[3].

Let's start with a model to predict whether a review is "good" or "bad". We will use the same training and testing data that we created above. Explore the summary output if desired (not shown), but we will focus on the predictions and metrics.

R

```
model_class_train = glm(
    rating_good ~
        review_year_0
        + release_year_0
        + age_sc
        + length_minutes_sc
        + total_reviews_sc
        + word_count_sc
        + genre
        ,
        df_train,
        family = binomial
)

summary(model_class_train)

# a numeric version to use later
y_target_testing_bin = ifelse(df_test$rating_good == "good", 1, 0)
```

Python

```
import statsmodels.api as sm
import statsmodels.formula.api as smf

model =  'rating_good ~ ' + " + ".join(features)

model_class_train = smf.glm(
    formula = model,
    data = df_train,
```

[3]The origin of the term "confusion matrix" is a bit muddled, and it's not clear why it's not just called a classification table/matrix (as it actually is from time to time). If you call it a classification table, probably everyone will know exactly what you mean, but if you call it a confusion matrix, probably few outside of data science (or domains that use it) will know what you're talking about.

```
    family = sm.families.Binomial()
).fit()


# model_class_train.summary()
```

Now that we have our model trained, we can use it to get the predicted probabilities for each observation[4].

R

```
predicted_prob = predict(
    model_class_train,
    newdata = df_test,
    type = "response"
)
```

Python

```
predicted_prob = model_class_train.predict(df_test)
```

We are going to take those probability values and make a decision to convert everything above .5 to the positive class (a "good" review), which we can do simply by rounding, or with an if-else type of approach. It is a bold assumption, but one that we will make at first!

R

```
predicted_class = round(predicted_prob)

predicted_class = ifelse(predicted_prob > .5, 1, 0)
```

Python

```
predicted_class = predicted_prob.round().astype(int)
```

Confusion matrix

The confusion matrix is a table that shows the number of correct and incorrect predictions made by the model. It's easy enough to get one from scratch, but

[4]Machine learning libraries in Python based on the scikit-learn API will have a `predict_proba` method that will give you the probability of each class, while the `predict` method will give you the predicted class. The latter typically makes the assumption that the cut-point for classification is .5.

we recommend using a function that will give you a nice table, and possibly all of the metrics you need along with it. To get us started, we can use a package function that will take our predictions and observed target as input to create the basic table.

R

We use mlr3verse in the machine learning chapters, so we'll use it here for our confusion matrix. Though our predictions are 0/1, we need to convert it to a factor for this function.

```
# we'll use the mlr3verse in the machine learning demos also
rating_cm = mlr3measures::confusion_matrix(
    factor(df_test$rating_good),  # requires factor
    factor(predicted_class),
    positive = "1"                # class 1 is 'good'
)
```

Python

We can get an extremely rudimentary confusion matrix by using the `confusion_matrix` function from `sklearn.metrics`. We'll use it here to get the basic table, but we'll use a more advanced function later.

```
from sklearn.metrics import confusion_matrix

rating_cm = confusion_matrix(df_test.rating_good, predicted_class)
```

Table 4.2: Example of a Confusion Matrix

	True 1	True 0
Predicted 1	TP: 117	FP: 27
Predicted 0	FN: 22	TN: 84

- **TP**: True Positive is an outcome where the model correctly predicts the positive class – the model correctly predicted that the review was good.
- **TN**: True Negative is an outcome where the model correctly predicts the negative class – the model correctly predicted that the review was not good.
- **FP**: False Positive is an outcome where the model incorrectly predicts the positive class – the model incorrectly predicted that the review was good when it was bad.
- **FN**: False Negative is an outcome where the model incorrectly predicts

the negative class – the model predicted that the review was bad when it was good.

In an ideal world, we would have all of our observations fitting nicely in the diagonal of that table. Unfortunately, we don't live in that world, and the greater amount we have in the off-diagonal (i.e., in the FN and FP spots), the worse our model is at classifying outcomes.

Let's look at some metrics that will help to see if we've got a suitable model or not. We'll describe each, then show them all after.

Accuracy

Accuracy's allure is in its simplicity, and because we use it for so many things in our everyday affairs. It is simply the proportion of correct predictions made by the model. But accuracy can also easily mislead you into believing a model is doing better than it is. If you have any **class imbalance**, where one class has far more observations than the other, you can get a high accuracy by simply predicting the majority class all of the time! To get around the false sense of confidence that accuracy alone can promote, we can look at a few other metrics.

> **i** Accuracy Is Not Enough
>
> Accuracy is the first thing you see and the last thing that you trust! Seriously, accuracy alone should not be your sole performance metric unless you have a perfectly even split in the target! If you find yourself in a meeting where people are presenting their classification models and they only talk about accuracy, you should be wary. This is especially true when those accuracy values seem too good to be true. At the very least, always be ready to compare it to the baseline rate, or prevalence of the majority class.

Sensitivity/Recall/True positive rate

Sensitivity, also known as **recall** or the **true positive rate**, is the proportion of observed positives that are correctly predicted by the model. If your focus is on the positive class above all else, sensitivity is the metric for you.

Specificity/True negative rate

Specificity, also known as the **true negative rate**, is the proportion of observed negatives that are correctly predicted as such. If you want to know how well your model will work with the negative class, specificity is a great metric.

Precision/Positive predictive value

The **precision** is the proportion of positive predictions that are correct, and it is often a key metric in many business use cases. While similar to sensitivity, precision focuses on positive predictions, while sensitivity focuses on observed positive cases[5].

Negative predictive value

The **negative predictive value** is the proportion of negative predictions that are correct, and is the complement to precision.

Now let's demystify this a bit and see how we'd do this ourselves. Starting with a basic confusion matrix of counts, we'll then extract the values to create the metrics we need.

R

```
our_cm = rating_cm$matrix

TN = our_cm[2, 2]
TP = our_cm[1, 1]
FN = our_cm[2, 1]
FP = our_cm[1, 2]

acc = (TP + TN) / sum(our_cm)  # accuracy
tpr = TP / (TP + FN)           # true positive rate, sensitivity, recall
tnr = TN / (TN + FP)           # true negative rate, specificity
ppv = TP / (TP + FP)           # positive predictive value, precision
npv = TN / (TN + FN)           # negative predictive value
```

Python

```
our_cm = rating_cm

TN = our_cm[0, 0]
TP = our_cm[1, 1]
FN = our_cm[1, 0]
FP = our_cm[0, 1]

acc = (TP + TN) / np.sum(our_cm)  # accuracy
```

[5]It's not obvious why people started using terms like recall/sensitivity and specificity. Neither of them is as clear as true positive rate and true negative rate in conveying the underlying metric in our opinion. Why anyone thought 'precision' was a good name for any metric is beyond us, given how many metrics could generically be called as such.

```
tpr = TP / (TP + FN)              # true positive rate, sensitivity, recall
tnr = TN / (TN + FP)              # true negative rate, specificity
ppv = TP / (TP + FP)              # positive predictive value, precision
npv = TN / (TN + FN)              # negative predictive value
```

Now that we have a sense of some metrics, let's get a confusion matrix and stats using packages that will give us a lot of these metrics at once. In both cases we have an 0/1 integer where 0 is a rating of "bad" and 1 is "good".

R

```
tibble(
    metric = c('ACC', 'TPR', 'TNR', 'PPV', 'NPV'),
    ours = c(acc, tpr, tnr, ppv, npv),
    package = rating_cm$measures[c('acc', 'tpr', 'tnr', 'ppv', 'npv')]
)
```

Python

We find pycm to be a great package for this purpose, as practically every metric based on a confusion matrix you can think of is available. You can also use sklearn.metrics and its corresponding `classification_report` function.

```
from pycm import ConfusionMatrix

rating_cm = ConfusionMatrix(
    df_test.rating_good.to_numpy(),
    predicted_class.to_numpy(),
    digit = 3
)

# print(rating_cm) # lots of stats!

package_result = [
    rating_cm.class_stat[stat][1] # get results specific to class 1
    for stat in  ['ACC', 'TPR', 'TNR', 'PPV', 'NPV']
]

pd.DataFrame({
    'metric':['ACC', 'TPR', 'TNR', 'PPV', 'NPV'],
    'ours': [acc, tpr, tnr, ppv, npv],
    'package': package_result
})
```

So now we have demystified some classification metrics as well! Your results may be slightly different due to the random nature of the data splits, but they should be very similar to these and should match the package results regardless.

Table 4.3: Classification Results (based on R data/model)

metric	ours	package
ACC	0.804	0.804
TPR	0.842	0.842
TNR	0.757	0.757
PPV	0.812	0.812
NPV	0.792	0.792

Ideal decision points for classification

Earlier when we obtained the predicted class, and subsequently all the metrics based on it, we used a predicted probability value of 0.5 as a cutoff for a 'good' vs. a 'bad' rating, and this is usually the default if we don't specify it explicitly. Assuming that this is the best for a given situation is actually a bold assumption on our part, and we should probably make sure that the cut-off value we choose is going to offer us the best result given the modeling context.

But what is the *best* result? That's going to depend on the situation. If we are predicting whether a patient has a disease, we might want to minimize false negatives, since if we miss the diagnosis, the patient could be in serious trouble. Meanwhile if we are predicting whether a transaction is fraudulent, we might want to minimize false positives, since if we flag a transaction as fraudulent when it isn't, we could be causing a lot of trouble for the customer, and add cost to the company to deal with it. In other words, we might want to maximize the true positive or true negative rates, respectively.

Whatever we decide, we ultimately are just shifting the metrics around relative to one another. As an easy example, if we were to classify all of our observations as 'good', we would have a sensitivity of 1 because all good ratings would be classified correctly. However, our positive predictive value would not be 1, and we'd have a specificity of 0. No matter which cut-point we choose, we are going to have to make a tradeoff.

Where this comes into play is with **model selection**, where we choose a model based on a particular metric, and something we will talk about very soon. If we are comparing models based on accuracy, we might choose a different model than if we are comparing based on sensitivity. And given a particular threshold, we might choose a different model based on the same metric than we would have with a different threshold.

To help us with the task of choosing a threshold, we will start by creating what's called a **Receiver Operating Characteristic** (ROC) curve. This curve plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings. The **area under the curve** (AUC) is a measure of how well the model is able to distinguish between the two classes. The closer the AUC is to 1, the better the model is at distinguishing between the two classes. The AUC is a very popular metric because it is not sensitive to our threshold, and concerns two metrics we are routinely interested in[6].

R

```
roc = performance::performance_roc(model_class_train, new_data = df_test)
roc

# requires the 'see' package
plot(roc)
```

Python

```
from sklearn.metrics import roc_curve, auc, RocCurveDisplay

fpr, tpr, thresholds = roc_curve(
    df_test.rating_good,
    predicted_prob
)

auc(fpr, tpr)
```

```
RocCurveDisplay(fpr=fpr, tpr=tpr).plot()
```

[6]The **precision-recall curve** is a very similar approach which visualizes the tradeoff between precision and recall. The area under the precision-recall curve (AUPRC) is its corresponding metric.

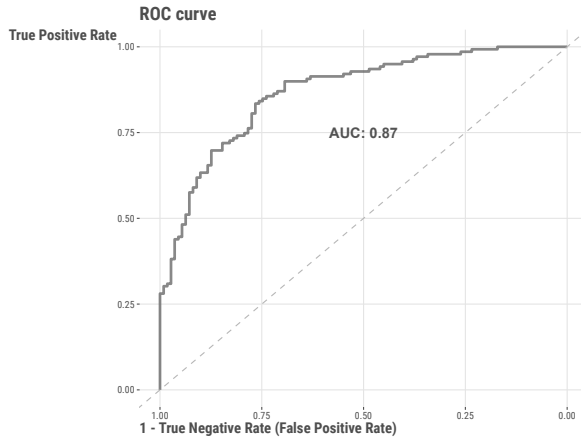


Figure 4.2: ROC curve and AUC value.

With ROC curves and AUC values in hand, now we can find the ideal cut-point for balancing the TPR and FPR. There are different ways to do this, but one common way is to use the Youden's J statistic, which we do here.

R

```
# produces the same value as before
roc_ = pROC::roc(df_test$rating_good, predicted_prob)
threshold = pROC::coords(roc_, "best", ret = "threshold")

predictions = ifelse(
    predict(model_class_train, df_test, type='response') >= threshold$threshold,
    1,
    0
)

cm_new = mlr3measures::confusion_matrix(
    factor(df_test$rating_good),
    factor(predictions),
    positive = "1"
)

tibble(
    threshold = threshold,
    TPR = cm_new$measures['tpr'],
    TNR = cm_new$measures['tnr']
)
```

```
# A tibble: 1 x 3
  threshold$threshold   TPR   TNR
                <dbl> <dbl> <dbl>
1               0.505 0.835 0.766
```

Python

```
cut = thresholds[np.argmax(tpr - fpr)]

pd.DataFrame({
    'threshold': [cut],
    'TPR': [tpr[np.argmax(tpr - fpr)]],
    'TNR': [1 - fpr[np.argmax(tpr - fpr)]],
})
```

```
   threshold   TPR   TNR
0      0.483 0.887 0.726
```

The result is a "best" decision cut-point for converting our predicted probabilities to classes, though again, there are different and equally valid ways of going about this. The take-home point is that instead of being naive about setting our probability to .5, this will provide a cut-point that will lead to a more balanced result that recognizes other metrics that are important beyond accuracy. We will leave it to you to take that ideal cut-point value and update your metrics to see how much of a difference it will make[7].

Note again that this only changes the metric values relative to one another, not the overall performance of the model - the actual predicted probabilities don't change after all. For example, accuracy may go down while recall increases. You'll need to match these metrics to your use case to see if the change is worth it. Whether it is a meager, modest, or meaningful improvement is going to vary from situation to situation, as will how you determine if your model is "good" or "bad". Is this a good model? Are you more interested in correctly identifying the positive class, or the negative class? Are you more interested in avoiding false positives/negatives? These are all questions that you will need to answer depending on the modeling context.

> **Multiclass Classification**
>
> The metrics we've discussed here are for binary classification, but there are also metrics for multiclass classification. The initial conversion from raw output to probability is done by a softmax function, which extends

[7] It is not lost on us that our R model actually chose ~ .5! But even then, you can see the slight difference in TPR/TNR.

the logistic/sigmoid function to multiple classes. So we get probabilities for each class, and can classify each observation to the class with the highest probability.
The confusion matrix is extended to show the number of correct and incorrect predictions for each class and the metrics are extended to show how well the model is able to distinguish between the classes. All metrics discussed have a multiclass variant, typically by averaging the metric across all classes, or by focusing on a particular class versus all other classes.

4.3 Model Selection and Comparison

Another important way to understand our model is by looking at how it compares to other models in terms of performance, however we choose to define that. One common way we can do this is by comparing models based on the metric(s) of our choice, for example, with RMSE or AUC. Let's see this in action for our regression model. Here we will compare three models: our original model, one with a subset of three features, and the three-feature model that includes interactions with genre. Our goal will be to see how these perform on the test set based on RMSE.

R

```
# create the models
model_lr_3feat = lm(
    rating ~ review_year_0 + release_year_0 + age_sc,
    df_train
)

model_lr_interact = lm(
    rating ~ review_year_0 * genre + release_year_0 * genre + age_sc * genre,
    df_train
)

model_lr_train = lm(
    rating ~
        review_year_0
        + release_year_0
        + age_sc
        + length_minutes_sc
```

```
        + total_reviews_sc
        + word_count_sc
        + genre
        ,
        df_train
)

# get the predictions, calculate RMSE
result = map(
    list(model_lr_3feat, model_lr_train, model_lr_interact),
    ~ predict(.x, newdata = df_test)
    ) |>
    map_dbl(
        ~ yardstick::rmse_vec(df_test$rating, .)
    )
```

Python

```
import statsmodels.formula.api as smf
from sklearn.metrics import root_mean_squared_error

# create the models
model_lr_3feat = smf.ols(
    'rating ~ review_year_0 + release_year_0 + age_sc',
    data=df_train
).fit()

model_lr_interact = smf.ols(
    'rating ~ review_year_0 * genre + release_year_0 * genre + age_sc * genre',
    data=df_train
).fit()

model_lr_train = smf.ols(
    formula=
    '''
    rating ~
    review_year_0
    + release_year_0
    + age_sc
    + length_minutes_sc
    + total_reviews_sc
    + word_count_sc
    + genre
```

```
    '''
    ,
    data=df_train
).fit()

models = [model_lr_3feat, model_lr_train, model_lr_interact]

# get the predictions, calculate RMSE
result = pd.DataFrame({
    'model': ['3 features', 'original', '3 feat+interact'],
    'rmse': [
        root_mean_squared_error(
            df_test.rating,
            model.predict(df_test[features])
        )
        for model in models
    ]
})
```

Table 4.4: RMSE for Different Models

model	rmse	% Δ RMSE[1]
original	0.46	27%
3 feat+interact	0.55	12%
3 features	0.63	0%

[1] % Δ is the percentage drop relative to the largest value.

In this case, the three-feature model does worst, but adding interactions of those features with genre improves the model. However, we see that our original model with seven features has the lowest RMSE, indicating that it is the best model *under these circumstances*. This suggests that the additional features have more to add to the model. This is a simple example, but it is a very typical way to compare models that you would use frequently. The same approach would work for classification models, just using an appropriate metric like AUC or F1.

Another thing to consider is that even with a single model, the model fitting procedure is always comparing a model with the current parameter estimates, or more generally the current objective function value, to a previous one with different parameter estimates. In this case, our goal is **model selection**, or how we choose the best result from a single model. While this is an automatic process, the details of how this actually happens is the focus of Chapter 6. In other cases, we are selecting models through the process of cross-validation

(Section 10.6), but the idea is largely the same in that we are comparing our current parameter estimates to other possibilities. We are always doing model selection and comparison, and as such, we'll be demonstrating these often.

4.4 Model Visualization

A key method for understanding how our model is performing is through visualization. You'll recall that we started out way back by look at the predicted values against the observed values to see if there was any correspondence (Figure 3.3), but another key way to understand our model is to look at the residuals, or errors in prediction, which again is the difference in our prediction versus the observed value. Here are a couple of plots that can help us understand our model:

- **Residuals vs. Fitted**: This type of plot shows predicted values vs. the residuals (or some variant of the residuals, like their square root). If you see a pattern, that potentially means your model is not capturing something in the data. For example, if you see a funnel shape, that would suggest that you are systematically having worse predictions for some part of the data. For some plots, patterns may suggest an underlying nonlinear relationship in the data is yet to be uncovered. For our main regression model, we don't see any patterns that would indicate that the model has a notable issue.

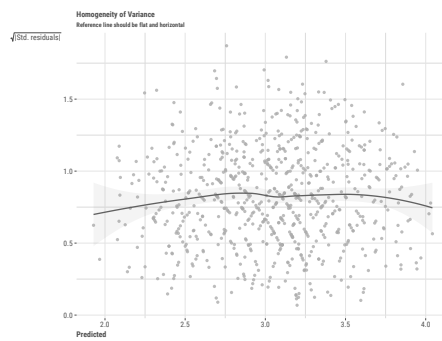


Figure 4.3: Residuals vs. fitted plot for our regression model.

- **Training/Test Performance**: For iterative approaches like deep learning, we may want to see how our model is performing across iterations, typically called "epochs". We can look at the training and testing performance to see if our model is overfitting or underfitting. We can actually do this with standard models as well if the estimation approach is iterative, but it's not as common. We can also visualize performance across samples of

the data, such as in cross-validation. The following shows performance for a model similar to the multilayer perceptron (MLP) demonstrated later (Section 11.7), and using the same features as our other models. Here we see it get to a relatively low objective function value after just a few epochs.

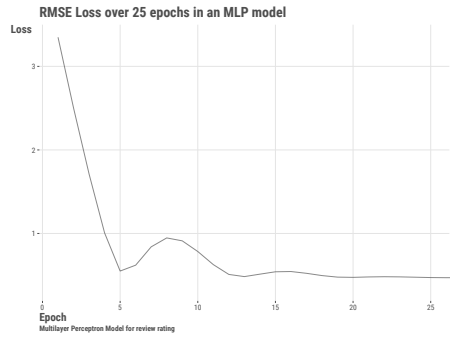


Figure 4.4: MSE loss over 25 epochs in an MLP model.

- **Predictive Check**: This is a basic comparison of predicted vs. observed target values. In the simplest case you can just examine your predictions vs. the observed values, and that's plenty for a quick assessment.

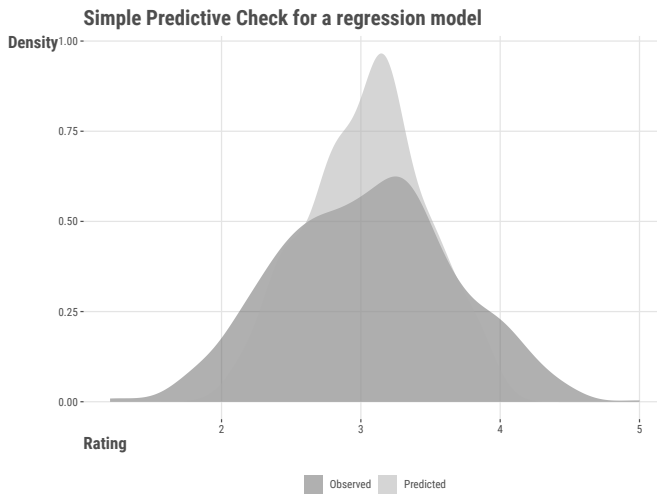


Figure 4.5: Predictive check for a regression model.

We may be getting ahead of ourselves to understand this completely yet, but it's worth knowing about ***posterior* predictive checks**, which are typically used with Bayesian models, but are not restricted to that case. A proper posterior

predictive check is a bit more involved, but there are packages that make it straightforward[8]. The basic idea is that we simulate the target based on the model parameter estimates and their uncertainty. And with that distribution of estimates (e.g., regression coefficients), we can then simulate random draws of predicted values. A step-by-step approach is as follows:

1. Simulate the parameters following the assumed distribution, e.g., a normal distribution for the regression coefficients.
2. For each simulated parameter set, make a model prediction.
3. Repeat this process many times to get a distribution of predictions.
4. Compare this distribution to the observed target distribution.

In our final step, we compare that distribution of predictions to the observed target distribution. If the two distributions are similar, then the model is doing a good job of capturing the target distribution.

This plot is ubiquitous in Bayesian modeling, but it can potentially be used for any model that has uncertainty estimates or is otherwise able to generate random draws of the target distribution. For our regression model, our predictions match the target distribution well.

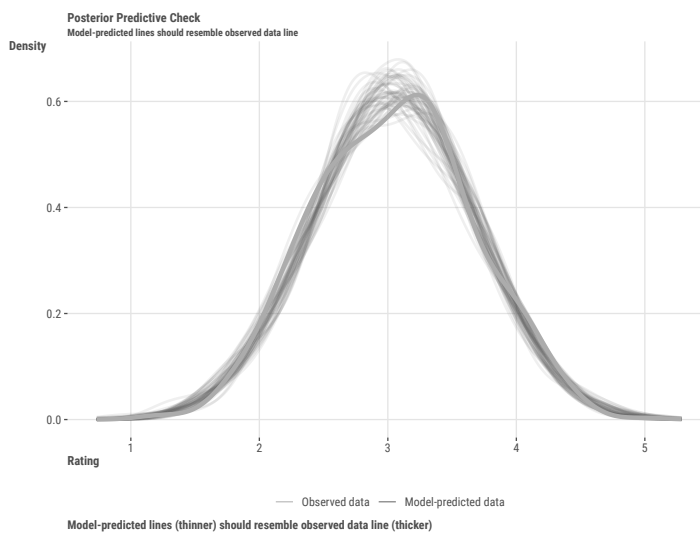


Figure 4.6: Bayesian posterior predictive check for a regression model.

[8]The performance package in R has a `check_predictions` function that can do this for you, and we used it here. Base R has a `simulate` function that can be used to generate random draws of the predictive distribution for `lm/glm` models. For Python, you'll need to use a package like arviz for Bayesian models, or write your own function for other models.

- **Other Plots**: Other plots may look at the distribution of residuals, check for extreme values, to see if there is an over-abundance of zero values, and other issues, some of which may be specific to the type of model you are using.

The following shows how to get a residuals vs. fitted plot and a posterior predictive check.

R

```
performance::check_model(model_lr_train, check = c('linearity', 'pp_check'))
```

Python

```
import seaborn as sns
import matplotlib.pyplot as plt

## Residual Plot
sns.residplot(
    x = model_lr_train.fittedvalues,
    y = df_train.rating,
    lowess = True,
    line_kws={'color': 'red', 'lw': 1}
)
plt.xlabel('Fitted values')
plt.ylabel('Residuals')
plt.title('Residuals vs. Fitted')
plt.show()


## Posterior Predictive Check
# get the model parameters
pp = model_lr_train.model.get_distribution(
    params = model_lr_train.params,
    scale  = model_lr_train.scale,
    exog   = model_lr_train.model.exog
)

# Generate 10 simulated predictive distributions
pp_samples = [pp.rvs() for _ in range(10)]

# Plot the distribution of pp_samples
for sample in pp_samples:
    sns.kdeplot(sample, label='pp.rvs()', alpha=0.25)
```

```
# Overlay the density plot of df_train.rating
sns.kdeplot(
    df_train.rating.to_numpy(),
    label='df_train.rating',
    linewidth=2
)

plt.xlabel('Rating')
plt.ylabel('Density')
plt.title('Distribution of predictions vs. observed rating')
plt.show()
```

i Tests of Assumptions

For standard GLM models there are an abundance of statistical tests available for some of these checks, for example heterogeneity of variance, or whether your residuals are normally distributed. On a practical level, these are not usually helpful and are often misguided. For example, if you have a large sample size, you will almost always reject the null hypothesis that your residuals are normally distributed. It also starts the turtles all the way down problem of whether you need to check the assumptions of your test of assumptions! We prefer the 'seeing is believing' approach. It is often pretty clear when there are model and data issues, and if it isn't, it's probably not a big deal.

4.5 Wrapping Up

It is easy to get caught up in the excitement of creating a model and then using it to make predictions. It is also easy to get caught up in the excitement of seeing a model perform well on a test set. It is much harder to take a step back and ask yourself, "Is this model really doing what I want it to do?" It takes a lot of work to trust what a model is telling you.

4.5.1 The common thread

Much of what you've seen in this chapter can be applied to any model. From linear regression to deep learning, we often use similar metrics to help select and compare models.

4.5.2 Choose your own adventure

Now that you've gotten a grasp of how to understand the model from a general perspective, you can focus on understanding more about how features relate to the target and make predictions. Head to Chapter 5 to find out more!

4.5.3 Additional resources

If this chapter has piqued your curiosity, we would encourage you to check out the following resources.

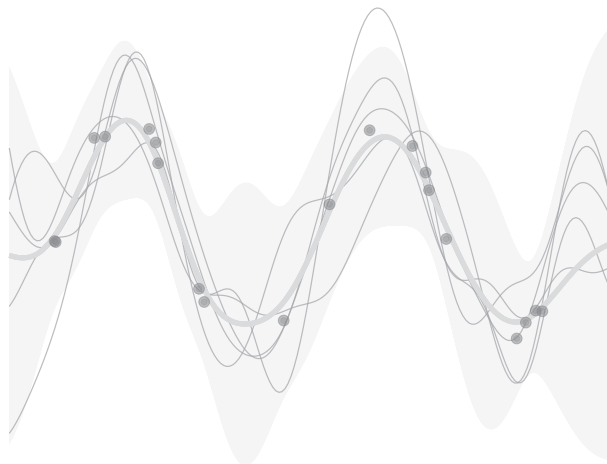
Even though we did not use the `mlr3` package in this chapter, the **Evaluation and Benchmarking** chapter of the associated book, Applied Machine Learning Using mlr3 in R, offers a great conceptual take on model metrics and evaluation.

For a more Pythonic look at model evaluation, we would highly recommend going through the sci-kit learn documentation on Model Evaluation. It has you absolutely covered with code examples and concepts.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

5

Understanding the Features



Assuming our model is adequate, let's now turn our attention to the features. There's a lot to unpack here, so let's get started!

5.1 Key Ideas

- There are many ways to get to know your features, and though it can be challenging, it can also be very rewarding!
- Visualizations can help you understand how your model is making predictions and which features are important.
- Feature importance is difficult to determine even in the simplest of model settings, but there are tools to help you understand how much each feature contributes to a prediction.

5.1.1 Why this matters

We often need to, or simply want to, know why our predictions come about. What's driving them? What happens for specific observations? By exploring the features, we can begin to better understand our model, and potentially improve it. We can also get a sense of what's important in our data, and what's not, and this can help us make better decisions.

5.1.2 Helpful context

We suggest having the linear model basics down pretty well, including basic logistic regression (Section 8.3), as much of what we explore here can be understood most easily in that setting. If you already have machine learning models down too, you're in good shape, as this can all apply in those modeling contexts as well.

5.2 Data Setup

We'll use the movie review data as with our previous chapters. Later on, we'll also use the world happiness dataset to explore some more advanced concepts. For the movie review data, we'll split the data into training and testing sets, and then fit a linear regression model and a logistic regression model to the training data. We'll then use the testing data to evaluate the models.

R

```
# all data found on github repo
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

set.seed(123)

initial_split = sample(
    x = 1:nrow(df_reviews),
    size = nrow(df_reviews) * .75,
    replace = FALSE
)

df_train = df_reviews[initial_split, ]
df_test = df_reviews[-initial_split, ]

model_lr_train = lm(
    rating ~
```

```
        review_year_0
        + release_year_0
        + age_sc
        + length_minutes_sc
        + total_reviews_sc
        + word_count_sc
        + genre
        ,
        df_train
)

model_class_train = glm(
    rating_good ~
        review_year_0
        + release_year_0
        + age_sc
        + length_minutes_sc
        + total_reviews_sc
        + word_count_sc
        + genre
        ,
        df_train,
        family = binomial
)
```

Python

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.model_selection import train_test_split

# all data found on github repo
df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

df_train, df_test = train_test_split(
    df_reviews,
    test_size = 0.25,
    random_state = 123
)
```

```
# we'll use this later
features = [
    "review_year_0",
    "release_year_0",
    "age_sc",
    "length_minutes_sc",
    "total_reviews_sc",
    "word_count_sc",
    "genre",
]

model =  'rating ~ ' + " + ".join(features)

model_lr_train = smf.ols(formula = model, data = df_train).fit()

model =  'rating_good ~ ' + " + ".join(features)

model_class_train = smf.glm(
    formula = model,
    data = df_train,
    family = sm.families.Binomial()
).fit()
```

5.3 Basic Model Parameters

We saw in the foundations chapter chapter (3.4.1) that we can get a lot out of the basic output from standard linear models like linear regression. Our starting point should be the coefficients or weights, which can give us a sense of the direction and magnitude of the relationship between the feature and the target given their respective scales. We can also look at the standard errors and confidence intervals to get a sense of the uncertainty in those estimates. Table 5.1 is a basic summary of the coefficients for our regression model on the training data.

Table 5.1: Coefficients for Our Regression Model

feature	estimate	std_error	statistic	p_value	conf_low	conf_high
intercept	2.44	0.08	32.29	0.00	2.29	2.59
review_year_0	0.01	0.00	2.65	0.01	0.00	0.01
release_year_0	0.01	0.00	5.29	0.00	0.01	0.01
age_sc	−0.05	0.02	−3.21	0.00	−0.09	−0.02
length_minutes_sc	0.19	0.02	10.40	0.00	0.15	0.22
total_reviews_sc	0.26	0.02	14.43	0.00	0.22	0.30
word_count_sc	−0.12	0.02	−6.94	0.00	−0.15	−0.08
genreComedy	0.53	0.06	8.26	0.00	0.40	0.65
genreDrama	0.58	0.04	13.50	0.00	0.50	0.67
genreHorror	0.00	0.08	0.03	0.98	−0.16	0.16
genreKids	0.07	0.07	1.05	0.30	−0.06	0.20
genreOther	0.03	0.07	0.36	0.72	−0.12	0.17
genreRomance	0.07	0.07	1.05	0.30	−0.06	0.21
genreSci-Fi	−0.01	0.08	−0.17	0.87	−0.18	0.15

We also noted how we can get a bit more relative comparison by using standardized coefficients, or some other scaling of the coefficients that allows for a bit of a more apples-to-apples comparison. But as we'll see, in the real world even if we have just apples, there are fuji, gala, granny smith, honeycrisp, and many other kinds, and some may be good for snacks, others for baking pies, some are good for cider, etc. In other words, **there is no one-size-fits-all approach to understanding how a feature contributes to understanding the target**, and the sooner you grasp that, the better.

5.4 Feature Contributions

We can also look at the contribution of a feature to the model's explanatory power, namely through its predictions. To start our discussion, we don't want to lean too heavily on the phrase **feature importance** yet, because as we'll see later, trying to rank features by an importance metric is difficult at best, and a misguided endeavor at worst. We can however look at the **feature contribution** to the model's predictions, and we can come to a conclusion about whether we think a feature is practically important, but we just need to be careful about how we do it.

Truly understanding feature contribution is a bit more complicated than just looking at the coefficient if we're using any model that isn't a linear regression, and there are many ways to go about it. We know we can't compare raw coefficients across features, because they are on different scales. But even when we put them on the same scale, it may be very easy for some features to move, e.g., one standard deviation, and very hard for others. Binary features can only be on or off, while numeric features can move around more, but numeric

features may also be highly skewed. We also can't use statistical significance based on p-values, because they reflect sample size as much or more than effect size.

So what are we to do? What you need to know to get started looking at a feature's contribution includes the following:

- feature range and variability
- feature distributions (e.g., skewness)
- representative values of the feature
- target range and variability
- feature interactions and correlations

We can't necessarily do a whole lot about these aspects, but we can at least be aware of them to help us understand any effects we want to explore. And just as importantly, knowing this sort of information can help us be aware of the limitations of our understanding of these effects. In any case, let's try to get a sense of how we can understand the contribution of a feature to our model.

5.5 Marginal Effects

One way to understand the contribution of a feature to the model is to look at the **marginal effect** of the feature, which conceptually attempts to boil a feature effect to something simple like an average. Unfortunately, not everyone means the same thing when they use this term and it can be a bit confusing. Marginal effects *typically* refer to a partial derivative of the target with respect to the feature. Oh no! Math! However, as an example, this becomes very simple for standard linear models with no interactions and all linear effects, as in linear regression. The derivative of our coefficient with respect to the feature is just the coefficient itself! But for more complicated models, even just a classification model like our logistic regression which is nonlinear on the probability scale, we need to do a bit more work to get the marginal effect, or other so-called *average* effects. Let's think about a couple of common versions:

- Marginal effect at the mean
- Average Marginal Effect, or Average slope
- Marginal Means (for categorical features)
- Counterfactuals and other predictions at key feature values

Note that if you need an introduction or refresher on logistic regression, you'll find it in Section 8.3.

5.5.1 Marginal effects at the mean

First let's think about an average slope. This is the average of the slopes across the feature's values, or values of another feature it interacts with. To begin, let's just look at the effect of word count first, and we'll do this for the logistic regression model to make things more interesting. A good question to start with is, how do we visualize the relationship/effect?

Figure 5.1 shows two plots, and both are useful, neither is inherently wrong, and yet they both tell us something different. The first plot shows the predicted probability of a good review as word count changes, *with all other features at their mean* (or mode for categorical features like genre). We can see that on the probability scale, this relationship is negative with a bit of a curve that gets steeper with higher word count values.

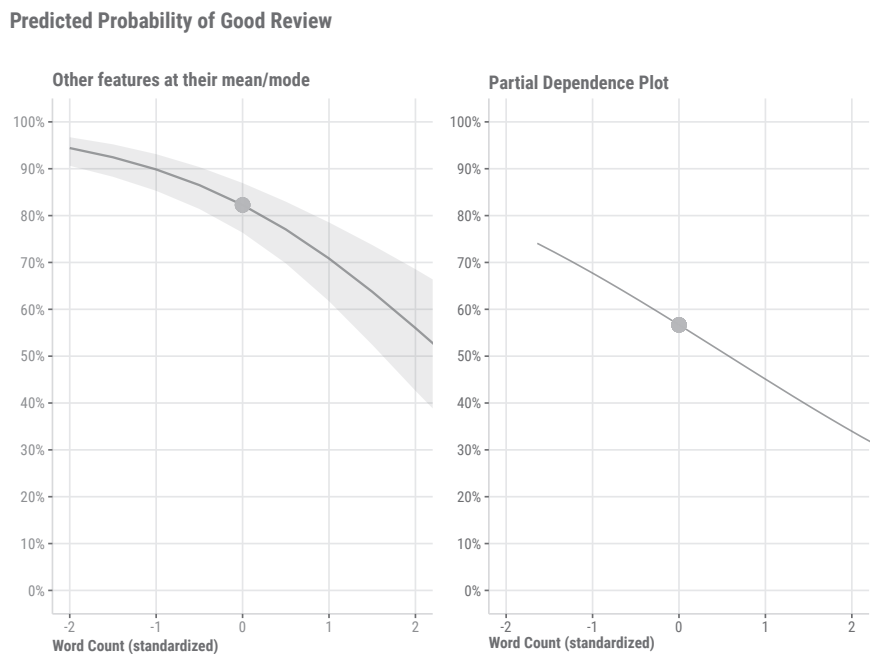


Figure 5.1: Marginal effect at the mean vs. partial dependence plot.

The second plot shows what is called a **partial dependence plot**, which shows the *average predicted probability* of a good review as word count changes. In both cases we make predictions with imputed values. The left plot imputes the other features to be their mean or mode, while the right plot leaves the other features at their actual values, and then, using a range of values for word count, gets a prediction as if every observation had that value for word count. We then plot the average of the predictions for each value in the range.

Let's demystify this by calculating it ourselves.

R

```
df_typical = tibble(
    genre = names(which.max(table(df_train$genre))),
    age_sc = mean(df_train$age_sc),
    release_year_0 = mean(df_train$release_year_0),
    length_minutes_sc = mean(df_train$length_minutes_sc),
    total_reviews_sc = mean(df_train$total_reviews_sc),
    word_count_sc = mean(df_train$word_count_sc),
    review_year_0 = mean(df_train$review_year_0)
)

# avg prediction when everything is typical
avg_pred = predict(model_class_train, newdata = df_typical, type = "response") |>
    mean()

# avg prediction when word count is at its mean
avg_pred_0 = predict(
    model_class_train,
    newdata = df_train |> mutate(word_count_sc = 0),
    type = 'response'
) |>
    mean()

c(avg_pred, avg_pred_0)
```

```
[1] 0.8212 0.5668
```

Python

```
df_typical = pd.DataFrame({
    "genre": df_train.genre.mode().values,
    "age_sc": df_train.age_sc.mean(),
    "release_year_0": df_train.release_year_0.mean(),
    "length_minutes_sc": df_train.length_minutes_sc.mean(),
    "total_reviews_sc": df_train.total_reviews_sc.mean(),
    "word_count_sc": df_train.word_count_sc.mean(),
    "review_year_0": df_train.review_year_0.mean()
})

# avg prediction when everything is typical
avg_pred_typical = model_class_train.predict(df_typical).mean()
```

```
# avg prediction when word count is at its mean
avg_pred_0 = model_class_train.predict(
    df_train.assign(word_count_sc = 0)
).mean()

avg_pred_typical.round(3), avg_pred_0.round(3)
```

```
(0.83, 0.575)
```

When word count is zero, i.e., its mean, and everything else is at its mean/mode, we'd predict a chance of a good review at about 82%. The average prediction we'd get if we predicted every observation as if it were the mean word count is more like 57%, which is notably less. Which is correct? Both, or neither! They are telling us different things, either of which may be useful, or not. If it's doubtful that the feature values used in the calculation are realistic, e.g., everything at its mean at the same time, or an average word count when length of a movie is at its minimum, then they may both be misleading. You have to know your features and your target to know how best to use the information.

5.5.2 Average marginal effects

Let's say we want to distill our understanding of the feature-target relationship to a single number. In this case, the coefficient is fine if we're dealing with an entirely linear model. In this classification case, the raw coefficient tells us what we need to know, but on the log-odds scale, which is not very intuitive for most folks. We can understand the probability scale, but this means things get nonlinear. As an example, a .1 to .2 change in the probability is doubling it, while a .8 to .9 change is a 12.5% increase in the probability. But is there any way we can stick with probabilities and get a single value to understand the change in the probability of a good review as word count changes by 1 unit?

Yes! We can look at what's called the **average marginal effect** of word count. This is the average of the slope of the predicted probability of a good review as word count changes. This is a bit more complicated than just looking at the coefficient, but it's still intuitive, and more so than a coefficient that regards odds. How do we get it? By a neat little trick where we predict the target with the feature at two values. We start with the observed value and then add or subtract a very small amount. Then we take the difference in the predictions for those two feature values. This results in the same thing as taking the derivative of the target with respect to the feature.

R

```
fudge_factor = 1e-3

fudge_plus = predict(
    model_class_train,
    newdata = df_train |> mutate(word_count_sc = word_count_sc + fudge_factor/2),
    type = "response"
)

fudge_minus = predict(
    model_class_train,
    newdata = df_train |> mutate(word_count_sc = word_count_sc - fudge_factor/2),
    type = "response"
)

# compare
# mean(fudge_plus - fudge_minus) / fudge_factor

marginaleffects::avg_slopes(
    model_class_train,
    variables = "word_count_sc",
    type = 'response'
)
```

```
         Term Estimate Std. Error     z Pr(>|z|)    S  2.5 %  97.5 %
 word_count_sc   -0.105     0.0155 -6.74   <0.001 35.8 -0.135 -0.0742

Columns: term, estimate, std.error, statistic, p.value, s.value, conf.low, conf.high
Type:  response
```

Python

```
fudge_factor = 1e-3

fudge_plus = model_class_train.predict(
    df_train.assign(
        word_count_sc = df_train.word_count_sc + fudge_factor/2
    )
)

fudge_minus = model_class_train.predict(
    df_train.assign(
```

```
        word_count_sc = df_train.word_count_sc - fudge_factor/2
    )
)

np.mean(fudge_plus - fudge_minus) / fudge_factor
```

```
-0.09447284318194568
```

```
# note that the marginaleffects is available in Python, but still very fresh!
# we'll add a comparison in the future, but it doesn't handle models
# with categoricals right now.

# import marginaleffects as me
# me.avg_slopes(model_class_train, variables = "word_count_sc")
```

Our result suggests we're getting about a -0.1 drop in the expected probability of a good review for a 1 standard deviation increase in word count on average. This is a bit more intuitive than the coefficient or odds ratio based on it, and we probably don't want to ignore that sort of change. It also doesn't take much to get with the right package, or even on our own. Another nice thing about this approach is that it can potentially be applied to any model, including ones that don't normally produce coefficients, like gradient boosting models or deep learning models.

5.5.3 Marginal means

Marginal means are just about getting an average prediction for the levels of *categorical features*. As an example, we can get the average predicted probability of a good review for each level of the genre feature, and then compare them. To do this, we just have to make predictions as if every observation had a certain value for genre, and then we average the predictions. This is also the exact same approach that produced the PDP for word count we saw earlier.

R

```
marginal_means = map_df(
    unique(df_train$genre),
    ~ tibble(
        genre = .x,
        avg_pred = predict(
            model_class_train,
            newdata = df_train |>
                mutate(genre = .x),
```

```
            type = "link"
        ) |>
            mean() |> # get the average log odds then transform
            plogis()
    )
)

# marginal_means
marginaleffects::avg_predictions(
    model_class_train,
    newdata = df_train,
    variables = "genre"
)
```

Python

```
inv_logit = lambda x: 1 / (1 + np.exp(-x))

marginal_means = pd.DataFrame({
    "genre": df_train.genre.unique(),
    "avg_pred": [
        inv_logit(
            model_class_train.predict(df_train.assign(genre = g), which='linear')
        ).mean()
        for g in df_train.genre.unique()
    ]
})

marginal_means
```

Table 5.2: Average Predicted Probability of a Good Review by Genre

genre	estimate	conf.low	conf.high
Comedy	0.85	0.72	0.92
Drama	0.82	0.76	0.87
Action/Adventure	0.38	0.31	0.46
Horror	0.37	0.22	0.56
Sci-Fi	0.45	0.26	0.64
Romance	0.49	0.33	0.65
Other	0.39	0.24	0.55
Kids	0.41	0.27	0.57

Select output from the R package marginaleffects.

These results suggest that we can expect a good review for Comedy and Drama, maybe not so much for the others, *on average*. It also seems that our probability of a good review is similar for Comedy and Drama, and the others kind of group together as well.

> **i** That's Not What I Got!
>
> Just a reminder that if you used a different seed for your split or are using Python vs. R, you will see slightly different results. This is normal and expected!

5.6 Counterfactual Predictions

The nice thing about having a model is that we can make predictions for any set of feature values we want to explore. This is a great way to understand the contribution of a feature to the model. We can make predictions for a range of feature values, and then compare the predictions to see how much the feature contributes to the model. **Counterfactual predictions** allow us to ask "what if?" questions of our model, and see how it responds. As an example, we can get a prediction as if every review was made for a drama, and then see what we'd expect if every review pertained to a comedy. This is a very powerful approach, and often utilized in causal inference, but it's also a great way to understand the contribution of a feature to a model in general.

Consider an experimental setting where we have lots of control over how the data is produced for different scenarios. Ideally we'd be able to look at the same instances in a setting where everything about them was identical, but in one case, the instance was part of the control group, and in another, part of the treatment group. Unfortunately, not only is it impossible to have *everything* be identical, but it's also impossible to have the same instance be in two experimental group settings at the same time! Counterfactual predictions are the next best thing though, because once we have a model, we can predict an observation *as if* it was in the treatment, and then when it is a control. If we do this for all observations, we can get a sense of the **average treatment effect**, one of the main points of interest in causal inference.

But you don't need an experiment for this. In fact, we've been doing this all along with our marginal effects examples. Take the marginal means, where we looked at the average predicted probability of a good review as if every observation was a specific genre. We could have also looked at any specific observation's predictions to compare the two genre settings, instead of getting an average. This is very much in the spirit of a counterfactual prediction.

Let's try a new dataset to help drive the point home. We'll use some data at the global stage: the world happiness dataset (Section C.2). For our model we'll predict the happiness score for a country, considering freedom to make life choices, GDP and other things associated with it. We'll then switch the freedom to make life choices and GDP values for the US and Russia, and see how the predictions change!

R

```
# data available on repo (full link in appendix)
df_happiness_2018 = read_csv('https://tinyurl.com/worldhappiness2018')

model_happiness = lm(
    happiness_score ~
    log_gdp_per_capita
    + healthy_life_expectancy_at_birth
    + generosity
    + freedom_to_make_life_choices
    + confidence_in_national_government,
    data = df_happiness_2018
)

df_us_russia = df_happiness_2018 |>
    filter(country %in% c('United States', 'Russia'))

happiness_gdp_freedom_values = df_us_russia  |>
    arrange(country) |>
    select(log_gdp_per_capita, freedom_to_make_life_choices)

base_predictions = predict(
    model_happiness,
    newdata = df_us_russia
)

# switch up their GDP and freedom!
df_switch = df_us_russia |>
    mutate(
        log_gdp_per_capita = rev(log_gdp_per_capita),
        freedom_to_make_life_choices = rev(freedom_to_make_life_choices)
    )

switch_predictions = predict(
    model_happiness,
    newdata = df_switch
```

```
)

tibble(
    country = c('Russia', 'USA'),
    base_predictions,
    switch_predictions
) |>
    mutate(
        diff_in_happiness = switch_predictions - base_predictions
    )
```

Python

```
# data available on repo (full link in appendix)
df_happiness_2018 = pd.read_csv('https://tinyurl.com/worldhappiness2018')

model_happiness = smf.ols(
    formula = 'happiness_score ~ \
        log_gdp_per_capita \
        + healthy_life_expectancy_at_birth \
        + generosity \
        + freedom_to_make_life_choices \
        + confidence_in_national_government',
    data = df_happiness_2018
).fit()

df_us_russia = df_happiness_2018[
        df_happiness_2018.country.isin(['United States', 'Russia'])
    ]

happiness_gdp_freedom_values = df_happiness_2018.loc[
    df_happiness_2018.country.isin(['United States', 'Russia']),
    ['log_gdp_per_capita', 'freedom_to_make_life_choices']
]

base_predictions = model_happiness.predict(df_us_russia)

# switch up their GDP and freedom!
df_switch = df_us_russia.copy()

df_switch[['log_gdp_per_capita', 'freedom_to_make_life_choices']] = df_switch[
    ['log_gdp_per_capita', 'freedom_to_make_life_choices']
].values[::-1]
```

```
switch_predictions = model_happiness.predict(df_switch)

pd.DataFrame({
    'country': ['Russia', 'USA'],
    'base_predictions': base_predictions,
    'switch_predictions': switch_predictions,
    'diff_in_happiness': switch_predictions - base_predictions
}).round(3)
```

Here are the results in a clean table.

Table 5.3: Counterfactual Predictions for Happiness Score

country	base_predictions	switch_predictions	diff_in_happiness
Russia	5.7	6.4	0.7
United States	6.8	6.1	−0.7

In this case, we see that the happiness score is expected to be very lopsided in favor of the US, which our base prediction would suggest the US to be almost a full standard deviation higher in happiness than Russia given their current values (SD happiness ~1.1). But if the US was just a bit more like Russia, we'd see a significant drop even if it maintained its life expectancy, generosity, and faith in government. Likewise, if Russia was a bit more like the US, we'd expect to see a significant increase in their happiness score.

It's very easy even with base package functions to see some very interesting things about our data and model. As an exercise, you might go back to the movie reviews data and see what happens if we switch the age of reviewer and length of a movie for a few observations. Counterfactual predictions get us thinking more explicitly about what the situation would be if things were much different, but in the end, we're just playing around with predicted values and thinking about possibilities!

5.7 SHAP Values

As we've suggested, most models are more complicated than can be explained by a simple coefficient, for example, nonlinear effects in generalized additive models. Or, there may not even be feature-specific coefficients available, like gradient boosting models. Or, we may even have many parameters associated with a feature, as in deep learning. Such models typically won't come with

statistical output like standard errors and confidence intervals either. But we'll still have some tricks up our sleeve to help us figure things out!

A very common interpretation tool is called a **SHAP value**. SHAP stands for **SHapley Additive exPlanations**, and it provides a means to understand how much each feature contributes to a specific prediction. It's based on a concept from game theory called the Shapley value, which is a way to understand how much each player contributes to the outcome of a game. For our modeling context, SHAP values break down a prediction to show the impact of each feature. The reason we bring it up here is that it has a nice intuition in the linear model case, and seeing it in that context is a good way to get a sense of how it works. Furthermore, it builds on what we've been talking about with our various prediction approaches.

While the actual computations behind the scenes can be tedious, the basic idea is relatively straightforward. For a given prediction at a specific observation with set feature values, we can calculate *the difference between the prediction at that observation versus the average prediction for the model as a whole*. We can break this down for each feature, and see how much each contributes to the difference. This provides us the **local effect** of the feature, or how it plays out for a specific observation, as opposed to the whole data. The SHAP approach also has the benefit of being able to be applied to *any* model, whether a simple linear or deep learning model. Very cool! To demonstrate, we'll use the simple model from our model comparison demo in the previous chapter but keep the features on the raw scale.

R

```
model_lr_3feat = lm(
    rating ~
    age
    + release_year
    + length_minutes,
    data = df_reviews
)

# inspect if desired
# summary(model_lr_3feat)
```

Python

```
model_lr_3feat = smf.ols(
    formula = 'rating ~ \
    age \
    + release_year \
```

```
    + length_minutes',
    data = df_reviews
).fit()

# inspect if desired
# model_lr_3feat.summary(slim = True)
```

With our model in place, let's look at the SHAP values for the features. We'll start by choosing the instance we want to explain. Here we'll consider an observation where the release year is 2020, age of reviewer is 30, and a movie length of 110 minutes. To aid our understanding, we calculate the SHAP values at that observation by hand, and using a package. The by-hand approach consists of the following steps.

1. Get the average prediction for the model
2. Get the prediction for the feature at the value of interest for all observations, and average the predictions
3. Calculate the SHAP value as the difference between the average prediction and the average prediction for the feature value of interest

Note that this approach only works for our simple linear regression case, and we'd need to use a package incorporating an appropriate method for more complicated settings. But this simplified approach helps get our bearings on what SHAP values tell us. Also, be aware that our focus is a feature's *marginal contribution* at a *single observation*. Our coefficient already tells us the average contribution of a feature across all observations for this linear regression setting, i.e., the average marginal effect discussed previously.

R

```
# first we need to get the average prediction
avg_pred = mean(predict(model_lr_3feat))

# observation of interest we want shap values for
obs_of_interest = tibble(
    age = 30,
    length_minutes = 110,
    release_year = 2020
)

# then we need to get the prediction for the feature value of interest
# for all observations, and average them
pred_age_30 = predict(
```

```
    model_lr_3feat,
    newdata = df_reviews |> mutate(age = obs_of_interest$age)
)

pred_year_2022 = predict(
    model_lr_3feat,
    newdata = df_reviews |>
        mutate(release_year = obs_of_interest$release_year)
)

pred_length_110 = predict(
    model_lr_3feat,
    newdata = df_reviews |>
        mutate(length_minutes = obs_of_interest$length_minutes)
)

# then we can calculate the shap values
shap_value_ours = tibble(
    age = mean(pred_age_30) - avg_pred,
    release_year = mean(pred_year_2022) - avg_pred,
    length_minutes = mean(pred_length_110) - avg_pred
)
```

Python

```
# first we need to get the average prediction
avg_pred = model_lr_3feat.predict(df_reviews).mean()

# observation of interest we want shap values for
obs_of_interest = pd.DataFrame({
    'age': 30,
    'release_year': 2020,
    'length_minutes': 110
}, index = ['new_observation'])

# then we need to get the prediction for the feature value of interest
# for all observations, and average them
pred_dat = df_reviews.assign(
    age = obs_of_interest.loc['new_observation', 'age']
)
pred_age_30 = model_lr_3feat.predict(pred_dat)

pred_dat = df_reviews.assign(
```

```
    release_year = obs_of_interest.loc['new_observation', 'release_year']
)
pred_year_2022 = model_lr_3feat.predict(pred_dat)

pred_dat = df_reviews.assign(
    length_minutes = obs_of_interest.loc['new_observation', 'length_minutes']
)
pred_length_110 = model_lr_3feat.predict(pred_dat)

# then we can calculate the shap values
shap_value_ours = pd.DataFrame({
    'age': pred_age_30.mean() - avg_pred,
    'release_year': pred_year_2022.mean() - avg_pred,
    'length_minutes': pred_length_110.mean() - avg_pred
}, index = ['new_observation'])
```

Now that we have our own part set up, we can use a package to do the work more formally, and compare the results.

R

```
# we'll use the DALEX package for this
explainer = DALEX::explain(model_lr_3feat, verbose = FALSE)

shap_value_package = DALEX::predict_parts(
    explainer,
    obs_of_interest,
    type = 'shap'
)

rbind(
    shap_value_ours,
    shap_value_package[
        c('age', 'release_year', 'length_minutes'),
        'contribution'
    ]
)
```

Python

```
# now use the shap package for this; it does not work with statsmodels though,
# but we still get there in the end!
import shap
from sklearn.linear_model import LinearRegression

# set data up for shap and sklearn
fnames = [
    'age',
    'release_year',
    'length_minutes'
]

X = df_reviews[fnames]
y = df_reviews['rating']

# use a linear model that works with shap
model_reviews = LinearRegression().fit(X, y)

# 1000 instances for use as the 'background distribution'
X_sample = shap.maskers.Independent(data = X, max_samples = 1000)

# # compute the SHAP values for the linear model
explainer = shap.Explainer(
    model_reviews.predict,
    X_sample
)

shap_values = explainer(obs_of_interest)

shap_value_package = pd.DataFrame(
    shap_values.values[0, :],
    index = fnames,
    columns = ['new_observation']
).T

pd.concat([shap_value_ours, shap_value_package])
```

The following table reveals that the results are identical.

Table 5.4: SHAP Value Comparison

source	age	release_year	length_minutes
Ours	0.063	0.206	−0.141
Package	0.063	0.206	−0.141

We can visualize these as well, via a **force plot** or **waterfall plot**, the latter of which is shown in the next plot. The dotted line at *E[f(x)]* represents the average prediction from our model (~3.05), and the prediction we have for the observation at *f(x)*, which is about 3.18.

With the average prediction as our starting point, we add the SHAP values for each feature to get the prediction for the observation. First we add the SHAP value for age, which bumps the value by 0.063, then the SHAP value for movie length, which decreases the prediction -0.141, and finally the SHAP value for release year, which brings us to the final predicted value by increasing the prediction 0.206.

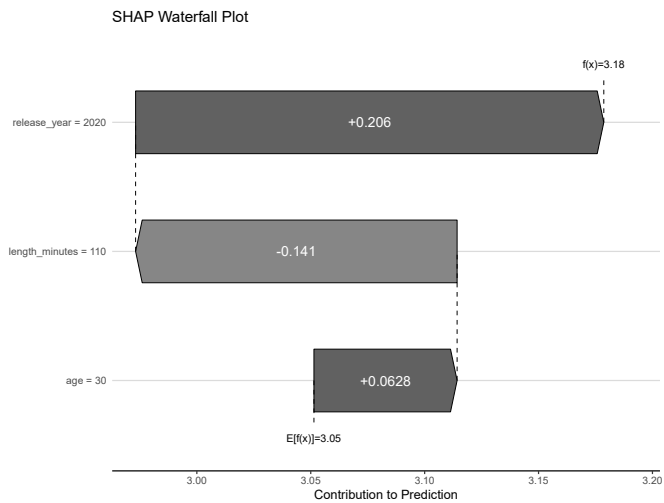


Figure 5.2: SHAP waterfall plot.

And there you have it- we've demystified the SHAP value! Things get more complicated in nonlinear settings, dealing with correlated features, and other cases, but hopefully this provides you some context. SHAP values are useful because they tell us how much each feature contributes to the prediction for the observation under consideration.

Pretty neat, huh? So for any observation we want to inspect, and more importantly, for any model we might use, we can get a sense of how features

contribute to that prediction. We also can get a sense of how much each feature contributes to the model as a whole by aggregating these values across all observations in our data, and this potentially provides a measure of **feature importance**, but we'll come back to that in a bit.

5.8 Related Visualizations

We've seen how we can get some plots for predictions in different ways previously with what's called a **partial dependence plot**, or PDP (Figure 5.1). In essence, a PDP shows the average prediction of a feature on the target across the feature values. This is also what we were just doing to calculate our SHAP value, and for the linear case, the PDP has a direct correspondence to the SHAP. In this setting, the SHAP value is the difference between the average prediction and the point on the PDP for a feature at a specific feature value. As an example, for a movie of 110 minutes, the line in the plot below corresponds to the value in the previous waterfall plot (Figure 5.2).

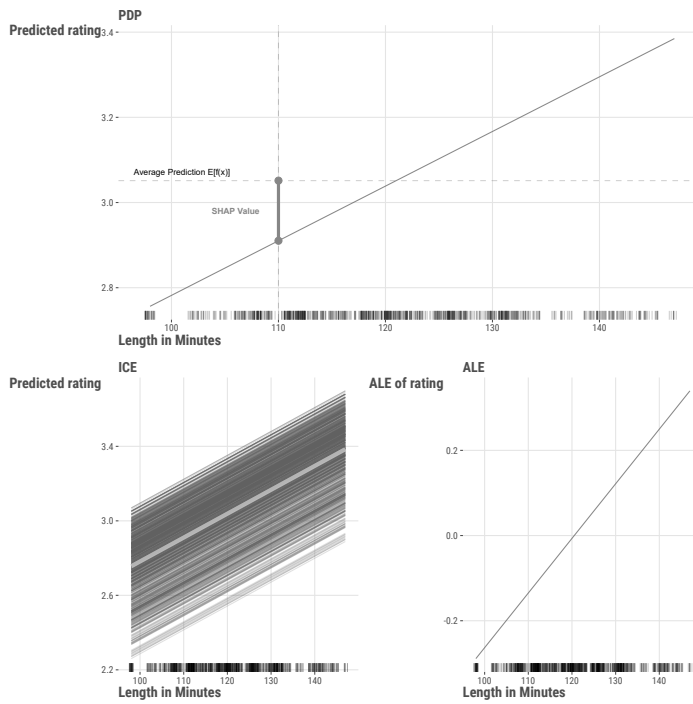


Figure 5.3: PDP, ICE, and ALE plots.

We can also look at the **individual conditional expectation** (ICE) plot, which is a PDP for a single observation, but across all values of a select feature. By looking at several observations, as in the second plot, we can get a sense of the variability in the feature's effect. As we can see, there is not much to tell beyond a PDP when we have a simple linear model, but it becomes more interesting when we have interactions or other nonlinear relationships in our model.

In addition, there are other plots that are similar to the PDP and ICE, such as the **accumulated local effect** (ALE) plot, shown last, which is more robust to correlated features than the PDP, while also showing the general feature-target relationship. Where the PDP and ICE plots show the average effect of a feature on the target, the ALE plot focuses on average *differences* in predictions for the feature at a specific value, versus predictions at feature values nearby, and then centers the result so that the average difference is zero. In general, all our plots reflect the positive linear relationship between movie length and rating.

> **i** Visualization Tools
>
> The waterfall plot was created using DALEX, but the shap Python package will also provide this. For PDP, ICE and ALE plots in R, you can look to the iml package, and in Python, the shap or scikit-learn package. Many others are available though, so feel free to explore!

5.9 Global Assessment of Feature Importance

Everything we've shown so far provides specific information about how a feature impacts a prediction for a specific observation. But more generally we often will ask: how important is a feature? It's a common question, and one that is often asked of models, but the answer ranges from 'it depends' and 'it doesn't matter'. Let's start with some hard facts:

- There is no single definition or metric of importance for any given model.
- There is no single metric for any model that will conclusively tell you how important a feature is relative to others in all data/model contexts.
- There are multiple metrics of importance for a given model that are equally valid, but which may come to different conclusions.
- Any non-zero feature contribution is potentially 'important', however small.
- Many metrics of importance fail to adequately capture interactions and/or deal with correlated features.

- All measures of importance are measured with uncertainty, and the uncertainty can be large.
- A question for feature importance is *relative to… what?* A poor model will still have relatively 'important' features, but they still may not be useful since the model itself isn't.
- It rarely makes sense to drop features based on importance alone, and doing so will typically drop performance as well.
- In the end, what will you do with the information?

As we noted previously, if I want to know how a feature relates to a target, I have to know how a feature moves, and I need to know what types of feature values are more likely than others, and what a typical movement in its range of values would be. If a feature is skewed, then the mean may not be the best value to use for prediction, and basing 'typical' movement on its standard deviation may be misguided. If a unit movement in a feature results in a movement in the target of 2 units, what does that mean? Is it a large movement? If I don't know the target very well, I can't answer that. As an example, if the target is in dollars, a $2 movement is nothing for salary, but might be large for a stock price. We have to know the target as well as we do the feature predicting it.

On top of all this, we need to know how the feature interacts with other features. If a feature is highly correlated with another feature, then it may not be adding much to the model even if some metrics would indicate a notable contribution. In addition, some approaches will either spread the contribution of correlated features across them, or just pick one of them as being important to the model. It may be mostly arbitrary which one is included, or you might miss both if the importance values are split.

If a feature interacts with another feature, then there really is no way to say how much it contributes to the model without knowing the value of the other feature. Full stop. Synergistic effects cannot be understood by pretending they don't exist. A number of metrics will still be provided for a single feature, either by trying to include its overall contribution or averaging over the values of the other feature. But this is a problematic approach because it still ignores the other feature values. As an example, if a drug doesn't work for your age group or for someone with your specific health conditions, do you really care if it works 'in general' or 'on average'?

To help us further understand this issue, consider the following two plots in Figure 5.4. On the left we show an interaction between two binary features. If we were to look at the contribution of each feature without the interaction, their respective coefficients would be estimated as essentially zero[1]. On the right we show a feature that has a strong relationship with the target, but

[1]To understand why, for the effect of Feature 1, just take the mean of the two points on the left vs. the mean of the two points on the right. It would basically be a straight line of no effect as you move from group 0 to group 1. For the effect of Feature 2, the two group means for A and B would be at the intersection of the two lines.

only for a certain range of values. If we were to look at a single 'effect' of the feature, we would likely underestimate how strong it is with smaller values and overestimate the relationship at the upper range.

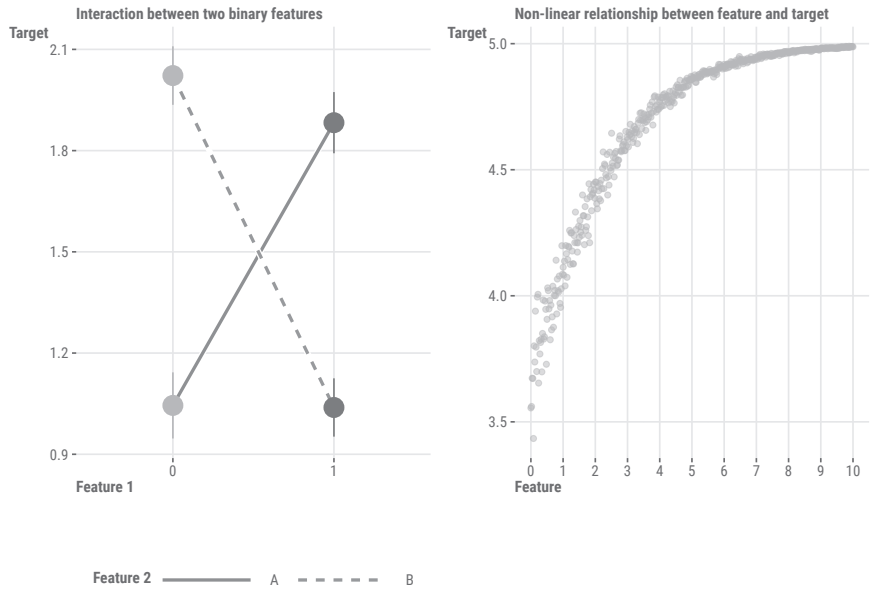


Figure 5.4: Two plots showing the importance of understanding feature interactions and nonlinear relationships.

All this is to say, as we get into measures of feature importance, we need to be very careful about how we interpret and use them!

5.9.1 Example: Feature importance for linear regression

To show just how difficult measuring feature importance is, we only have to stick with our simple linear regression. Think again about R^2: it tells us the proportion of the target explained by our features. An ideal measure of importance would be able to tell us how much each feature contributes to that proportion, or in other words, one that *decomposes* R^2 into the relative contributions of each feature. One of the most common measures of importance in linear models is the **standardized coefficient** we have demonstrated previously. You know what it doesn't do? It doesn't decompose R^2 into relative feature contributions. Even the more complicated SHAP approach will not do this.

The easiest situation we could hope for with regard to feature importance is the basic linear regression model we've been using. Everything is linear, with no interactions or other things going on, as in our demonstration model. And yet there are many logical ways to determine feature importance. Some metrics even break down R^2 into relative feature contributions, but they won't necessarily agree with each other in ranking or relative differences. If you can get a measure of statistical difference between whatever metric you choose, it's often the case that 'top' features will not be statistically different from other features. So what do we do? We'll show a few methods here, but the main point is that there is no single answer, and it's important to understand what you're trying to do with the information.

Let's start things off by using one of our previous linear regression models with several features, but which has no interactions or other complexity (Section 3.5.1). It's just a model with simple linear relationships and nothing else. We even remove categorical features to avoid having to aggregate group effects. In short, it doesn't get any easier than this!

R

```
model_importance = lm(
    rating ~
        word_count
        + age
        + review_year
        + release_year
        + length_minutes
        + children_in_home
        + total_reviews,
    data = df_reviews
)
```

Python

```
model_importance = smf.ols(
    formula = 'rating ~ \
        word_count \
        + age \
        + review_year \
        + release_year \
        + length_minutes \
        + children_in_home \
        + total_reviews',
```

```
    data = df_reviews
).fit()
```

Our first metric available for us to use is just the raw coefficient value, but they aren't comparable because the features are on very different scales. Moving one unit in movie length is not the same as moving a unit in age. We can standardize them which helps in this regard, and you might start there.

Another approach we can use comes from the SHAP value, which, as we saw, provides a measure of contribution of a feature to the prediction. These can be positive or negative and are specific to the observation. But if we take the **average absolute SHAP value** for each feature, we can maybe get a sense of the typical contribution size for the features. We can then rank order them accordingly. Here we see that the most important features are the number of reviews and the length of the movie. Note that we can't speak to direction here, only magnitude. We can also see that word count is relatively less important.

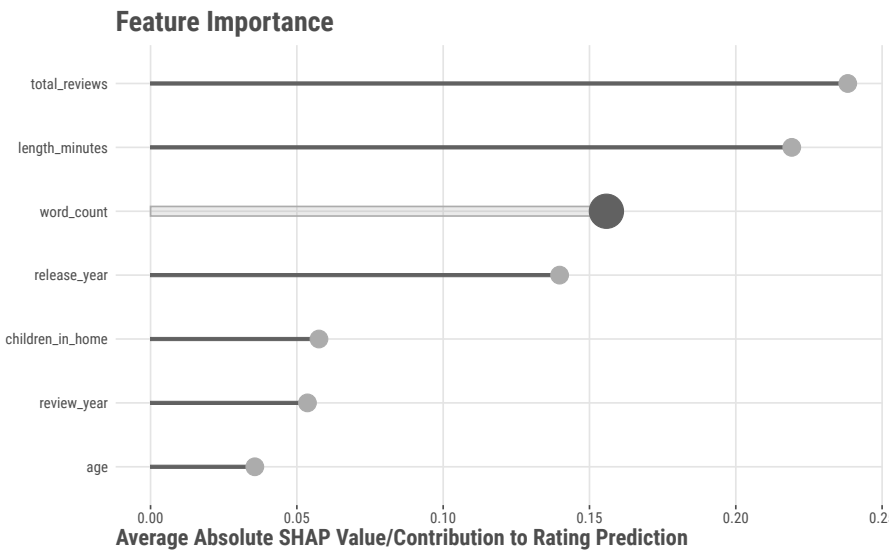


Figure 5.5: SHAP importance.

Now here are some additional methods[2]. Some of these decompose R^2 into the relative contributions to it from each feature (car, lmg, and pratt). The others

[2]The car, lmg, pratt, and beta-squared values were provided by the relaimpo package in R. See the documentation there for details. Permutation-based importance was provided by the iml package, though we supplied a custom function that returns the drop in R-squared value. SHAP values were calculated using the fastshap package. In Python you can use the shap output, and sklearn has a permutation-based importance function as well.

do not (SHAP, permutation-based, standardized coefficient squared). On the left, values represent the proportion of the R^2 value that is attributable to the feature– their sum is equal to the overall $R^2 = 0.32$. These are in agreement for the most part and seem to think more highly of word count as a feature, but they aren't actually the same value.

The others on the right suggest word count and age should rank lower, and length and review year higher. Which is best? Which is correct? Any of them. But by looking at a few of these, we can at least get a sense that total reviews, word count, release year, and length in minutes are probably useful features to our model, while age, review year, and children in the home are less so, *at least in the context of the model we have.*

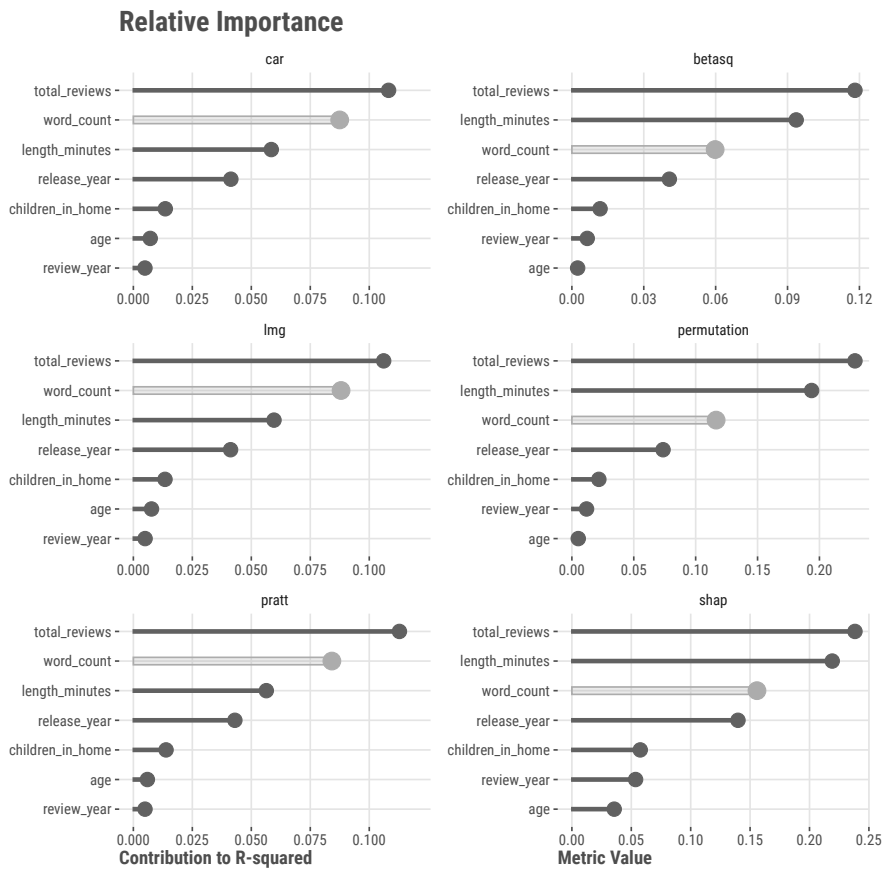


Figure 5.6: Feature importance by various methods.

All of the metrics shown have uncertainty in their estimate, and some packages make it easy to plot or extract. As an example one could bootstrap a metric, or use the permutations as a means to get at the uncertainty. However, the behavior and distribution of these metrics is not always well understood, and in some cases, the computation would often be notable (e.g., with SHAP). You could also look at the range of the ranks created by bootstrapping or permuting, and take the upper bound as the worst case for a given feature. Although this might possibly be conservative, the usual problem is that people are too optimistic about their feature importance result, so this might be a good thing.

The take-home message is that in the best of circumstances, there is no automatic way of saying one feature is more important than another. It's nice that we can use approaches like SHAP and permutation methods for more complicated models like boosting and deep learning models, but they're not perfect, and they still suffer from most of the same issues as the linear model. In the end, understanding a feature's role within a model is ultimately a matter of context, and highly dependent on what you're trying to do with the information.

> **i** SHAP as a Global Metric
>
> SHAP values can be useful for observational-level interpretation under the right circumstances, but they really shouldn't be used for importance. The mean of the absolute values is not a good measure of importance except in the very unlikely case you have purely balanced/symmetric features of the exact same scale, and which do not correlate with each other, or have any interactions. But if you had such a setting, you actually could just use the standardized coefficients.

5.10 Feature Metrics for Classification

All of what's been demonstrated for feature metrics applies to classification models, some of which were explicitly demonstrated. In general, counterfactual predictions, average marginal effects, SHAP, and permutation-based methods for feature importance would be done in the exact same way. The only real difference is the reference target – our results would be in terms of probabilities and using a different loss metric to determine importance, that sort of thing. But in general, everything we've talked about holds for both regression and binary classification settings as well.

5.11 Wrapping Up

It's generally a good idea to look at which features are pulling the most weight in your model to better understand how predictions are being made. But it takes a lot of work to trust what a model is telling you.

5.11.1 The common thread

As with Chapter 4, much of what you've seen here applies to many modeling scenarios. In many settings we are interested in feature-level interpretation. This can take a statistical focus, or use tools that are more model-agnostic like SHAP. In these situations, we are usually interested in *how* the model is making its predictions, and how we can use that information to make better decisions.

5.11.2 Choose your own adventure

If you haven't already, feel free to take your linear models further in Chapter 9 and Chapter 8, where you'll see how to handle different distributions for your target, add interactions, nonlinear effects, and more. Otherwise, you've got enough at this point to try your hand at the Chapter 10 section, where you can dive into machine learning!

5.11.3 Additional resources

To get the most out of DALEX visualizations, check out the authors' book Explanatory Model Analysis.

We also recommend checking out Christoph Molnar's book, Interpretable Machine Learning. It is a great resource for learning more about model explainers and how to use them, and it provides a nice package that has a lot of the functionality we've shown here.

The marginal effects zoo, written by the marginaleffects package author, is your go-to for getting started with marginal effects, but we also recommend the excellent blog post by Andrew Heiss as a very nifty overview and demonstration.

5.12 Guided Exploration

Let's put our model understanding that we've garnered in this and the previous chapter to the test. You'll use the world happiness dataset.

- Start by creating a model of your choosing that predicts happiness score. This can be the one you did for the previous exercise (Section 3.10).
- Now create a simpler or more complex model for comparison. The simple model should just remove a feature, while the complex model should add features to your initial model.
- Interpret the first model as a whole, then compare it to the simpler/more complex model. Did anything change in how you assessed the features they have in common?
- Create a counterfactual prediction for an observation, either one from the observed data or one that is of interest to you.
- Choose the 'best' model, and justify your reason for doing so, e.g., using a specific metric of your choosing.
- As a bonus, get feature importance metrics for your chosen model. If using R, we suggest using the iml package (example), and in Python, run the linear regression with scikit-learn and use the permutation-based importance function.

R

```
library(iml)

model = model_happiness
y = df_happiness_2018$happiness_score
X = model$model[,-1]  # if using lm -1 removes the happiness score
mod = Predictor$new(model, data = X, y = y)


# Compute feature importances as the performance drop in mean absolute error
imp = FeatureImp$new(mod, loss = "mse")

# Plot the results directly
plot(imp)
```

Python

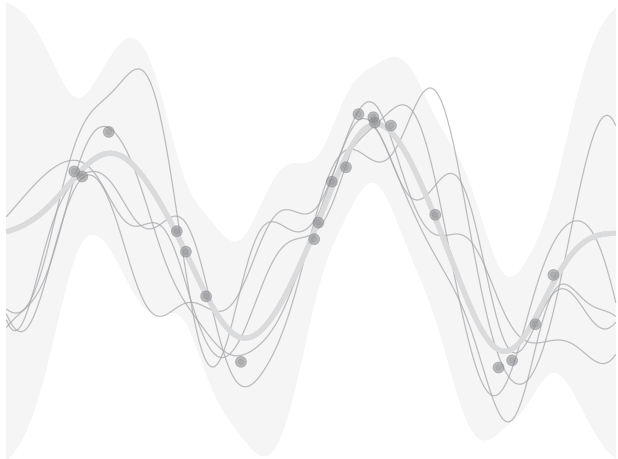
```
from sklearn.linear_model import LinearRegression
from sklearn.inspection import permutation_importance
import pandas as pd

y = df_happiness_2018['happiness_score']
X = df_happiness_2018[[MY_FEATURES]]
model = LinearRegression().fit(X, y)

importance = permutation_importance(
```

```
    model,
    X,
    y,
    n_repeats=30,
    random_state=0
)

(
    pd.DataFrame(
        importance.importances_mean,
        index=X.columns
    )
    .sort_values(by=0, ascending=False)
    .plot(kind='bar')
)
```

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

6

Model Estimation and Optimization



In our initial linear model, the coefficients for each feature are the key parameters. But how do we know what the coefficients are, and how did we come to those values? When we explore a linear model using some package function, they appear magically, but it's worth knowing a little bit about how they come to be, so let's try and dive a little deeper. As we do so, we'll end up going a long way into 'demystifying' the modeling process.

Model estimation is the process of finding the parameters associated with a model that allow us to reach a particular modeling goal. Different types of models will have different parameters to estimate, and there are different ways to estimate them. In general though, the goal is the same, find the set of parameters that will lead to the best predictions under the current data modeling context.

With model estimation, we can break things down into the following steps:

1. Start with an **initial guess** for the parameters.
2. Calculate the **prediction error** based on those parameters, or some function of it, or some other value that represents our model's **objective**.

3. **Update** the guess.
4. Repeat steps 2 and 3 until we find a 'best' guess.

Pretty straightforward, right? Well, it's not always so simple, but this is the general idea in most applications, so keep it in mind! In this chapter, we'll show how to do this ourselves to take away the mystery a bit from when you run standard model functions in typical contexts. Hopefully, then you'll gain more confidence when you do use them!

6.1 Key Ideas

A few concepts we'll keep using here are fundamental to understanding estimation and optimization. We should note that we're qualifying our present discussion of these topics to typical linear models, machine learning, and similar settings, but they are much more broad and general than presented here. We've seen some of this before, but we'll be getting a bit cozier with the concepts now.

- **Parameters** are the values associated with a model that we have to estimate.
- **Estimation** is the process of finding the parameters associated with a model.
- The **objective (loss) function** takes input and produces a value that we want to maximize or minimize.
- **Prediction error** is the difference between the observed value of the target and the predicted value, and is often used to calculate the objective function.
- **Optimization** is the specific process of finding the parameters that maximize or minimize some objective function.
- **Model selection** is the process of choosing the best model from a set of models.

> **i** Estimation vs. Optimization
>
> We can use **estimation** as general term for finding parameters, while **optimization** can be seen as a term for finding parameters that maximize or minimize some **objective function**, or even a combination of objectives. In some cases, we can estimate parameters without optimization, because there is a known way of solving the problem, but in most modeling situations we are going to use some optimization approach to find a 'best' set of parameters.

6.1.1 Why this matters

When it comes to modeling, even knowing just a little bit about what goes on behind the scenes is a great demystifier. And if models are less of a mystery, you'll feel more confident using them. Much of what you see here is part of almost every common model used for statistics and machine learning, and adding this knowledge provides you a more solid foundation for expanding your modeling skills.

6.1.2 Helpful context

This chapter is more involved and technical than most of the others, so it might be more suited for those who like to get their hands dirty. It's all about DIY, and so we'll be doing a lot of the work ourselves. If you're not one of those types of people who gets much out of that, that's okay, you can skip this chapter and still get a lot out of the rest of the book. But if you're curious about how models work, or you want to be able to do more than just run a canned function, then we think you'll find the following useful. You'd want to at least have your linear model basics down (Chapter 3).

6.2 Data Setup

For the examples here, we'll use the world happiness dataset for the year 2018. We'll use the happiness score as our target. Let's take an initial look at the data here, but for more information see the Appendix Section C.2.

Figure 6.1: World happiness data summary.

Our happiness score has values from around 3-7, life expectancy and GDP appear to have some notable variability, and corruption perception is skewed toward lower values. We can also see that the features and target are correlated, which is not surprising.

Table 6.1: Correlation Matrix for World Happiness Data

term	happiness	life_exp	log_gdp_pc	corrupt
happiness	1.00	0.78	0.82	−0.47
life_exp	0.78	1.00	0.86	−0.34
log_gdp_pc	0.82	0.86	1.00	−0.34
corrupt	−0.47	−0.34	−0.34	1.00

For our purposes here, we'll drop any rows with missing values, and we'll use scaled features so that they have the same variance, which, as noted in the data chapter (Chapter 14), can help make estimation easier.

R

```
df_happiness = read_csv('https://tinyurl.com/worldhappiness2018') |>
    drop_na() |>
    rename(happiness = happiness_score) |>
    select(
        country,
        happiness,
        contains('_sc')
    )
```

Python

```
import pandas as pd

df_happiness = (
    pd.read_csv('https://tinyurl.com/worldhappiness2018')
    .dropna()
    .rename(columns = {'happiness_score': 'happiness'})
    .filter(regex = '_sc|country|happ')
)
```

6.2.1 Other Setup

For the R examples, after the above, nothing beyond base R is needed. For Python examples, the following should be enough to get you through the examples.

```
import pandas as pd
import numpy as np

import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

from scipy.optimize import minimize
from scipy.stats import norm
```

6.3 Starting out by Guessing

So, we'll start with a model in which we predict a country's level of happiness by their life expectancy. If you can expect to live longer, you're probably in a country with better health care, higher incomes, and other important factors. As such, we can probably expect higher happiness scores with higher life expectancy. We'll stick with a linear regression model as well to start out.

As a starting point, we can just guess what the parameter should be. But how would we know what to guess? How would we know which guesses are better than others? Let's try a couple guesses and see what happens. Let's say that we think all countries start at the bottom on the happiness scale (around 3), but life expectancy makes a big impact– for every standard deviation of life expectancy, we go up a whole point on happiness[1]. We can plug this into the model and see what we get:

$$\text{prediction} = 3.3 + 1 \cdot \text{life\_exp}$$

For a different model, we'll say countries start with a mean of happiness score, and moving up a standard deviation of life expectancy would move us up a half point of happiness.

$$\text{prediction} = \overline{\text{happiness}} + .5 \cdot \text{life\_exp}$$

How do we know which is better? Let's find out!

[1]Since life expectancy is scaled, a standard deviation is 1, and in this case represents about 7 years.

6.4 Prediction Error

We've seen that a key component to model assessment involves comparing the predictions from the model to the actual values of the target. This difference is known as the **prediction error**, or **residuals** in more statistical contexts. We can express this as:

$$\epsilon = y - \hat{y}$$

$$\text{error} = \text{target} - \text{model-based guess}$$

This prediction error tells us how far off our model prediction is from the observed target values, but it also gives us a way to compare models. How? With our measure of prediction error, we can calculate a total error for all observations/predictions (Section 4.2), or similarly, the average error. If one model or parameter set has less total or average error, we can say it's a better model than one that has more (Section 4.3). Ideally we'd like to choose a model with the least possible error, but we'll see that this is not always possible[2].

However, if we just take the average of our errors from a linear regression model, you'll see that it is roughly zero! This is by design for many common models and is even made explicit in their mathematical depiction. So, to get a meaningful error metric, we need to use the squared error value or the absolute value. These also allow errors of similar value above and below the observed value to cost the same[3]. As we've done elsewhere, we'll use squared error here, and we'll calculate the mean of the squared errors for all our predictions, i.e., the **Mean Squared Error**(MSE).

R

```
y = df_happiness$happiness

# Calculate the error for the guess of 4
prediction = min(df_happiness$happiness) + 1*df_happiness$life_exp_sc
mse_model_A = mean((y - prediction)^2)

# Calculate the error for our other guess
```

[2]It turns out that our error metric is itself an *estimate* of the true error. We'll get more into this later, but for now this means that we can't ever know the true error, and so we can't ever really know the best or true model. However, we can still choose a good or better model relative to others based on our error estimate.

[3]We don't have to do it this way, but it's the default in most scenarios. As an example, maybe for your situation, overshooting is worse than undershooting, and so you might want to use an approach that would weight those errors more heavily.

```
prediction = mean(y) + .5 * df_happiness$life_exp_sc
mse_model_B = mean((y - prediction)^2)

tibble(
    Model = c('A', 'B'),
    MSE = c(mse_model_A, mse_model_B)
)
```

Python

```
y = df_happiness['happiness']

# Calculate the error for the guess of four
prediction = np.min(df_happiness['happiness']) + \
    1 * df_happiness['life_exp_sc']

mse_model_A = np.mean((y - prediction)**2)

# Calculate the error for our other guess
prediction = y.mean() + .5 * df_happiness['life_exp_sc']
mse_model_B = np.mean((y - prediction)**2)

pd.DataFrame({
    'Model': ['A', 'B'],
    'MSE': [mse_model_A, mse_model_B]
})
```

Now let's look at our MSE, and we'll also inspect the square root of it, or the **Root Mean Squared Error** (), as that puts things back on the original target scale and tells us the standard deviation of our prediction errors. We also add the **Mean Absolute Error** (MAE) as another metric with straightforward interpretation.

Table 6.2: Comparison of Error Metrics for Two Models

Model	MSE	RMSE	MAE	RMSE % drop	MAE % drop
A	5.09	2.26	1.52		
B	0.64	0.80	0.58	65%	62%

Inspecting the metrics, we can see that we are off on average by over a point for model A (MAE), and a little over half a point on average for model B. So we can see that model B is not only better, but it results in a 65% drop in RMSE, and similar for MAE. We'd definitely prefer it over model A. This approach is also how we can compare models in a general fashion.

Now all of this is useful, and at least we can say one model is better than another. But you're probably hoping there is an easier way to get a good guess for our model parameters, especially when we have possibly dozens of features and/or parameters to keep track of, and there is!

6.5 Ordinary Least Squares

In a simple linear model, we often use the **Ordinary Least Squares** (OLS) method to estimate parameters. This method finds the coefficients that minimize the sum of the squared differences between the predicted and actual values[4]. In other words, it finds the coefficients that minimize the sum of the squared differences between the predicted values and the actual values, which is what we just did in our previous example. The sum of the squared errors is also called the **Residual Sum of Squares** (RSS), as opposed to the 'total' sums of squares (i.e., the variance of the target), and the part explained by the model ('model' or 'explained' sums of squares). We can express this as follows, where y_i is the observed value of the target for observation i, and $\hat{y}_i$ is the predicted value from the model.

$$\text{Value} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{6.1}$$

It's called *ordinary* least squares because there are other least squares methods, generalized least squares, weighted least squares, and others, but we don't need to worry about that for now. The sum or mean of the squared errors is our **objective value**. The **objective function** takes the predictions and observed target values as inputs and returns the objective value as an output. We can use this value to find the best parameters for a specific model, as well as compare models with different parameters.

Now let's calculate the OLS estimate for our model. We need our own function to do this, but it doesn't take much to create one. We need to map our inputs to our output, which are the model predictions. We then calculate the error, square it, and then average the squared errors to provide the mean squared error.

[4]Some disciplines seem to confuse models with estimation methods and link functions. It doesn't really make sense, nor is it informative, to call something an OLS model or a logit model. Many models are estimated using a least squares objective function, even deep learning, and different types of models use a logit link, from logistic regression, to beta regression, to activation functions used in deep learning.

R

```
# for later comparison
model_lr_happy = lm(happiness ~ life_exp_sc, data = df_happiness)

ols = function(X, y, par) {
    # add a column of 1s for the intercept
    X = cbind(1, X)

    # Calculate the predicted values
    y_hat = X %*% par  # %*% is matrix multiplication

    # Calculate the error
    error = y - y_hat

    # Calculate the value as mean squared error
    value = sum(error^2) / nrow(X)

    # Return the objective value
    return(value)
}
```

Python

```
# for later comparison
model_lr_happy = smf.ols('happiness ~ life_exp_sc', data = df_happiness).fit()

def ols(par, X, y):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate the predicted values
    y_hat = X @ par  # @ is matrix multiplication

    # Calculate the mean of the squared errors
    value = np.mean((y - y_hat)**2)

    # Return the objective value
    return value
```

We'll want to make a bunch of guesses for the parameters, so let's create data for those guesses. We'll then choose the guess that gives us the lowest objective value. But before getting carried away, try it out for just one guess to see how it works!

R

```
# create a grid of guesses
guesses = crossing(
    b0 = seq(1, 7, 0.1),
    b1 = seq(-1, 1, 0.1)
)

# Example for one guess
ols(
    X = df_happiness$life_exp_sc,
    y = df_happiness$happiness,
    par = unlist(guesses[1, ])
)
```

```
[1] 23.78
```

Python

```
# create a grid of guesses
from itertools import product

guesses = pd.DataFrame(
    product(
        np.arange(1, 7, 0.1),
        np.arange(-1, 1, 0.1)
    ),
    columns = ['b0', 'b1']
)

# Example for one guess
ols(
    par = guesses.iloc[0,:],
    X = df_happiness['life_exp_sc'],
    y = df_happiness['happiness']
)
```

```
23.77700449624871
```

Now we'll calculate the loss for each guess and find which one gives us the smallest function value.

R

```
# Calculate the function value for each guess, mapping over
# each combination of b0 and b1
guesses = guesses |>
    mutate(
        objective = map2_dbl(
            guesses$b0,
            guesses$b1,
            \(b0, b1) ols(
                par = c(b0, b1),
                X = df_happiness$life_exp_sc,
                y = df_happiness$happiness
            )
        )
    )

min_loss = guesses |> filter(objective == min(objective))

min_loss
```

```
# A tibble: 1 x 3
     b0    b1 objective
  <dbl> <dbl>     <dbl>
1   5.4   0.9     0.491
```

Python

```
# Calculate the function value for each guess, mapping over
# each combination of b0 and b1
guesses['objective'] = guesses.apply(
    lambda x: ols(
        par = x,
        X = df_happiness['life_exp_sc'],
        y = df_happiness['happiness']
    ),
    axis = 1
)

min_loss = guesses[guesses['objective'] == guesses['objective'].min()]

min_loss
```

```
       b0    b1  objective
899 5.400 0.900      0.491
```

The following plot shows the objective value for each guess in a smooth fashion as if we had a more continuous space. Darker values indicate we're getting closer to our smallest objective value. The star notes our best guess.

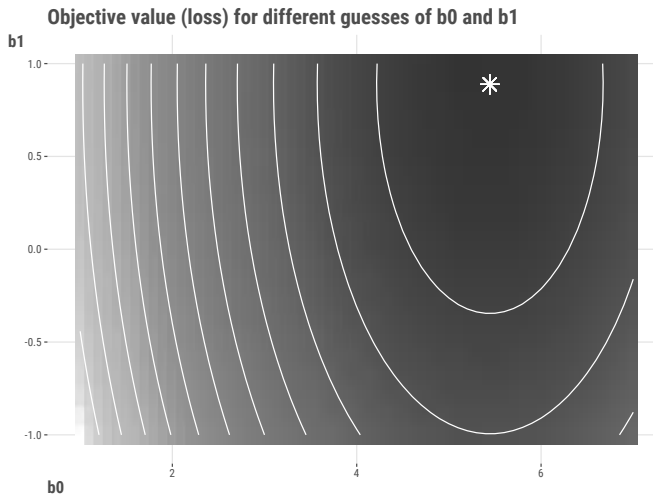


Figure 6.2: Results of parameter search.

Now let's run the model using standard functions.

R

```
model_lr_happy_life = lm(happiness ~ life_exp_sc, data = df_happiness)

# not shown
c(
    coef(model_lr_happy_life),
    performance::performance_mse(model_lr_happy_life)
)
```

Python

```
model_lr_happy_life = sm.OLS(
    df_happiness['happiness'],
    sm.add_constant(df_happiness['life_exp_sc'])
).fit()

# not shown
model_lr_happy_life.params, model_lr_happy_life.scale
```

If we inspect our results from the standard functions, we had estimates of 5.44 and 0.89 for our coefficients versus the best guess from our approach of 5.4 and 0.9. These are very similar but not exactly the same, but this is mostly due to the granularity of our guesses. Even so, in the end we can see that we get pretty dang close to what our basic `lm` or `statsmodels` functions would get us. Pretty neat!

> **i** Estimation as 'Learning'
>
> Estimation and/or optimization can be seen as the process of a model *learning* which parameters will best allow the predictions to match the observed data, and hopefully, predict as-yet-unseen future data. This is a very common way to think about estimation in *machine learning*, and it is a useful way to think about our simple linear model also.
> One thing to keep in mind is that it is not a magical process. It takes good data, a good idea (model), and an appropriate estimation method to get good results.

6.6 Optimization

Before we get into other objective functions, let's think about a better way to find good parameters for our model. Rather than just guessing, we can use a more systematic approach, and thankfully, there are tools out there to help us. We just use a function like our OLS function, give it a starting point, and let the algorithms do the rest! These tools eventually arrive at a pretty good set of parameters and are optimized for speed.

Previously we created a set of guesses and tried each one. This approach is often called a **grid search**, as we search over a grid of possible parameters as in Figure 6.2, and it is a bit of a brute-force approach to finding the best fitting model. You can maybe imagine a couple of unfortunate scenarios for this approach. One is just having a very large number of parameters to search, a common issue in deep learning. Or it may be that our range of guesses doesn't allow us to find the right set of parameters. Or perhaps we specify a very large range, but the best fitting model is within a very narrow part of that, so that it takes a long time to find them. In any of these cases, we waste a lot of time or may not find an optimal solution.

In general, we can think of **optimization** as employing a smarter, more efficient way to find what you're looking for. Here's how it works:

- **Start with an initial guess** for the parameters.
- **Calculate the objective function** given the parameters.

- **Update the parameters** to a new guess (that hopefully improves the objective function).
- **Repeat**, until the improvement is small enough to not be worth continuing, or an arbitrary maximum number of iterations is reached.

With optimization, the key aspect is how we *update* the old parameters with a new guess at each iteration. Different **optimization algorithms** use different approaches to find those new guesses. The process stops when the improvement is smaller than a set **tolerance** level, or the maximum number of iterations is reached. If the objective is met, we say that our model has **converged**. Sometimes, the number of iterations is not enough for us to reach convergence in terms of tolerance, and we have to try again with a different set of parameters, a different algorithm, maybe use some data transformations, or something else.

So, let's try it out! We start out with several inputs:

- the objective function,
- the initial guess for the parameters to get things going,
- other related inputs to the objective function, such as the data, and
- options for the optimization process, e.g., algorithm, maximum number of iterations, etc.

With these inputs, we'll let the chosen optimization function do the rest of the work. We'll again compare our results to the standard functions to make sure we're on the right track.

R

We'll use the `optim` function in R.

```
our_ols_optim = optim(
    par = c(1, 0),    # initial guess for the parameters
    fn  = ols,
    X   = df_happiness$life_exp_sc,
    y   = df_happiness$happiness,
    method  = 'BFGS', # optimization algorithm
    control = list(
        reltol   = 1e-6, # tolerance
        maxit = 500   # max iterations
    )
)

our_ols_optim
```

Python

We'll use the `minimize` function in Python.

```
from scipy.optimize import minimize

our_ols_optim = minimize(
    fun  = ols,
    x0   = np.array([1., 0.]),  # initial guess for the parameters
    args = (
        np.array(df_happiness['life_exp_sc']),
        np.array(df_happiness['happiness'])
    ),
    method  = 'BFGS',           # optimization algorithm
    tol     = 1e-6,             # tolerance
    options = {
        'maxiter': 500          # max iterations
    }
)

our_ols_optim
```

Optimization functions typically return multiple values, including the best parameters found, the value of the objective function at that point, and sometimes other information like the number of iterations it took to reach the returned value and whether or not the process converged. This can be quite a bit of stuff, so we don't show the raw output, but we definitely encourage you to inspect it closely. The following table shows the estimated parameters and the objective value for our model, and we can compare it to the standard functions to see how we did.

Table 6.3: Comparison of OLS Results to a Standard Function

Parameter	Standard	Our Result
Intercept	5.4450	5.4450
Life Exp. Coef.	0.8880	0.8880
Objective/MSE	0.4890	0.4890

So, our little function and the right tool allow us to come up with the same thing as base R and `statsmodels`! I hope you're feeling pretty good at this point because you should! You just proved you could do what seemed before to be like magic, but really all it took is just a little knowledge about some key concepts to demystify the process. So, let's keep going!

> **i** A Note on Terminology
>
> The objective function is often called the **loss function**, and sometimes the **cost function**. However, these both imply that we are trying to minimize the function, which is not always the case[5], and it's arbitrary whether you want to minimize or maximize the function.
> The term **metric**, such as the MSE or AUROC we've seen elsewhere, is a value that you might also want to use to evaluate the model. Some metrics are also used as an objective function. For instance, we might minimize MSE as our objective, but also calculate other metrics like Adjusted R-squared or Mean Absolute Error to evaluate the model. It's fine to use MSE as the sole objective/metric as well.
> This can be very confusing when starting out! We'll try to stick to the term *metric* for additional values that we might want to examine, apart from the *objective function value*, which is specifically used for estimating model parameters.

6.7 Maximum Likelihood

In our example, we've been minimizing the mean of the squared errors to find the best parameters for our model. But let's think about this differently. Now we'd like you to think about the **data generating process**. Ignoring the model, imagine that each happiness value is generated by some random process, like drawing from a normal distribution. So, something like this would describe it mathematically:

$$\text{happiness} \sim N(\text{mean}, \text{sd})$$

where the `mean` is just the mean of happiness, and `sd` is its standard deviation. In other words, we can think of happiness as a random variable that is drawn from a normal distribution with mean and standard deviation as the parameters of that distribution.

[5]You may find that some packages will *only* minimize (or maximize) a function, even to the point of reporting nonsensical things like negative squared values, so you'll need to take care when implementing your own metrics.

Let's apply this idea to our linear model setting. In this case, the mean is a function of life expectancy, and we're not sure what the standard deviation is, but we can go ahead and write our model as follows.

$$\text{mean} = \beta_0 + \beta_1 * \text{life\_exp}$$

$$\text{happiness} \sim N(\text{mean}, \text{sd})$$

Now, our model is estimating the parameters of the normal distribution. We have an extra parameter to estimate: the standard deviation, which is similar to our RMSE.

In our analysis, instead of merely comparing the predicted happiness score to the actual score by looking at their difference, we do something a little different to get a sense of their correspondence. We consider how *likely* it is to observe the actual happiness score based on our prediction. The value known as the **likelihood** depends on our model's parameters. The likelihood is determined by the statistical distribution we've selected for our analysis. We can write this as:

$$\Pr(\text{happiness} \mid \text{life\_exp}, \beta_0, \beta_1, \text{sd})$$

$$\Pr(\text{happiness} \mid \text{mean}, \text{sd})$$

Thinking more generally, the likelihood gives us the probability of the observed data given the parameter estimates.

$$\Pr(\text{Data} \mid \text{Parameters})$$

The following shows how to calculate a likelihood for our model. The values you see are called **probability density** values. They're not exactly probabilities, but they show the **relative likelihood** of each observation[6]. You can think of them like you do for probabilities, but remember that likelihoods are slightly different.

R

```
# two example life expectancy scores, at the mean (0) and 1 sd above
life_expectancy = c(0, 1)

# observed happiness scores
happiness = c(4, 5.2)
```

[6]The actual probability of a *specific value* in this setting is 0, but the probability of a range of values is greater than 0. You can find out more about likelihoods and probabilities at the StackExchange discussion on the difference between likelihood and probability, but many traditional statistical texts will cover this also.

```
# predicted happiness with rounded coefs
mu = 5 + 1 * life_expectancy

# just a guess for sigma
sigma = .5

# likelihood for each observation
L = dnorm(happiness, mean = mu, sd = sigma)
L
```

```
[1] 0.1080 0.2218
```

Python

```
from scipy.stats import norm

# two example life expectancy scores, at the mean (0) and 1 sd above
life_expectancy = np.array([0, 1])

# observed happiness scores
happiness = np.array([4, 5.2])

# predicted happiness with rounded coefs
mu = 5 + 1 * life_expectancy

# just a guess for sigma
sigma = .5

# likelihood for each observation
L = norm.pdf(happiness, loc = mu, scale = sigma)
L
```

```
array([0.1080, 0.2218])
```

With a guess for the parameters and an assumption about the data's distribution, we can calculate the likelihood of each data point, which is what our final result `L` shows. We eventually get a total likelihood for all observations, similar to how we added squared errors previously. But unlike errors, we want *more* likelihood, not less. In theory, we'd multiply each likelihood, but in practice we sum the *log of the likelihood*, otherwise values would get too small for our computers to handle. We can also turn our problem into a minimization problem by calculating the negative log-likelihood, and then minimizing that value, which many optimization algorithms are designed to do[7].

[7] The negative log-likelihood is often what is reported in the model output as well.

The following is a function we can use to calculate the likelihood of the data given our parameters. This value, just like MSE, can also be used to compare models with different parameter guesses[8]. For the estimate of sigma, note that we are estimating its log value by exponentiating the parameter guess. This will keep it positive, as the standard deviation must be positive. We'll hold off with our result until Table 6.4, which is shown next.

R

```
max_likelihood = function(par, X, y) {

    # setup
    X = cbind(1, X)     # add a column of 1s for the intercept
    beta = par[-1]      # coefficients
    sigma = exp(par[1]) # error sd, exp keeps positive
    N = nrow(X)

    LP = X %*% beta     # linear predictor
    mu = LP             # identity link in the glm sense

    # calculate (log) likelihood
    ll = dnorm(y, mean = mu, sd = sigma, log = TRUE)

    value = -sum(ll)    # negative for minimization

    return(value)
}

our_max_like = optim(
    par = c(1, 0, 0),   # first param is sigma
    fn  = max_likelihood,
    X   = df_happiness$life_exp_sc,
    y   = df_happiness$happiness
)

our_max_like
# logLik(model_lr_happy_life) # one way to extract the LL from a model
```

[8]Those who have experience here will notice we aren't putting a lower bound on sigma. You typically want to do this, otherwise you may get nonsensical results by not keeping sigma positive. You can do this by setting a specific argument for an algorithm that uses boundaries, or more simply by exponentiating the parameter so that it can only be positive. In the latter case, you'll have to exponentiate the final parameter estimate to get back to the correct scale. We leave this detail out of the code for now to keep things simple.

Python

```
def max_likelihood(par, X, y):

    # setup
    X = np.c_[np.ones(X.shape[0]), X]  # add a column of 1s for the intercept
    beta = par[1:]           # coefficients
    sigma = np.exp(par[0])   # error sd, exp keeps positive
    N = X.shape[0]

    LP = X @ beta            # linear predictor
    mu = LP                  # identity link in the glm sense

    # calculate (log) likelihood
    ll = norm.logpdf(y, loc = mu, scale = sigma)

    value = -np.sum(ll)      # negative for minimization

    return value

our_max_like = minimize(
    fun  = max_likelihood,
    x0   = np.array([1, 0, 0]), # first param is sigma
    args = (
        np.array(df_happiness['life_exp_sc']),
        np.array(df_happiness['happiness'])
    )
)

our_max_like
```

```
# model_lr_happy_life.llf # one way to extract the log likelihood from a model
```

We can compare our result to a built-in function that has capabilities beyond OLS, and the table shows we're duplicating the basic result. We show more decimal places on the log-likelihood estimate to prove we aren't getting *exactly* the same result.

Table 6.4: Comparison of Max. Likelihood Results to a Standard Function

Parameter	Standard	Our Result
Intercept	5.445	5.445
Life Exp. Coef.	0.888	0.888
Sigma[1]	0.705	0.699
LogLik (neg)	118.804	118.804

[1]Parameter estimate is exponentiated for the by-hand approach.

To use a maximum likelihood approach for linear models, you can use functions like `glm` in R or `GLM` in Python, which is the reference used in the table above. We can also use different likelihoods corresponding to the binomial, Poisson and other distributions. Still other packages would allow even more distributions for consideration. In general, we choose a distribution that we feel best reflects the data generating process. For binary targets for example, we typically would feel a Bernoulli or binomial distribution is appropriate. For count data, we might choose a Poisson or negative binomial distribution. For targets that fall between 0 and 1, we might go for a beta distribution. You can see some of these demonstrated in Chapter 8.

There are many distributions to choose from, and the best one depends on your data. Sometimes, even if one distribution seems like a better fit, we might choose another one because it's easier to use. Some distributions are special cases of others, or they might become more like a normal distribution under certain conditions. For example, the exponential distribution is a special case of the gamma distribution, and a t-distribution with many degrees of freedom looks like a normal distribution. Here is a visualization of the relationships among some of the more common distributions (Wikipedia (2023)).

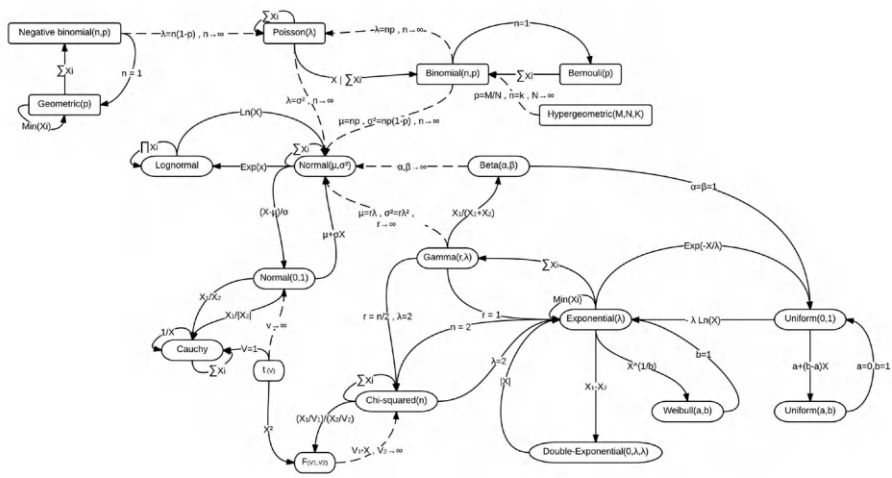


Figure 6.3: Relationships among some probability distributions.

When you realize that many distributions are closely related, it's easier to understand why we might choose one over another. But also, you can see why we might use a simpler option even if it's not the best fit – you likely won't come to a different practical conclusion about your model. Ultimately, you'll get a better feel for this as you work with different types of data and models.

Here are examples of standard GLM functions, which just require an extra argument for the family of the distribution.

R

```
glm(happiness ~ life_exp_sc, data = df_happiness, family = gaussian)
glm(binary_target ~ x1 + x2, data = some_data, family = binomial)
glm(count ~ x1 + x2, data = some_data, family = poisson)
```

Python

```
import statsmodels.formula.api as smf

smf.glm(
    'happiness ~ life_exp_sc',
    data = df_happiness,
    family = sm.families.Gaussian()
)

smf.glm(
    'binary_target ~ x1 + x2',
    data = some_data,
    family = sm.families.Binomial()
)

smf.glm(
    'count ~ x1 + x2',
    data = some_data,
    family = sm.families.Poisson()
)
```

6.7.1 Diving deeper

Let's think more about what's going on here, as it applies to estimation and optimization in general. It turns out that our objective function defines a 'space' or 'surface'. You can imagine the process as searching for the lowest point on a landscape, with each guess a point on this landscape. Let's start to get a sense of this with the following visualization, based on a single parameter. The following visualization shows this for a single parameter. The data comes from a variable with a true average of 5. As our guesses get closer to 5, the likelihood increases. However, with more and more data, the final guess converges on the true value. Model estimation finds that maximum on the curve, and optimization algorithms are the means to find it.

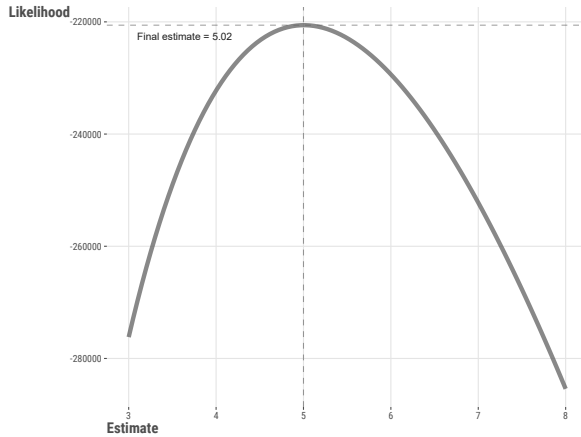


Figure 6.4: Likelihood function for one parameter.

Now let's add a parameter. If we have more than one parameter, we now have an objective surface to deal with. Given a starting point, an optimization procedure then travels along the surface looking for a minimum/maximum point. For simpler settings such as this, we can visualize the likelihood surface and its minimum point. However, even our simple model has three parameters plus the likelihood, so it would be difficult to visualize without additional complexity. Instead, we show the results for an alternate model where happiness is standardized also, which means the intercept is zero[9], and so it is not shown.

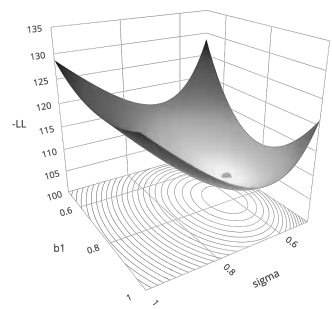


Figure 6.5: Likelihood surface for two parameters.

[9]Linear regression will settle on a line that cuts through the means, and when standardizing all variables, the mean of the features and target are both zero, so the line goes through the origin.

We can also see the path our estimates take. Starting at a fairly bad estimate, the optimization algorithm quickly updates to estimates that result in a better likelihood value. We also see little exploratory jumps creating a star-like pattern, before things ultimately settle to the best values. In general, these updates and paths are dependent on the optimization algorithm one uses.

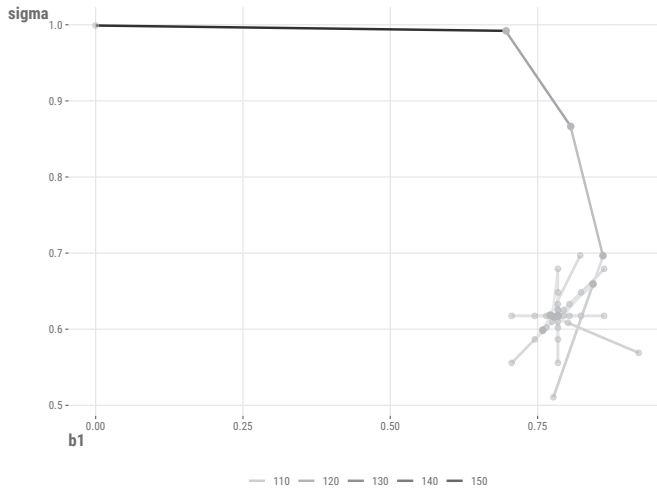


Figure 6.6: Optimization path for two parameters.

What we've shown here with maximum likelihood applies in general to searching for the best solution along an objective function surface. In most modeling circumstances, the surface is very complex with lots of points that might represent 'local' minimums, even though we'd ideally find the 'global' minimum. This is why we need optimization algorithms to help us find the best solution[10]. The optimization algorithms we'll use are general purpose, and can be used for many different types of problems. The key is to define an appropriate objective function, and then let the algorithm do the work.

> **i** MLE and OLS
>
> For linear regression assuming a normal distribution, the maximum likelihood estimate of the standard deviation is the OLS estimate of the standard deviation of the residuals. Furthermore, the maximum likelihood coefficient estimates and OLS estimates converge to the same

[10]There is no way to know if we have found a global minimum, and truthfully, we probably rarely do with complex models. But optimization algorithms are designed to find the best solution they can under the data circumstances, and some minimums may be practically indistinguishable from the global minimum anyway (we hope!).

estimates as the sample size increases. In practice these estimates are indistinguishable, and the OLS estimate is the maximum likelihood estimate for linear regression. So OLS and variants (such as those used for GLM) are maximum likelihood estimation methods.

6.8 Penalized Objectives

One thing we may want to take into account with our models is their complexity, especially in the context of **overfitting**. We talk about this with machine learning also (Chapter 10), but the basic idea is that we can get too familiar with the data we have, and when we try to predict on new data the model hasn't seen before, model performance suffers. In other words, we are not generalizing well (Section 10.4).

One way to deal with this is to penalize the objective function value for complexity, or at least favor simpler models that might do as well. In some contexts this is called **regularization**, and in other contexts **shrinkage**, since the parameter estimates are typically shrunk toward some specific value (e.g., zero).

As a starting point, in our basic linear model we can add a penalty that is applied to the size of coefficients. This is called **ridge regression**, or, more mathily, **L2 regularization**. We can write this formally as:

$$\text{Value} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \tag{6.2}$$

The first part is the same as basic OLS (Equation 6.1), but the second part is the penalty for p features. The penalty is the sum of the squared coefficients multiplied by some value, which we call λ. This is an additional model parameter that we typically want to estimate, for example, through cross-validation. This kind of parameter is often called a **hyperparameter**, mostly just to distinguish it from those that may be of actual interest. For example, we could probably care less what the actual value for λ is, but we would still be interested in the coefficients.

In the end this is just a small change to OLS regression (Equation 6.1), but it can make a big difference. It introduces some bias in the coefficients – recall that OLS is unbiased if assumptions are met – but it can help to reduce variance, which can help the model perform better on new data (Section 10.4.2). In other words, we are willing to accept some bias in order to get a model that generalizes better.

But let's get to an example to demystify this a bit! Here is an example of a function that calculates the ridge objective. To make things interesting, let's add the other features we talked about regarding GDP per capita and perceptions of corruption.

R

```
ridge = function(par, X, y, lambda = 0) {
    # add a column of 1s for the intercept
    X = cbind(1, X)

    mu = X %*% par # linear predictor

    # Calculate the value as sum squared error
    error = sum((y - mu)^2)

    # Add the penalty
    value = error + lambda * sum(par^2)

    return(value)
}

X = df_happiness |>
    select(-happiness, -country) |>
    as.matrix()

our_ridge = optim(
    par = c(0, 0, 0, 0),
    fn = ridge,
    X = X,
    y = df_happiness$happiness,
    lambda = 0.1,
    method = 'BFGS'
)

our_ridge$par
```

Python

```
# we use lambda_ because lambda is a reserved word in Python
def ridge(par, X, y, lambda_ = 0):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]
```

```
    # Calculate the predicted values
    mu = X @ par

    # Calculate the error
    value = np.sum((y - mu)**2)

    # Add the penalty
    value = value + lambda_ * np.sum(par**2)

    return value

our_ridge = minimize(
    fun  = ridge,
    x0   = np.array([0, 0, 0, 0]),
    args = (
        np.array(df_happiness.drop(columns=['happiness', 'country'])),
        np.array(df_happiness['happiness']),
        0.1
    )
)

our_ridge['x']
```

We can compare this to built-in functions as we have before[11], and we can see that the results are very similar, but not exactly the same. We would not worry about such differences in practice, but the main point is again, that we can use simple functions that do just about as well as those from common packages.

Table 6.5: Comparison of Ridge Regression Results

Parameter	Standard[1]	Our Result
Intercept	5.44	5.44
life_exp_sc	0.49	0.52
corrupt_sc	−0.12	−0.11
gdp_pc_sc	0.42	0.44

[1]Showing results from the R glmnet package with alpha = 0, lambda = .1.

[11]For R, one can use the glmnet, while for Python, the Ridge class in scikit-learn is a good choice.

> **i** Analytical Solution for Ridge Regression
>
> It turns out that given a fixed λ penalty ridge regression estimates can be derived analytically. Clark (2021a) (one of your authors) has an example in his Model Estimation by Example demos.

Another very common penalized approach is to use the sum of the absolute value of the coefficients, which is called **lasso regression** or **L1 regularization**. An interesting property of the lasso is that in typical implementations, it will potentially produce a value of zero for some coefficients, which is the same as dropping the associated feature from the model altogether. This is a form of **feature selection** or **variable selection**. The true values are never zero, but if we want to use a 'best subset' of features, this is one way we could do so. We can write the lasso objective as follows. The chapter exercise asks you to implement this yourself.

$$\text{Value} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{6.3}$$

6.9 Classification

So far, we've been assuming a continuous target, but what if we have a categorical target? Now we have to learn a bunch of new stuff for that situation, right? Actually, no! *When we want to model categorical targets, conceptually, nothing changes with our estimation approach*! We still have an objective function that maximizes or minimizes some goal, and we can use the same algorithms to estimate parameters. However, we need to think about how we can do this in a way that makes sense for the binary target, or generalize to the multiclass case.

6.9.1 Misclassification rate

A straightforward correspondence to common loss functions we've seen is a function that minimizes classification error, or by the same token, maximizes accuracy. In other words, we can think of the objective function as the proportion of incorrect classifications. This is called the **misclassification rate**.

$$\text{Value} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}(y_i \neq \hat{y}_i) \tag{6.4}$$

In the equation, y_i is the actual value of the target for observation i, arbitrarily coded as 1 or 0, and $\widehat{y}_i$ is the predicted class from the model. The $\mathbb{1}$ is an indicator function that returns 1 if the condition is true, and 0 otherwise. In other words, we are counting the number of times the predicted value is not equal to the actual value, and dividing by the number of observations. Very straightforward, so let's do this ourselves!

R

```
misclassification = function(par, X, y, class_threshold = .5) {
    X = cbind(1, X)

    # Calculate the 'linear predictor'
    mu = X %*% par

    # Convert to a probability ('sigmoid' function)
    p = 1 / (1 + exp(-mu))

    # Convert to a class
    predicted_class = as.integer(
        ifelse(p > class_threshold, 'good', 'bad')
    )

    # Calculate the mean error
    value = mean(y - predicted_class)

    return(value)
}
```

Python

```
def misclassification_rate(par, X, y, class_threshold = .5):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate the 'linear predictor'
    mu = X @ par

    # Convert to a probability ('sigmoid' function)
    p = 1 / (1 + np.exp(-mu))

    # Convert to a class
    predicted_class = np.where(p > class_threshold, 1, 0)
```

```
    # Calculate the mean error
    value = np.mean(y - predicted_class)

    return value
```

Note that our function first adds a step to convert the linear combination of features, the linear predictor (called `mu`), to a probability. Once we have a probability, we use some threshold to convert it to a 'class'. In this case, we use 0.5 as the threshold, but this could be different depending on the context, something we talk more about elsewhere (Section 4.2.2). We'll leave it as an exercise for you to play around with this function, as the next objective function is far more commonly used. But at least you can see how easy it can be to switch from a numeric target to the classification case.

6.9.2 Log loss

A very common objective function for classification is **log loss**, sometimes called logistic loss, or **cross-entropy**[12]. For a binary target, it is:

$$\text{Value} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{6.5}$$

Where y_i is the actual value of the target for observation i, and $\hat{y}_i$ is the predicted value from the model (essentially a probability). It turns out that *this is the same as the log-likelihood used in a maximum likelihood approach for logistic regression*, made negative so we can minimize it.

We typically prefer this objective function to classification error because it results in a *smooth* optimization surface, like in the visualization we showed before for maximum likelihood (Section 6.7.1), which means it is *differentiable* in a mathematical sense. This is important because it allows us to use optimization algorithms that rely on derivatives in updating the parameter estimates. You don't really need to get into that too much, but just know that a smoother objective function is something we prefer. Here's some code to try out.

R

```
log_loss = function(par, X, y) {
    X = cbind(1, X)

    # Calculate the predicted values on the raw scale
```

[12] A nice demo from the pytorch perspective can be found at Raschka (2022a).

```
    y_hat = X %*% par

    # Convert to a probability ('sigmoid' function)
    y_hat = 1 / (1 + exp(-y_hat))

    # likelihood
    ll = y * log(y_hat) + (1 - y) * log(1 - y_hat)

    # alternative
    # dbinom(y, size = 1, prob = y_hat, log = TRUE)

    value = -sum(ll)

    return(value)
}
```

Python

```
def log_loss(par, X, y):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # Calculate the predicted values
    y_hat = X @ par

    # Convert to a probability ('sigmoid' function)
    y_hat = 1 / (1 + np.exp(-y_hat))

    # likelihood
    ll = y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat)

    value = -np.sum(ll)

    return value
```

Let's go ahead and demonstrate this. To create a classification problem, we'll say that a country is 'happy' if the happiness score is greater than 5.5, and 'unhappy' otherwise. We'll use the same features as before.

R

```
df_happiness_bin = df_happiness |>
    mutate(happiness = ifelse(happiness > 5.5, 1, 0))

model_logloss = optim(
    par = c(0, 0, 0, 0),
    fn = log_loss,
    X = df_happiness_bin |>
        select(life_exp_sc:gdp_pc_sc) |>
        as.matrix(),
    y = df_happiness_bin$happiness
)

model_glm = glm(
    happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc,
    data   = df_happiness_bin,
    family = binomial
)

model_logloss$par
```

Python

```
df_happiness_bin = df_happiness.copy()
df_happiness_bin['happiness'] = np.where(df_happiness['happiness'] > 5.5, 1, 0)

model_logloss = minimize(
    log_loss,
    x0 = np.array([0, 0, 0, 0]),
    args = (
        df_happiness_bin[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']],
        df_happiness_bin['happiness']
    )
)

model_glm = smf.glm(
    'happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc',
    data   = df_happiness_bin,
    family = sm.families.Binomial()
).fit()

model_logloss['x']
```

Once again, we can see that the results are very similar between our classification result and the built-in function.

Table 6.6: Comparison of Log Loss Results

parameter	Standard	Our result
LogLike	40.6635	40.6635
intercept	−0.1637	−0.1642
life_exp_sc	1.8172	1.8168
corrupt_sc	−0.4648	−0.4638
gdp_pc_sc	1.1311	1.1307

So, when it comes to classification, you should feel confident in what's going on under the hood, just like you did with a numeric target. Too much is made of the distinction between 'regression' and 'classification', and it can be confusing to those starting out. In reality, classification just requires a slightly different way of thinking about the target, not something fundamentally different about the modeling process.

6.10 Optimization Algorithms

When it comes to optimization, there are many algorithms that have been developed over time. The main thing to keep in mind is that these are all just different ways to find the best fitting parameters for a model. Some may be better suited for certain data tasks, or provide computational advantages, but usually the choice of algorithm is not as important as many other modeling choices you'll have to make.

6.10.1 Common methods

Here are some of the options available in R's `optim` or scipy's `minimize` function. In addition, they are commonly used behind the scenes in many modeling functions.

- Nelder-Mead
- BFGS
- L-BFGS-B (provides constraints)
- Conjugate gradient
- Simulated annealing
- Newton's method

Other common optimization methods include:

- Genetic algorithms
- Other popular SGD extensions and variants
  - RMSProp
  - Adam/momentum
  - AdaGrad

The main reason to choose one method over another usually is based on factors like speed, memory use, or how well the method works for certain models. For statistical contexts, many functions for generalized linear models use Newton's method by default, but more complicated models, for example, mixed models, may implement a different approach for better convergence. In machine learning, stochastic gradient descent is popular because it can be relatively efficient in large data settings and easy to implement.

In general, we can always try different methods to see which works best, but usually the results will be similar if the results reach convergence. We'll now demonstrate one of more common methods to get a sense of how these work.

6.10.2 Gradient descent

One of the most popular approaches in optimization is called **gradient descent**. It uses the gradient of the function we're trying to optimize to find the best parameters. We still use objective functions as before, and gradient descent is just a way to find that path along the objective function surface as we discussed previously in the deep dive (Section 6.7.1).

More formally, the gradient is the vector of partial derivatives of the objective function with respect to each parameter. That may not mean much to you, but the basic idea is that the gradient provides a direction that points in the direction of steepest increase in the function. So if we want to maximize the objective function, we can take a step in the direction of the gradient, and if we want to minimize it, we can take a step in the opposite direction of the gradient (use the negative gradient).

The size of the 'step' is called the **learning rate**. Like the penalty parameter we saw with penalized regression, it is a *hyperparameter* that we can tune through cross-validation (Section 10.7). If the learning rate is too small, it will take a longer time to converge. If it's too large, we might overshoot the objective and miss the best parameters. There are a number of variations on gradient descent that have been developed over time. Let's see this in action with the world happiness model.

R

```
gradient_descent = function(
    par,
    X,
    y,
    tolerance = 1e-3,
    maxit = 1000,
    learning_rate = 1e-3
) {

    X = cbind(1, X) # add a column of 1s for the intercept
    N = nrow(X)

    # initialize
    beta = par
    names(beta) = colnames(X)
    mse = crossprod(X %*% beta - y) / N  # crossprod provides sum(x^2)
    tol = 1
    iter = 1

    while (tol > tolerance && iter < maxit) {
        LP = X %*% beta
        grad = t(X) %*% (LP - y)
        betaCurrent = beta - learning_rate * grad
        tol = max(abs(betaCurrent - beta))
        beta = betaCurrent
        mse = append(mse, crossprod(LP - y) / N)
        iter = iter + 1
    }

    output = list(
        par    = beta,
        loss   = mse,
        MSE    = crossprod(LP - y) / nrow(X),
        iter   = iter,
        predictions = LP
    )

    return(output)
}
```

Python

```
def gradient_descent(
    par,
    X,
    y,
    tolerance = 1e-3,
    maxit = 1000,
    learning_rate = 1e-3
):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]

    # initialize
    beta = par
    loss = np.sum((X @ beta - y)**2)
    tol = 1
    iter = 1

    while (tol > tolerance and iter < maxit):
        LP = X @ beta
        grad = X.T @ (LP - y)
        betaCurrent = beta - learning_rate * grad
        tol = np.max(np.abs(betaCurrent - beta))
        beta = betaCurrent
        loss = np.append(loss, np.sum((LP - y)**2))
        iter = iter + 1

    output = {
        'par': beta,
        'loss': loss,
        'MSE': np.mean((LP - y)**2),
        'iter': iter,
        'predictions': LP
    }

    return output
```

With our functions in hand, let's apply them to the world happiness data. We'll keep the default settings, but feel free to play around with the `learning_rate` and `tolerance` parameters.

R

```
X = df_happiness |>
    select(life_exp_sc:gdp_pc_sc) |>
    as.matrix()

our_gd = gradient_descent(
    par = c(0, 0, 0, 0),
    X = X,
    y = df_happiness$happiness
)
```

Python

```
our_gd = gradient_descent(
    par = np.array([0, 0, 0, 0]),
    X = df_happiness[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']].to_numpy(),
    y = df_happiness['happiness'].to_numpy()
)
```

We compare our results in Table 6.7. As usual, we see that the results are very similar to the standard linear regression approach. Once again, we have demystified a step in the modeling process!

Table 6.7: Comparison of Gradient Descent Results

Value	Standard	Our Result
Intercept	5.445	5.437
life_exp_sc	0.525	0.521
corrupt_sc	−0.105	−0.107
gdp_pc_sc	0.438	0.439
MSE	0.367	0.367

In addition, when we visualize the loss function across iterations, we see a smooth decline in the MSE value as we go along each iteration (Figure 6.7). This is a good sign that we are converging to an optimal solution.

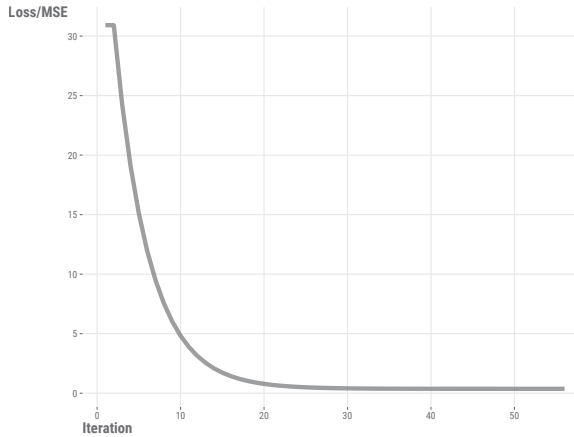


Figure 6.7: Loss with gradient descent.

6.10.3 Stochastic gradient descent

Stochastic gradient descent (SGD) is a version of gradient descent that uses a random sample of data to calculate the gradient, instead of using all the data. This makes it less accurate in some ways, but it's faster and can be parallelized across the CPU/GPU cores of the computing hardware environment. This speed is useful in machine learning when there's a lot of data, which often makes the discrepancy of results small between standard GD and SGD. As such, you will see variants of it incorporated in many models in deep learning, but know that it can be used with much simpler models as well.

Let's see this in action with the happiness model. The following is a conceptual version of the *AdaGrad* approach[13], which is a variation of SGD that adjusts the learning rate for each parameter. We will also add a variation that you can explore that averages the parameter estimates across iterations, which is a common approach to improve the performance of SGD.

We are going to use a **batch size** of one, which is similar to a 'streaming' or 'online' version where we update the model with each observation. Since our data are alphabetically ordered, we'll shuffle the data first. We'll also use a `stepsize_tau` parameter, which is a way to adjust the learning rate at early iterations. The values for the learning rate and `stepsize_tau` are arbitrary, selected after some initial playing around, but you can play with them to see how they affect the results.

[13]MC wrote this function a long time ago but does not recall exactly what the origin is, except that Murphy's PML book was something he was poring through at the time (Murphy (2012)).

R

```
stochastic_gradient_descent = function(
    par, # parameter estimates
    X,   # model matrix
    y,   # target variable
    learning_rate = 1, # the learning rate
    stepsize_tau = 0,   # if > 0, a check on the LR at early iterations
    seed = 123
) {
    # initialize
    set.seed(seed)

    # shuffle the data
    idx = sample(1:nrow(X), nrow(X))
    X = X[idx, ]
    y = y[idx]

    X = cbind(1, X)
    beta = par

    # Collect all estimates
    betamat = matrix(0, nrow(X), ncol = length(beta))

    # Collect fitted values at each point))
    fits = NA

    # Collect loss at each point
    loss = NA

    # adagrad per parameter learning rate adjustment
    s = 0

    # a smoothing term to avoid division by zero
    eps = 1e-8

    for (i in 1:nrow(X)) {
        Xi = X[i, , drop = FALSE]
        yi = y[i]

        # matrix operations not necessary here,
        # but makes consistent with previous gd func
        LP = Xi %*% beta
        grad = t(Xi) %*% (LP - yi)
```

```
        s = s + grad^2  # adagrad approach

        # update
        beta = beta - learning_rate /
            (stepsize_tau + sqrt(s + eps)) * grad
        betamat[i, ] = beta

        fits[i] = LP

        loss[i] = crossprod(LP - yi)

    }

    LP = X %*% beta
    lastloss = crossprod(LP - y)

    output = list(
        par = beta,           # final estimates
        par_chain = betamat,  # estimates at each iteration
        MSE = sum(lastloss) / nrow(X),
        predictions = LP
    )

    return(output)
}
```

Python

```
def stochastic_gradient_descent(
    par, # parameter estimates
    X,   # model matrix
    y,   # target variable
    learning_rate = 1, # the learning rate
    stepsize_tau = 0,  # if > 0, a check on the LR at early iterations
    average = False    # a variation of the approach
):
    # initialize
    np.random.seed(1234)

    # shuffle the data
    idx = np.random.choice(
        df_happiness.shape[0],
        df_happiness.shape[0],
```

```
        replace = False
    )
    X = X[idx, :]
    y = y[idx]

    X = np.c_[np.ones(X.shape[0]), X]
    beta = par

    # Collect all estimates
    betamat = np.zeros((X.shape[0], beta.shape[0]))

    # Collect fitted values at each point))
    fits = np.zeros(X.shape[0])

    # Collect loss at each point
    loss = np.zeros(X.shape[0])

    # adagrad per parameter learning rate adjustment
    s = 0

    # a smoothing term to avoid division by zero
    eps = 1e-8

    for i in range(X.shape[0]):
        Xi = X[None, i, :]
        yi = y[i]

        # matrix operations not necessary here,
        # but makes consistent with previous gd func
        LP = Xi @ beta
        grad = Xi.T @ (LP - yi)
        s = s + grad**2 # adagrad approach

        # update
        beta = beta - learning_rate / \
            (stepsize_tau + np.sqrt(s + eps)) * grad

        betamat[i, :] = beta

        fits[i] = LP
        loss[i] = np.sum((LP - yi)**2)

    LP = X @ beta
    lastloss = np.sum((LP - y)**2)
```

```
    output = {
        'par': beta,          # final estimates
        'par_chain': betamat, # estimates at each iteration
        'MSE': lastloss / X.shape[0],
        'predictions': LP
    }

    return output
```

Let's now use the functions as we did before. We'll show the results in Table 6.8.

R

```
X = df_happiness |>
    select(life_exp_sc, corrupt_sc, gdp_pc_sc) |>
    as.matrix()

y = df_happiness$happiness

our_sgd = stochastic_gradient_descent(
    par = c(mean(df_happiness$happiness), 0, 0, 0),
    X = X,
    y = y,
    learning_rate = .15,
    stepsize_tau = .1
)

c(our_sgd$par, our_sgd$MSE)
```

Python

```
X = df_happiness[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']].to_numpy()
y = df_happiness['happiness'].to_numpy()

our_sgd = stochastic_gradient_descent(
    par = np.array([np.mean(df_happiness['happiness']), 0, 0, 0]),
    X = X,
    y = y,
    learning_rate = .15,
    stepsize_tau = .1
)

our_sgd['par'], our_sgd['MSE']
```

Now we'll compare it to OLS estimates and our previous 'batch' gradient descent results. Even though SGD normally would not be used for such a small dataset, we at least get close[14]!

Table 6.8: Comparison of Stochastic Gradient Descent Results

Value	Standard	Our Result	Batch SGD
Intercept	5.445	5.469	5.437
life_exp_sc	0.525	0.514	0.521
corrupt_sc	−0.105	−0.111	−0.107
gdp_pc_sc	0.438	0.390	0.439
MSE	0.367	0.370	0.367

Figure 6.8 shows the estimates as they moved along the data. For this plot we don't include the intercept, as it's on a notably different scale. We can see that the estimates are moving around a bit, but they appear to be converging to a solution.

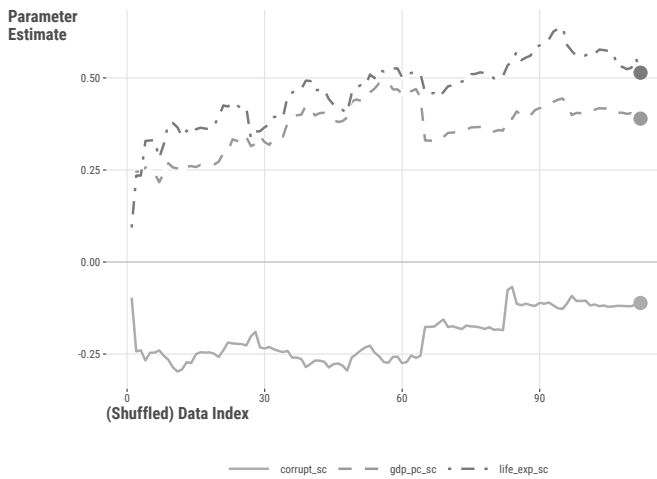


Figure 6.8: Stochastic gradient descent path.

To wrap things up, here are the results for the happiness model using different optimization algorithms, with a comparison to the standard linear regression

[14]You'd get better results in a couple of ways. The easiest is just to repeat the process a couple of times and average the results. This is a common approach in SGD. The initial shuffling that we did can help with convergence as well, and it would be done each repetition. With larger data and repeated runs/epochs, shuffling allows the samples/batches to be more representative of the entire dataset. Also, we 'hand-tune' our learning rate and step size here, but normally we'd use cross-validation to find the best values.

model function. We can see that the results are very similar, and for simpler modeling endeavors they should converge on the same result.

Table 6.9: Comparison of Optimization Results

parameter	NM[1]	BFGS[2]	CG[3]	GD[4]	Standard[5]
Intercept	5.445	5.445	5.445	5.437	5.445
life_exp_sc	0.525	0.525	0.525	0.521	0.525
gdp_pc_sc	−0.105	−0.105	−0.105	−0.107	−0.105
corrupt_sc	0.437	0.438	0.438	0.439	0.438
MSE	0.367	0.367	0.367	0.367	0.367

[1]NM = Nelder-Mead
[2]BFGS = Broyden–Fletcher–Goldfarb–Shanno
[3]CG = Conjugate gradient
[4]GD = Our gradient descent function
[5]Standard = Standard package function

Before leaving our estimation discussion, we should mention there are other approaches one could use to estimate model parameters, including variations on least squares, the **method of moments**, **generalized estimating equations**, robust estimation, and more. We've focused on the most common ones generally, but it's good to be aware of others that might be more popular in some domains.

6.11 Wrapping Up

Wow, we covered a lot here! But this is the sort of stuff that can take you from just having some fun with data, to doing that and also understanding how things are actually happening. Just having the basics of how modeling actually is done 'under the hood' makes so many other things make sense, and it can give you a lot of confidence, even in less familiar modeling domains.

6.11.1 The common thread

Simply put, the content in this chapter ties together any and every model you will ever undertake, from linear regression to reinforcement learning, computer vision, and large language models. Estimation and optimization are the core of any modeling process, and understanding the basics is key to understanding how models work in general.

6.11.2 Choose your own adventure

Seriously, after this chapter, you should feel fine with any of the others in this book, so dive in!

6.11.3 Additional resources

OLS and Maximum Likelihood Estimation:

For OLS and maximum likelihood estimation, there are so many resources out there, so we recommend just taking a look and seeing which one suits you best. Practically any more technically-oriented statistical book will cover these topics in detail.

- A list of classical references

Gradient Descent:

- Gradient Descent, Step-by-Step StatQuest with Josh Starmer (2019a)
- Stochastic Gradient Descent, Clearly Explained StatQuest with Josh Starmer (2019b)
- A Visual Explanation of Gradient Descent Methods Jiang (2020)

More demonstration of the simple AdaGrad algorithm used above:

- Brownlee (2021)
- DataBricks (2019)

6.12 Guided Exploration

For this exercise, you'll have two tasks:

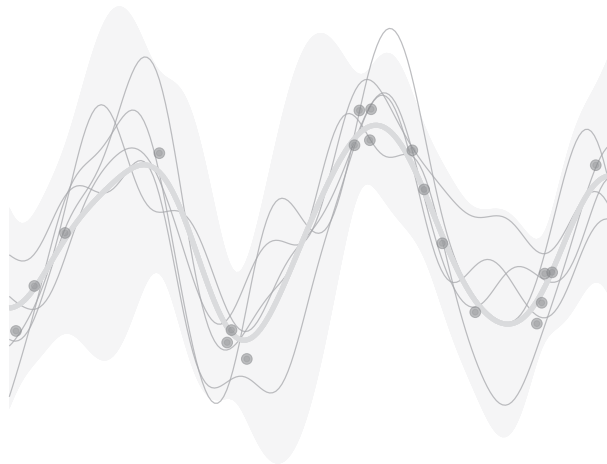
- Try creating an objective function for a continuous target that uses the mean absolute error, and compare your estimated parameters to the previous results for ordinary least squares. Use the OLS function from the chapter as a basis (Section 6.5), but modify it for the new objective.
- Use the estimation function we demonstrated for ridge regression (Section 6.8) and change it to use the lasso approach.

Both of these can be done by changing one line in the previous functions used.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

7

Estimating Uncertainty



Our focus thus far has been on estimating the best parameters for a model. But we also want to know how certain we are about those estimates. There are different ways to estimate **uncertainty**, and understanding the uncertainty in our results helps us make better decisions from our model. We'll briefly cover a few approaches here, but realize we are merely scratching the surface on these approaches. There are whole books, and even philosophies, dedicated to the topic of uncertainty estimation.

7.1 Key Ideas

- There are multiple ways to estimate uncertainty in parameters or prediction.
- Many statistical models provide formulaic interval estimates for parameters and predictions, couched in a **frequentist** framework.
- **Monte Carlo** methods use a simulation approach to estimate uncertainty.
- **Bootstrap** methods use resampling to estimate uncertainty.

- **Bayesian** methods provide a different way to estimate uncertainty and an alternative philosophical spirit.
- **Conformal** prediction provides a way to estimate uncertainty in predictions where other methods falter.

7.1.1 Why this matters

Understanding uncertainty is crucial for making decisions based on model results. It's difficult to make informed decisions if we don't know how certain we are about our estimates. This is especially important in high-stakes decisions, where the consequences of being wrong are severe. For example, in medical diagnosis, we want to be as certain as possible about the diagnosis before starting treatment. In finance, we want to be as certain as possible about the risk of an investment before making it. In all these cases, understanding uncertainty is key to making the best decision.

7.1.2 Helpful context

If you are comfortable with standard linear models, you should be okay here. This chapter does get a bit more technical and is more DIY than others, but the examples should prove straightforward.

7.2 Data Setup

Data setup follows the estimation chapter for consistency (Section 6.2).

R

```
df_happiness = read_csv('https://tinyurl.com/worldhappiness2018') |>
    drop_na() |>
    rename(happiness = happiness_score) |>
    select(
        country,
        happiness,
        contains('_sc')
    )
```

Python

```
import pandas as pd

df_happiness = (
    pd.read_csv('https://tinyurl.com/worldhappiness2018')
    .dropna()
    .rename(columns = {'happiness_score': 'happiness'})
    .filter(regex = '_sc|country|happ')
)
```

Nothing beyond base R is needed. For Python examples, the following are required.

```
import numpy as np

import statsmodels.api as sm
import statsmodels.formula.api as smf

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

from scipy import stats
```

7.3 Standard Frequentist

We talked a bit about the frequentist approach in our discussion of confidence intervals (Section 3.4.1). There we described the process using the interval to *capture* the 'true' parameter value a certain percentage of the time. The key assumption is that the true parameter is fixed, and the interval is a random variable that will contain the true value with some percentage frequency. With this approach, if you were to repeat the experiment, i.e., data collection and analysis, many times, each interval would be slightly different. Although they would be different, any one of the intervals is as good or valid as the others. You also know that a certain percentage of them will contain the true value, and a (usually small) percentage will not. You will never know if a specific interval does actually capture the true value, because we don't know the true value in practice.

This is a common approach in traditional statistical analysis, and so it's used in many modeling contexts. If no particular estimation approach is specified, the default is usually a frequentist one. The approach not only provides confidence

intervals for the parameters, but we can also get them for predictions, which, as we've seen elsewhere, is typically also a goal.

Here is an example using our previous model to get interval estimates for predictions. Here we get so-called 'confidence' or 'prediction' intervals. Both are confidence intervals in the frequentist sense, just for different purposes. The confidence interval is for the average prediction, while the prediction interval is for a future observation. The prediction interval is wider because it includes the uncertainty in the model parameters as well as the uncertainty in the prediction itself.

R

```
model = lm(
    happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc,
    data = df_happiness
)

confint(model)

predict(model, interval = 'confidence')  # for an average prediction
predict(model, interval = 'prediction')  # for a future observation (wider)
```

Python

```
model = smf.ols(
    'happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc',
    data = df_happiness
).fit()

model.conf_int()

# both 'confidence' and 'prediction' intervals
model.get_prediction().summary_frame()
```

The confidence interval is narrower because it only includes the uncertainty in the model parameters, while the prediction interval is wider because it includes the uncertainty in the model parameters and the prediction itself. The linear regression model provides these intervals by default, but we can also calculate them by hand. Here we show how to calculate the intervals for the predictions by hand by essentially performing the formula for the interval estimates. A sample of results are shown in Table 7.1.

R

```
X = model.matrix(model)

# get the prediction
y_hat = X %*% coef(model)

# get the standard error
se = sqrt(diag(X %*% vcov(model) %*% t(X)))

# critical value for 95% confidence
cv = qt(0.975, df = model$df.residual)

# get the confidence interval
tibble(
    prediction = y_hat[,1],
    lower = y_hat[,1] - cv * se,
    upper = y_hat[,1] + cv * se
) |>
    head()

predict(model, interval = 'confidence') |> head()

# get the prediction interval
se_pred = sqrt(se^2 + summary(model)$sigma^2)

data.frame(
    prediction = y_hat[,1],
    lower = y_hat[,1] - cv * se_pred,
    upper = y_hat[,1] + cv * se_pred
) |>
    head()

predict(model, interval = 'prediction') |> head()
```

Python

```
X = model.model.exog

# get the prediction
y_hat = X @ model.params

# get the standard error
se = np.sqrt(np.diag(X @ model.cov_params() @ X.T))
```

```
# critical value for 95% confidence
cv = stats.t.ppf(0.975, model.df_resid)

# get the confidence interval
pd.DataFrame({
    'prediction': y_hat,
    'lower': y_hat - cv * se,
    'upper': y_hat + cv * se
}).head()

model.get_prediction().summary_frame().head()

# get the prediction interval
se_pred = np.sqrt(se**2 + model.mse_resid)

pd.DataFrame({
    'prediction': y_hat,
    'lower': y_hat - cv * se_pred,
    'upper': y_hat + cv * se_pred
}).head()

model.get_prediction().summary_frame().head()
```

Table 7.1: Confidence and Prediction Interval Estimates

prediction	our_lwr	our_upr	lm_lwr	lm_upr
Confidence				
3.99	3.72	4.25	3.72	4.25
5.50	5.29	5.70	5.29	5.70
5.68	5.50	5.85	5.50	5.85
Prediction				
3.99	2.74	5.24	2.74	5.24
5.50	4.26	6.74	4.26	6.74
5.68	4.44	6.91	4.44	6.91

These interval estimates for parameters and predictions are actually not easy to get right for more complicated models beyond generalized linear models. Given this, one should be cautious when moving beyond standard linear models. The next two approaches we'll discuss are often used within the frequentist framework to estimate uncertainty in more complex models.

7.4 Monte Carlo

Monte Carlo methods derive their name from the famous casino in Monaco[1]. The idea is to use random sampling to estimate a value. With statistical models, we can use Monte Carlo methods to estimate uncertainty in our model parameters and predictions. The general idea is as follows:

1. **Estimate the model parameters** using the data and their range of possible values (e.g., based on a probability distribution).
2. **Simulate new data** from the model using the estimated parameters and assumed probability distributions for those parameters.
3. **Estimate the metrics of interest** using the simulated data.
4. **Repeat** many times.

The result is a distribution of the value of interest, be it a parameter, a prediction, or maybe an evaluation metric like RMSE. This distribution can then be used to provide a sense of uncertainty in the value, such as an interval estimate. We can use Monte Carlo methods to estimate the uncertainty in predictions for our happiness model as follows.

R

```
# we'll use the model from the previous section
model = lm(
    happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc,
    data = df_happiness
)

# number of simulations
mc_predictions = function(
    model,
    nsim = 2500,
    seed = 42
) {
    set.seed(seed)

    params_est = coef(model)
    params = mvtnorm::rmvnorm(
```

[1]The name originates with Stanislav Ulam, who worked on the Manhattan Project and would actually come up with the idea from playing solitaire. He is also the one who inspired the name of the Bayesian probabilistic programming language Stan!

```
        n = nsim,
        mean = params_est,
        sigma = vcov(model)
    )

    sigma = summary(model)$sigma
    X = model.matrix(model)

    y_hat = X %*% t(params) + rnorm(n = nrow(X) * nsim, sd = sigma)

    pred_int = apply(y_hat, 1, quantile, probs = c(.025, .975))

    return(pred_int)
}

our_mc = mc_predictions(model)
```

Python

```
# we'll use the model from the previous section
model = smf.ols(
    'happiness ~ life_exp_sc + corrupt_sc + gdp_pc_sc',
    data = df_happiness
).fit()

def mc_predictions(model, nsim=2500, seed=42):
    np.random.seed(seed)

    params_est = model.params
    params = np.random.multivariate_normal(
        mean = params_est,
        cov = model.cov_params(),
        size = nsim
    )

    sigma = model.mse_resid**.5
    X = model.model.exog

    y_hat = X @ params.T + \
        np.random.normal(scale = sigma, size = (X.shape[0], nsim))

    pred_int = np.quantile(y_hat, q = [.025, .975], axis = 1)
```

```
    return pred_int

our_mc = mc_predictions(model)
```

Here are the results of the Monte Carlo simulation for the prediction intervals. They are pretty close to what we'd already have available from the model package used for linear regression. However, we can use this for other models where uncertainty estimates are not readily available, providing a more general tool.

Table 7.2: Monte Carlo Prediction Intervals

observed_value	prediction	lower	upper	lower_lm	upper_lm
3.63	3.99	2.78	5.19	2.74	5.24
4.59	5.50	4.32	6.73	4.26	6.74
6.39	5.68	4.43	6.84	4.44	6.91
4.32	5.41	4.21	6.67	4.17	6.65
7.27	6.97	5.71	8.19	5.72	8.21
7.14	6.88	5.63	8.13	5.64	8.12

Results based on the R simulation.

Monte Carlo simulation is a very popular approach in modeling, and a variant of it, **Markov Chain Monte Carlo** (MCMC), is the basis for Bayesian estimation, which we'll also talk about in more detail later.

7.5 Bootstrap

An extremely common method for estimating uncertainty is the **bootstrap**. The bootstrap is a method where we create new datasets by randomly sampling the original data with replacement. This means that each new dataset is the same size as the original, but some observations may be selected multiple times, while others may not be selected at all. We then estimate our model with each dataset, and each time, we can collect parameter estimates, predictions, or any other calculations we are interested in. Ultimately, we end up with a *distribution* of all the things we calculated. The nice thing about this is that we don't need to know the specific distribution (e.g., normal, or t-distribution) of the values we want to get uncertainty estimates for, we can just use the data we have to produce that distribution. And this is a key distinction from the Monte Carlo method just discussed.

The results of bootstrapping give us a range of possible values, which is useful for inference[2], as we can use the distribution to calculate interval estimates. The average parameter estimate is typically the same as whatever the underlying model used would produce, so not really useful for that in the context of simpler linear models. Even so, we can calculate derivatives of the parameters, like say a ratio or sum, or a model metric like R^2, or a prediction. Some of these normally would not be estimated as part of the model, or maybe the model tool does not provide anything beyond the value itself. Yet the bootstrap provides a way to get at a measure of uncertainty for the values of interest, with fewer assumptions about how that distribution should take shape.

The approach is very flexible, and it can potentially be used with any model whether in a statistical or machine learning context. Let's see this in action with the happiness data. We'll create a bootstrap function, then we'll use it to estimate the uncertainty in the coefficients for the model.

R

```
bootstrap = function(X, y, nboot = 100, seed = 123) {

    N = nrow(X)
    p = ncol(X) + 1 # add one for intercept

    # initialize
    beta = matrix(NA, p*nboot, nrow = nboot, ncol = p)
    colnames(beta) = c('Intercept', colnames(X))
    mse = rep(NA, nboot)

    # set seed
    set.seed(seed)

    for (i in 1:nboot) {
        # sample with replacement
        idx = sample(1:N, N, replace = TRUE)
        Xi = X[idx,]
        yi = y[idx]

        # estimate model
        mod = lm(yi ~., data = Xi)

        # save results
```

[2]We're using inference here in the standard statistical/philosophical sense, not as a synonym for prediction or generalization, which is how it is often used in machine learning. We're not exactly sure how that terminological muddling arose in ML, but be on the lookout for it.

```
        beta[i, ] = coef(mod)
        mse[i] = sum((mod$fitted - yi)^2) / N
    }

    # given mean estimates, calculate MSE
    y_hat = cbind(1, as.matrix(X)) %*% colMeans(beta)
    final_mse = sum((y - y_hat)^2) / N

    output = list(
        par = as_tibble(beta),
        MSE = mse,
        final_mse = final_mse
    )

    return(output)
}

X = df_happiness |>
    select(life_exp_sc:gdp_pc_sc)

y = df_happiness$happiness

our_boot = bootstrap(
    X = X,
    y = y,
    nboot = 1000
)
```

Python

```
def bootstrap(X, y, nboot=100, seed=123):
    # add a column of 1s for the intercept
    X = np.c_[np.ones(X.shape[0]), X]
    N = X.shape[0]

    # initialize
    beta = np.empty((nboot, X.shape[1]))

    # beta = pd.DataFrame(beta, columns=['Intercept'] + list(cn))
    mse = np.empty(nboot)

    # set seed
    np.random.seed(seed)
```

```
    for i in range(nboot):
        # sample with replacement
        idx = np.random.randint(0, N, N)
        Xi = X[idx, :]
        yi = y[idx]

        # estimate model
        model = LinearRegression(fit_intercept=False) # from sklearn
        mod = model.fit(Xi, yi)

        # save results
        beta[i, :] = mod.coef_
        mse[i] = np.sum((mod.predict(Xi) - yi)**2) / N

    # given mean estimates, calculate MSE
    y_hat = X @ beta.mean(axis=0)
    final_mse = np.sum((y - y_hat)**2) / N

    output = {
        'par': beta,
        'mse': mse,
        'final_mse': final_mse
    }

    return output

our_boot = bootstrap(
    X = df_happiness[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']],
    y = df_happiness['happiness'],
    nboot = 1000
)
```

Here are the results of the interval estimates for the coefficients in Table 7.3. Each parameter has the mean estimate, the lower and upper bounds of the 95% confidence interval, and the width of the interval. The bootstrap intervals are a bit wider than the OLS intervals, but for this model these should converge as the number of observations increases.

Table 7.3: Bootstrap Parameter Estimates

Parameter	mean	Lower BS	Upper BS	Lower OLS	Upper OLS	Diff Width[1]
Intercept	5.45	5.34	5.55	5.33	5.56	−0.01
life_exp_sc	0.52	0.30	0.74	0.35	0.70	0.09
corrupt_sc	−0.11	−0.30	0.08	−0.25	0.04	0.09
gdp_pc_sc	0.45	0.17	0.76	0.24	0.64	0.19

[1]Width of bootstrap estimate minus width of OLS estimate.

Let's look more closely at the distributions for each coefficient in Figure 7.1. Standard statistical estimates assume a specific distribution like the normal. But the bootstrap method provides more flexibility, even though it often leans toward the assumed distribution. We can see these distributions aren't perfectly symmetrical like a normal distribution, but they suit our needs in that we can extract the lower and upper quantiles to create an interval estimate.

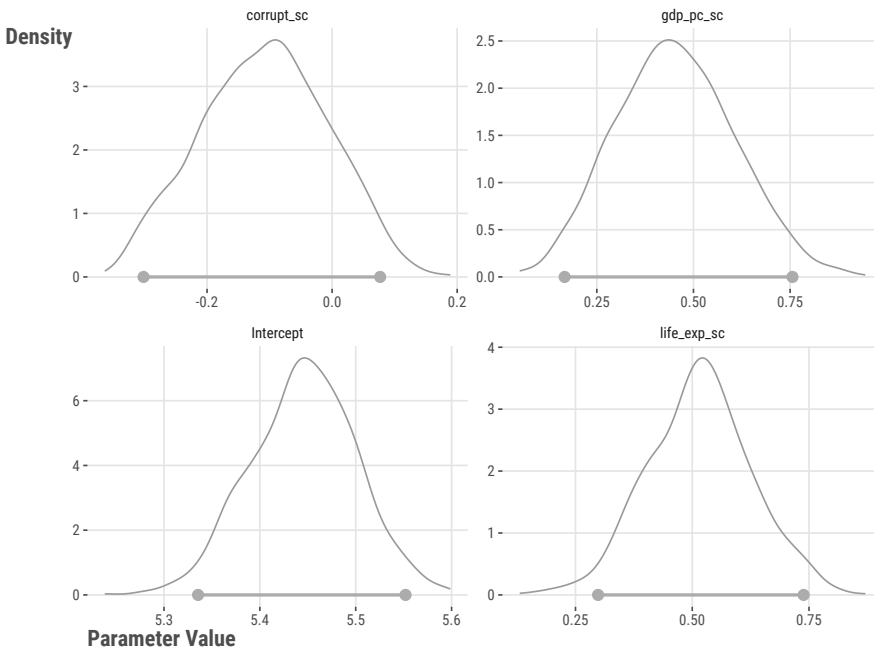


Figure 7.1: Bootstrap distributions of parameter estimates.

As mentioned, the bootstrap is often used to provide uncertainty for unmodeled parameters, predictions, and other metrics. However, because we repeatedly run the model or some aspect of it over and over, it is computationally inefficient, and might not be suitable with large data sizes. It also may not estimate the appropriate uncertainty for some types of statistics (e.g., extreme values) or in

some data contexts (e.g., correlated observations) without extra considerations. Variants exist to help deal with some of these issues, and despite limitations, the bootstrap method is a useful tool and can be used together with other methods to understand uncertainty in a model.

7.6 Bayesian

The **Bayesian** approach to modeling is many things: a philosophical viewpoint, an entirely different way to think about probability, a different way to measure uncertainty, and on a practical level, just another way to get model parameter estimates. It can be as frustrating as it is fun to use, and one of the really nice things about using Bayesian estimation is that it can handle model complexities that other approaches don't do well or at all.

The basis of Bayesian estimation is the **likelihood**, the same as with maximum likelihood, and everything we did there applies here. So you need a good grasp of maximum likelihood to understand the Bayesian approach. However, the Bayesian approach is different because it also lets us use our knowledge about the parameters through **prior distributions**. For example, we may think that the coefficients for a linear model come from a normal distribution centered on zero with some variance. That would serve as our prior distribution for those parameters.

The combination of a prior distribution with the likelihood results in the **posterior distribution**, which is a *distribution* of possible parameter values. It falls somewhere between the prior and the likelihood. With more data, it tends toward the likelihood result, and with less data, it tends toward what the prior would have suggested. The posterior distribution is what we ultimately use to make inferences about the parameters, and it can be used to estimate uncertainty in the same way as the bootstrap.

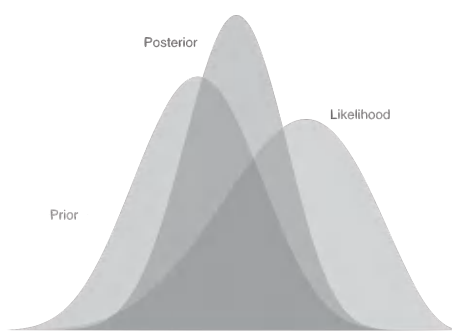


Figure 7.2: Prior, likelihood, and posterior distributions.

Example

Let's do a simple example to show how this comes about. We'll use a binomial model where we have penalty kicks taken for a soccer player, and we want to estimate the probability of the player making a goal, which we'll call θ.

For our prior distribution, we'll use a beta distribution that has a mean of 0.5, suggesting that we think this person would have about a 50% chance of converting the kick on average. However, we will keep this prior fairly loose, with a range that spans most of the (0, 1) interval. For the likelihood, we'll use a binomial distribution. We also use this in our discussion of generalized linear models (see Equation 8.3), which, as we have also noted, is akin to using the log loss (Section 6.9.2). We'll then calculate the posterior distribution for the probability of making a shot, given our prior and the evidence at hand, i.e., the data.

Let's start with some data, and just like our other estimation approaches, we'll have some guesses for θ which represents the probability of making a goal. We'll use the prior distribution to represent our beliefs about those parameter values, assigning more weight to values around 0.5. We'll then calculate the likelihood of the data given the parameter, which will put more weight on values closer to the observed chance of scoring a goal. Finally, we calculate the posterior distribution.

R

```
pk = c(
    'goal','goal','goal','miss','miss',
    'goal','goal','miss','goal','goal'
)

# convert to numeric, arbitrarily picking goal=1, miss=0

N = length(pk)               # sample size
n_goal = sum(pk == 'goal')   # number of pk made
n_miss = sum(pk == 'miss')   # number of those miss

# grid of potential theta values
theta = seq(
    from = 1 / (N + 1),
    to = N / (N + 1),
    length = 10
)

### prior distribution
# beta prior with mean = .5, but fairly diffuse
```

```
# examine the prior
# theta = rbeta(1000, 5, 5)
# hist(theta, main = 'Prior Distribution', xlab = 'Theta', col = 'lightblue')
p_theta = dbeta(theta, 5, 5)

# Normalize so that values sum to 1
p_theta = p_theta / sum(p_theta)

# likelihood (binomial)
p_data_given_theta = choose(N, n_goal) * theta^n_goal * (1 - theta)^n_miss

# posterior (combination of prior and likelihood)
# p_data is the marginal probability of the data used for normalization
p_data = sum(p_data_given_theta * p_theta)

p_theta_given_data = p_data_given_theta*p_theta / p_data  # Bayes theorem

# final estimate
theta_est = sum(theta * p_theta_given_data)
theta_est
```

```
[1] 0.6
```

Python

```
from scipy.stats import beta

pk = np.array([
    'goal','goal','goal','miss','miss',
    'goal','goal','miss','goal','goal'
])

# convert to numeric, arbitrarily picking goal=1, miss=0
N = len(pk)                     # sample size
n_goal = np.sum(pk == 'goal')   # number of pk made
n_miss = np.sum(pk == 'miss')   # number of those miss

# grid of potential theta values
theta = np.linspace(1 / (N + 1), N / (N + 1), 10)

### prior distribution
# beta prior with mean = .5, but fairly diffuse
# examine the prior
# theta = beta.rvs(5, 5, size = 1000)
```

```
# plt.hist(theta, bins = 20, color = 'lightblue')
p_theta = beta.pdf(theta, 5, 5)

# Normalize so that values sum to 1
p_theta = p_theta / np.sum(p_theta)

# likelihood (binomial)
p_data_given_theta = np.math.comb(N, n_goal) * theta**n_goal * \
    (1 - theta)**n_miss


# posterior (combination of prior and likelihood)
# p_data is the marginal probability of the data used for normalization
p_data = np.sum(p_data_given_theta * p_theta)

p_theta_given_data = p_data_given_theta * p_theta / p_data  # Bayes theorem

# final estimate
theta_est = np.sum(theta * p_theta_given_data)
theta_est
```

```
0.599999996503221
```

Table 7.4 that puts all this together. Our prior distribution is centered around a θ of 0.5 because we made it that way. The likelihood is centered closer to 0.7 because that's the observed chance of scoring a goal. The posterior distribution is a combination of the two. It gives no weight to smaller values, or to the max value. Our final estimate is 0.6, which falls between the prior and likelihood values that have the most weight. With more evidence in the form of data, our estimate will shift more and more toward what the likelihood would suggest. This is a simple example, but it shows how the Bayesian approach works, and this conceptually holds for more complex parameter estimation as well.

Table 7.4: Bayesian Demo Results

theta	prior	like	post
0.09	0.00	0.00	0.00
0.18	0.03	0.00	0.00
0.27	0.09	0.01	0.00
0.36	0.16	0.03	0.03
0.45	0.22	0.08	0.14
0.55	0.22	0.16	0.28
0.64	0.16	0.24	0.32
0.73	0.09	0.26	0.19
0.82	0.03	0.18	0.04
0.91	0.00	0.05	0.00

> **i** Priors as Regularization
>
> In the context of penalized estimation and machine learning, the prior distribution can be thought of as a form of **regularization** (See Section 6.8 and Section 10.5 later). In this context, the prior shrinks the estimate, pulling the parameter estimates toward it, just like the penalty parameter does in the penalized estimation methods. In fact, many penalized methods can be thought of as a Bayesian approach with a specific prior distribution. An example would be ridge regression, which can be thought of as a Bayesian linear regression with a normal prior distribution for the coefficients. The variance of the prior is inversely related to the ridge penalty parameter.

Application

Just like with the bootstrap which also provided distributions for the parameters, we can use the Bayesian approach to understand how certain we are about our estimates. We can look at any range of values in the posterior distribution to get what is often referred to as a **credible interval**, which is the Bayesian equivalent of a confidence interval[3]. Here is an example of the posterior distribution for the parameters of our happiness model, along with 95% intervals[4].

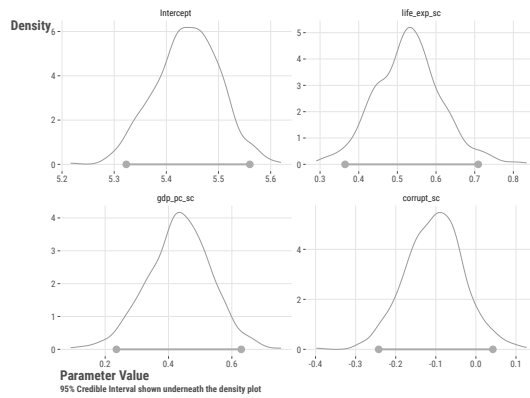


Figure 7.3: Posterior distribution of parameters.

[3]Many people's default interpretation of a standard confidence interval is usually something like 'the range we expect the parameter to reside within'. Unfortunately, that's not quite right, though it is how you interpret the Bayesian interval. The frequentist confidence interval is a range that, if we were to repeat the experiment/data collection many times, contains the true parameter value a certain percentage of the time. For the Bayesian, the parameter is assumed to be random, and so the interval is that which we expect the parameter to fall within a certain percentage of the time. The Bayesian is probably a bit more intuitive for most, even if it's not the more widely used.

[4]We used the R package for brms for these results.

With Bayesian estimation we also provide starting values for the algorithm, which is a form of Monte Carlo estimation[5], to get things going. We also typically specify a number of iterations, or times the model will run, as the **stopping rule**. Each iteration gives us a new guess for each parameter, which amounts to a random draw from the posterior distribution. With more iterations the model takes longer to run, but the length often reflects the complexity of the model.

We also specify multiple **chains**, which do the same estimation procedure, but due to the random nature of the Bayesian approach and starting point, take different estimation paths[6]. We can then compare the chains to see if they are converging to the same result, which is a check on the model. If they are not converging, we may need to run the model longer, or it may indicate a problem with how we set up the model.

Here's an example of the four chains for our happiness model for the life expectancy coefficient. The chains bounce around a bit from one iteration to the next, but on average, they're giving very similar results, so we know the model is working well. Nowadays, we have default statistics in the output that also provide this information, which makes it easier to quickly check convergence for many parameters.

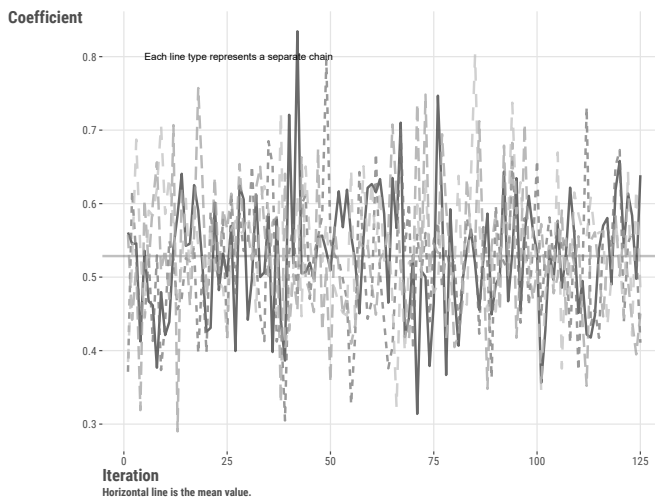


Figure 7.4: Bayesian chains for life expectancy coefficient.

[5]The most common method for the Bayesian approach is Markov Chain Monte Carlo (MCMC), which is a way to sample from the posterior distribution. There are many MCMC algorithms, many of which are a form of the now fairly old Metropolis-Hastings algorithm, which you can find a demo of at Michael's doc.

[6]Some deep learning implementations will use multiple random starts for similar reasons.

When we are interested in making predictions, we can use the results to generate a distribution of possible predictions *for each observation*, which can be very useful when we want to quantify uncertainty for complex models. This is referred to as **posterior predictive distribution**, which is explored in a non-Bayesian context in Section 4.4. Here is a plot of several draws of predicted values against the true happiness scores.

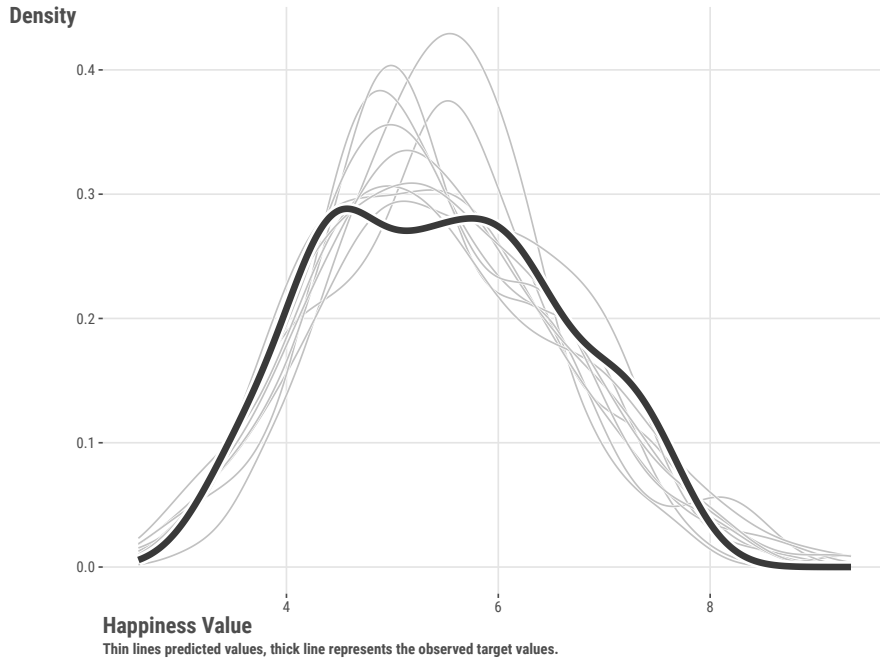


Figure 7.5: Posterior predictive distribution of happiness values.

With the Bayesian approach, every metric we calculate has a range of possible values, not just one. For example, if you have a classification model and want to know the accuracy, AUROC, or true positive rate of the model, instead of a single number, you would now have access to a whole distribution of values for that metric. How? For each possible set of model parameters from the posterior distribution, we apply those values and model to data to make a prediction. We can then assign it to a class, and compare it to the actual class. This gives us a range of possible predictions and classes. We can then calculate metrics like accuracy or true positive rate for each possible prediction set. As an example, we did this for our happiness model with a numeric target to obtain the interval estimate for R-squared in Table 7.5. Pretty neat!

Table 7.5: Bayesian R^2

Estimate	Lower	Upper
0.71	0.65	0.75

95% Credible interval for R-squared.

> **Frequentist PP check**
>
> As we saw in Section 4.4, nothing is keeping you from doing 'predictive checks' with other estimation approaches, and it's a very good idea to do so. For example, with a GLM you can use Monte Carlo simulation or the Bootstrap to generate a distribution of predictions, and then compare that to the actual data. This is a good way to check the model's assumptions and see if it's doing what you think it's doing. It's more straightforward with the Bayesian approach, since many modeling packages will do it for you with little effort.

Additional Thoughts

It turns out that any standard (frequentist) statistical model can be seen as a Bayesian one from a certain point of view[7]. Here are a couple.

- GLM and related models estimated via maximum likelihood: Bayesian estimation with a flat/uniform prior on the parameters.
- Ridge Regression: Bayesian estimation with a normal prior on the coefficients, penalty parameter is related to the variance of the prior.
- Lasso Regression: Bayesian estimation with a Laplace prior on the coefficients, penalty parameter is related to the variance of the prior.
- Mixed Models: Random effects are, as the name suggests, random, and they are estimated as a distribution of possible values, which is conceptually in line with the Bayesian approach.

So, in many modeling contexts, you're actually doing a restrictive form of Bayesian estimation already.

The Bayesian approach is very flexible, can be used for many different types of models, and can be used to get at uncertainty in a model in ways that other approaches can't. It's not always the best approach, even when appropriate due to the computational burden and just diagnostic complexity, but it's a good one to have in your toolbox[8]. Hopefully we've helped to demystify the Bayesian approach a bit here, and you feel more comfortable trying it out.

[7] Cue Obi-Wan Kenobi.

[8] R has excellent tools here for modeling and post-processing, like brms and tidybayes, and Python has pymc3, numpyro, and arviz, which are also useful. Honestly, R has way

7.7 Conformal Methods

Conformal approaches bring us back to the frequentist world, and specifically regard prediction uncertainty. One of the primary strengths of the approach is that it is model agnostic and theoretically can work for any model, from linear regression to deep learning. Like the bootstrap and Bayesian methods, conformal prediction makes us think in terms of distributions of possible values, but it focuses on residuals or errors in prediction.

It is based on the idea that we can estimate the uncertainty in our predictions by looking at the distribution of the predictions from the model, or more specifically, the prediction error. Using the observed prediction error on a calibration set that was not used to train the model, we can order those errors and find the quantile corresponding to the desired uncertainty coverage/error rate[9]. When predicting on new data, we assume the predictions and corresponding errors come from a similar distribution as what we've seen already in our training/calibration process. We do this with no particular assumption about that distribution. We then use the estimated quantile to create upper and lower bounds for a prediction for a new observation.

While the implementation for various settings can get quite complicated, the conceptual approach is mostly straightforward. As an example, we can demonstrate the **split-conformal** procedure with the following steps.

1. **Split Data**: Split the dataset into training and calibration sets.
2. **Train Model**: Train the model using the training set.
3. **Calculate Scores**: Calculate conformity scores on the calibration set. These are the *absolute* residuals between the predicted and actual values on the calibration set.
4. **Quantile Calculation**: Determine the quantile value of the conformity scores for the desired confidence level.
5. **Generate Intervals**: Generate prediction intervals for new data points. For new data points, use the trained model to make predictions. Adjust these predictions by adding and subtracting the quantile value obtained from the conformity scores to generate the lower and upper bounds of the prediction intervals.

Let's now demonstrate the split-conformal method with our happiness model. We'll start by defining the split-conformal function. The function takes the training data, the target variable, and new data for which we want to make

more going on here, with many packages devoted to Bayesian estimation of specific models even, but if you want to stick with Python it's gotten a lot better recently.

[9]The error rate (α) is the proportion of the data that would fall outside the prediction interval, while the coverage rate/interval is 1 - α.

predictions. It also takes an α value, which is the error rate we want to control, and a calibration split, which is the proportion of the data we use for calibration. And finally, we designate new data for which we want to make predictions.

R

```
split_conformal = function(
    X,
    y,
    new_data,
    alpha = .05,
    calibration_split = .5
) {
    # Splitting the data into training and calibration sets
    idx = sample(1:nrow(X), size = floor(nrow(X) / 2))

    train_data = X |> slice(idx)
    cal_data   = X |> slice(-idx)
    train_y = y[idx]
    cal_y   = y[-idx]

    N = nrow(train_data)

    # Train the base model
    model = lm(train_y ~ ., data = train_data)

    # Calculate residuals on calibration set
    cal_preds = predict(model, newdata = cal_data)
    residuals = abs(cal_y - cal_preds)

    # Sort residuals and find the quantile corresponding to (1-alpha)
    residuals = sort(residuals)
    quantile  = quantile(residuals, (1 - alpha) * (N / (N + 1)))

    # Make predictions on new data and calculate prediction intervals
    preds = predict(model, newdata = new_data)
    lower_bounds = preds - quantile
    upper_bounds = preds + quantile

    # Return predictions and prediction intervals
    return(
        list(
            cp_error     = quantile,
            preds        = preds,
```

```
            lower_bounds = lower_bounds,
            upper_bounds = upper_bounds
        )
    )
}
```

Python

```
def split_conformal(X, y, new_data, alpha = .05, calibration_split = .5):
    # Splitting the data into training and calibration sets
    X_train, X_cal, y_train, y_cal = train_test_split(
        X,
        y,
        test_size = calibration_split,
        random_state = 123
    )

    N = X_train.shape[0]

    # Train the base model
    model = LinearRegression().fit(X_train, y_train)

    # Calculate residuals on calibration set
    cal_preds = model.predict(X_cal)
    residuals = np.abs(y_cal - cal_preds)

    # Sort residuals and find the quantile corresponding to (1-alpha)
    residuals = np.sort(residuals)

    # The correction here is useful for small sample sizes
    quantile  = np.quantile(residuals, (1 - alpha) * (N / (N + 1)))

    # Make predictions on new data and calculate prediction intervals
    preds = model.predict(new_data)
    lower_bounds = preds - quantile
    upper_bounds = preds + quantile

    # Return predictions and prediction intervals
    return {
        'cp_error': quantile,
        'preds': preds,
        'lower_bounds': lower_bounds,
```

```
        'upper_bounds': upper_bounds
    }
```

With our functions in place, we can now use them to calculate the prediction intervals for the happiness model. The `cp_error` value gives us the quantile value that we use to generate the prediction intervals. Raw result is not shown, but Table 7.6 shows the first few predictions and their corresponding prediction intervals.

R

```
# split data
set.seed(123)

idx_train = sample(nrow(df_happiness), nrow(df_happiness) * .8)
idx_test = setdiff(1:nrow(df_happiness), idx_train)

df_train = df_happiness |>
    slice(idx_train) |>
    select(happiness, life_exp_sc, gdp_pc_sc, corrupt_sc)

y_train = df_happiness$happiness[idx_train]

df_test = df_happiness |>
    slice(idx_test) |>
    select(life_exp_sc, gdp_pc_sc, corrupt_sc)

y_test = df_happiness$happiness[idx_test]

# apply the function
cp_error = split_conformal(
    df_train |> select(-happiness),
    y_train,
    df_test,
    alpha = .1
)

# cp_error[['cp_error']]

tibble(
    prediction = cp_error[['preds']],
    lower_bounds = cp_error[['lower_bounds']],
    upper_bounds = cp_error[['upper_bounds']]
```

```
) |>
    head()
```

Python

```
# split data
X = df_happiness[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']]
y = df_happiness['happiness']

X_train, X_test, y_train, y_test = train_test_split(
    df_happiness[['life_exp_sc', 'corrupt_sc', 'gdp_pc_sc']],
    df_happiness['happiness'],
    test_size = 0.5,
    random_state = 123
)

our_cp_error = split_conformal(
    X_train,
    y_train,
    X_test,
    alpha = .1
)

# print(our_cp_error['cp_error'])

pd.DataFrame({
    'prediction': our_cp_error['preds'],
    'lower_bounds': our_cp_error['lower_bounds'],
    'upper_bounds': our_cp_error['upper_bounds']
}).head()
```

Table 7.6: Split-Conformal Prediction Intervals

prediction	lower bound	upper bound
4.04	2.94	5.13
5.27	4.18	6.37
6.84	5.74	7.94
4.34	3.24	5.44
4.15	3.05	5.24
7.26	6.16	8.36

Result based on the R code.

As a method of uncertainty estimation, conformal prediction is not without its challenges. It is computationally intensive for large datasets or complex models. There are multiple variants of conformal prediction, most of which attempt to alleviate a deficiency of simpler approaches. But they generally further increase the computational burden.

Conformal prediction still relies on the assumptions about the data and the underlying model, and violations of these assumptions can lead to invalid prediction intervals. Furthermore, conformal prediction methods assume that the training and test data come from the same distribution, which may not always be the case in real-world applications due to distribution shifts or domain changes. In addition, validation sets must be viable splits of the data, which default splitting methods may not always provide. In general, conformal prediction provides an alternative to other frequentist or Bayesian approaches that, under the right circumstances, may produce a better estimate of uncertainty, but it does not come for free.

7.8 Wrapping Up

Understanding uncertainty is key to understanding the quality of your model. It's not just about the point estimate, or getting a prediction, but also about how confident you are in value. We've covered several avenues from the basics of estimation to the more complex Bayesian and conformal methods. If the model provides a standard statistical solution, take it. Otherwise, the bootstrap is easy to understand and implement. Bayesian methods are more complex but can provide more information about the uncertainty in your model. Conformal prediction is a good choice when you want to make predictions without making strong assumptions about the underlying model, and may be the best option for many contexts.

We hope you now have a better understanding of how to estimate uncertainty in your models, and how to use that information to make better decisions.

7.8.1 The common thread

No model is without uncertainty, so any of these techniques may be applicable to your work. The choice of method depends largely on how you want to tackle the issue.

7.8.2 Choose your own adventure

This chapter colors all others that focus on specific modeling techniques. You can think about how you might implement uncertainty estimation for any of them.

7.8.3 Additional resources

Frequentist Approaches:

- Most statistical texts cover uncertainty estimation from the frequentist perspective. Pick one you like.
- Error Statistics Deborah Mayo's blog and comments on other blogs have always provided a strong philosophical defense of frequentist statistics.

Monte Carlo:

- Monte Carlo Methods John Guttag's MIT Course lecture on YouTube.

Bootstrap:

Classical treatments:

- Efron and Tibshirani (1994)
- Davison and Hinkley (1997)

A more fun demo:

- Bootstrapping Main Ideas StatQuest with Josh Starmer (2021)

Bayesian:

- Bayesian Data Analysis, Gelman et al. (2013). For many, this is the Bayesian bible.
- Statistical Rethinking, McElreath (2020). A fantastic modeling book, Bayesian or otherwise.
- Choosing priors

Conformal Prediction:

General:

- A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification, Angelopoulos and Bates (2022); Example Python Notebooks

R demos:

- Conformal inference for regression models
- Conformal prediction

Python demos:

- Introduction To Conformal Prediction With Python, Molnar (2024)
- Mapie Docs

Other: - Sources of Uncertainty in Machine Learning – A Statisticians' View, Gruber et al. (2023)

7.9 Guided Exploration

We find that simulation is a great way to understand models, and the Monte Carlo approach to uncertainty definitely puts simulation at the forefront. The next chapter focuses on generalized linear models, so if you're not familiar with logistic regression, head there first. If you are familiar, see if you can apply the Monte Carlo approach to get predicted probabilities for a logistic regression model. You really only need to change two lines from our previous code.

R

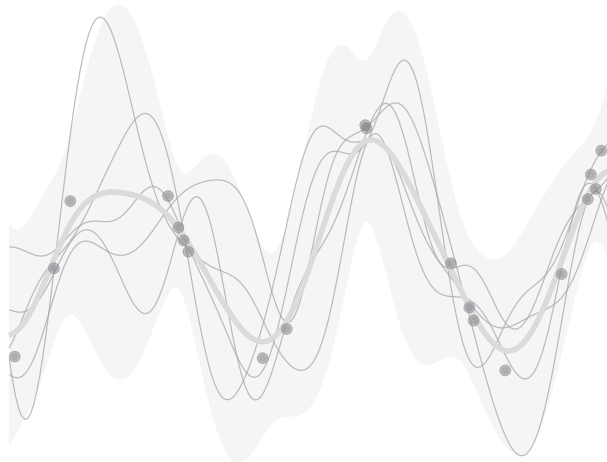
```
mc_predictions = function(
    model,
    nsim = 2500,
    seed = 42
) {
    ...
    # we aren't dealing with a normal distribution for this
    # how should we change this line?
    yhat =  X %*% t(params) + rnorm(n = nrow(X) * nsim, sd = sigma)

    # how do we get probabilities from this?
    ???? = ????

    # proceed as before
    pred_int = apply(y_hat, 1, quantile, probs = c(.025, .975))

    return(pred_int)
}
```

Python

```
def mc_predictions(model, nsim=2500, seed=42):
    ...
    # we aren't dealing with a normal distribution for this
    # how should we change this line?
    yhat = X @ params + \
        np.random.normal(scale = sigma, size = (X.shape[0], nsim))

    # how do we get probabilities from this?
    ???? = ????

    # proceed as before
    pred_int = np.quantile(yhat, q = [.025, .975], axis = 1)

    return pred_int
```

8

Generalized Linear Models



What happens when your target variable isn't really something you feel comfortable modeling with a normal distribution? Maybe you've got a binary condition, like good or bad, or maybe you've got a skewed count of something, like the number of times a person who has been arrested has been reincarcerated. In these cases, you can use a linear regression, but it often won't get you exactly what you want in terms of predictive performance. Instead, you can *generalize* your approach to handle these scenarios.

Generalized linear models allow us to implement different probability distributions, taking us beyond the normal distribution that is assumed for linear regression. This allows us to use the *same* linear model framework that we've been using, but with different types of targets. As such, these models *generalize* the linear model to better capture the nuances of different types of feature-target relationships.

8.1 Key Ideas

- A simple tweak to our previous approach allows us to generalize our linear model to account for other types of target data.
- Common distributions such as binomial, Poisson, and others can often improve model fit and interpretability.
- Getting familiar with just a couple of distributions will allow you to really expand your modeling repertoire.

8.1.1 Why this matters

The linear model is powerful on its own, but even more so when you realize you can extend it to many other data settings, some of which may have implicitly nonlinear feature-target relationships! When we want to classify observations, count them, or deal with proportions and other things, very simple tweaks of our standard linear model allow us to handle such situations.

8.1.2 Helpful context

Generalized linear models are a broad class of models that extend the linear model to different distributions of the target variable. In general, you'd need to have a pretty good grasp of linear regression before getting too carried away here.

8.2 Distributions and Link Functions

Remember how linear regression models really enjoy the whole Gaussian, i.e., 'normal', distribution scene? We saw that the essential form of the linear model can be expressed as follows. With probabilistic models such as these, the formula is generally expressed as $y|X, \theta \sim ...$, where X is the matrix of features (data) and θ the parameters estimated by the model. We simplify this as y^* here.

$$y|X, \beta, \sigma \sim \mathrm{N}(\mu, \sigma^2) \tag{8.1}$$

$$y^* \sim \mathrm{N}(\mu, \sigma^2)$$
$$\mu = \alpha + X\beta$$

We create the linear combination of our features, and then we employ a normal distribution that uses that combination as the mean, which will naturally vary for each sample of data. However, this may not be the best approach in many cases. Instead, we can use some other distribution that potentially fits the data better. But often these other distributions don't have a direct link to our features, and that's where a **link function** comes in.

Think of the link function as a bridge between our features and the distribution we want to use. It lets us use a linear combination of features to predict the mean or other parameters of the distribution. As an example, we can use a log to link the mean to the linear predictor, or conversely, exponentiate the linear predictor to get the mean. In this example, the log is the link function and we use its inverse to map the linear predictor back to the mean.

More generically, we can write this as follows, with some parameter θ, and where g is the link function with g^{-1} is its inverse. The link function and its inverse relates x to θ.

$$g(\theta) = x \tag{8.2}$$

$$\theta = g^{-1}(x)$$

If you know a distribution's 'canonical' link function, which is like the default for a given distribution, that is all the deeper you will probably ever need to go. At the end of the day, these link functions will link your model output to the parameters required for the distribution. The take-away here is that the link function describes how the mean or other parameters of interest are generated from the (linear) combination of features.

> **i** Conditional Reminder
>
> One thing to note, when we switch distributions for GLMs, we're still concerning ourselves with the *conditional* distribution of the target variable given the features. The distribution of the target variable itself is not changing per se, even though its nature, e.g., as a binary variable, is suggesting to us to try something that would allow us to produce a binary outcome from the model. But just like we don't assume the target itself is normal in a linear regression, here we are assuming that the conditional distribution of the target given the features is the distribution we are specifying.

8.3 Logistic Regression

As we've seen, you will often have a binary variable that you might want to use as a target – it could be dead/alive, lose/win, quit/retain, etc. You might be tempted to use a linear regression, but you will quickly find that it's not the best option in that setting. So let's try something else.

8.3.1 The binomial distribution

Logistic regression differs from linear regression mostly because it is used with a binary target instead of a continuous one as with linear regression. As a result, we typically assume that the target follows a **binomial distribution**. Unlike the normal distribution, which is characterized by its mean (μ) and variance (σ^2), the binomial distribution is defined by the parameters: p (also commonly π) and a known value n. Here, p represents the probability of a specific event occurring (like flipping heads, winning a game, or defaulting on a loan), and n is the number of trials or attempts under consideration.

It's important to note that the binomial distribution, which is commonly employed in GLMs for logistic regression, doesn't just describe the probability of a single event that is observed or not. It actually represents the distribution of the number of successful outcomes in n trials, which can be greater than 1. In other words, *it's a count distribution* that tells us how many times we can expect the event to occur in a given number of trials.

Let's see how the binomial distribution looks with 100 trials and probabilities of 'success' at $p =$ .25, .5, and .75:

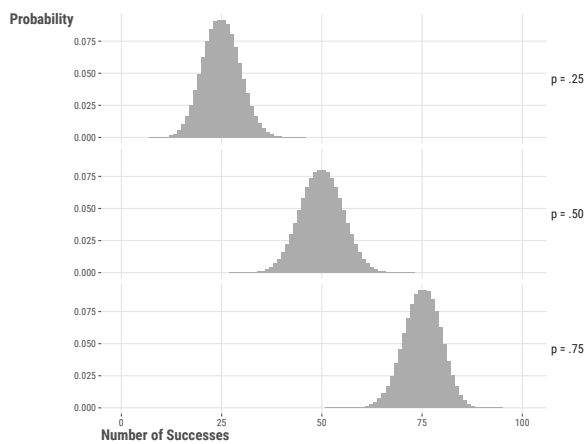


Figure 8.1: Binomial distributions for different probabilities.

If we examine the distribution for a probability of .5, we will see that it is roughly centered over a total success count of 50 out of the 100 trials. This tells us that we have the highest probability of encountering 50 successes if we ran 100 trials. Shifting our attention to a .75 probability of success, we see that our distribution is centered over 75. In practice we probably end up with something around that value, but on average and over repeated runs of 100 trials, the value would be $p \cdot n$. Try it yourself.

R

Note that R switches 'size' and 'n' names relative to numpy. `n` regards the number of values you want returned.

```
set.seed(123)

# produces a count whose mean is n*p
rbinom(n = 6, size = 100, prob = .75)

# produces a binary 0, 1 as seen in logistic regression target (with mean p)
rbinom(n = 6, size = 1, prob = .75)
```

```
[1] 77 72 76 70 68 82
[1] 1 0 1 1 0 1
```

Python

Note that numpy switches 'size' and 'n' names relative to R. Here `n` is the same as depicted in the formulas later.

```
import numpy as np

np.random.seed(123)

# produces a count whose mean is n*p
np.random.binomial(n = 100, p = .75, size = 6)

# produces a binary 0, 1 as seen in logistic regression target (with mean p)
np.random.binomial(n = 1, p = .75, size = 6)
```

```
array([73, 78, 78, 75, 73, 76])
array([0, 1, 1, 1, 1, 1])
```

Since we are dealing with a number of trials, it is worth noting that the binomial distribution is a discrete distribution. If we have any interest in knowing the probability for a number of successes, we can use the following

formula, where n is the number of trials, x is the number of successes, and p is the probability of success:

$$P(x) = \frac{n!}{(n-x)!x!}p^x(1-p)^{n-x} \tag{8.3}$$

Now let's see how the binomial distribution relates to the linear model space:

$$y^* \sim \text{Binomial}(n, p)$$

$$\text{logit}(p) = \alpha + X\beta \tag{8.4}$$

In this case, we are using the **logit** function to map the linear combination of our features to the probability of success. The logit function is defined as:

$$\log\frac{p}{1-p}$$

We are literally just taking the log of the odds of whichever label we're calling 'success' in the binomial sense. For example, if we are predicting the probability of a person subscribing to a membership, we might call 'subscribes' the 'success' label.

Now we can map this back to our model:

$$\log\frac{p}{1-p} = \alpha + X\beta$$

And finally, we can take that logistic function and use the **inverse-logit** function to produce the probabilities:

$$p = \frac{\exp(\alpha + X\beta)}{1 + \exp(\alpha + X\beta)}$$

or equivalently:

$$p = \frac{1}{1 + \exp(-(\alpha + X\beta))}$$

Whenever we get results for a logistic regression model, the default coefficients and predictions are almost always on the log-odds scale. We usually exponentiate the coefficients to get the **odds ratio**. For example, if we have a coefficient of .5, we would say that for every one-unit increase in the feature, the odds of the target being a 'success' increase by a factor of `exp(.5)` $= 1.6$. And we can

convert the predicted log odds to probabilities using the inverse-logit function. We explore this more in the next section.

8.3.2 Probability, odds, and log odds

Probability lies at the heart of a logistic regression model, so let's look more closely at the relationship between the probability and log odds. In our model, the log odds are produced by the linear combination of our features. Let's say we have a model that gives us those values for each observation. We can then convert them from the linear space to the (nonlinear) probability space with our inverse-logit function, which might look something like this.

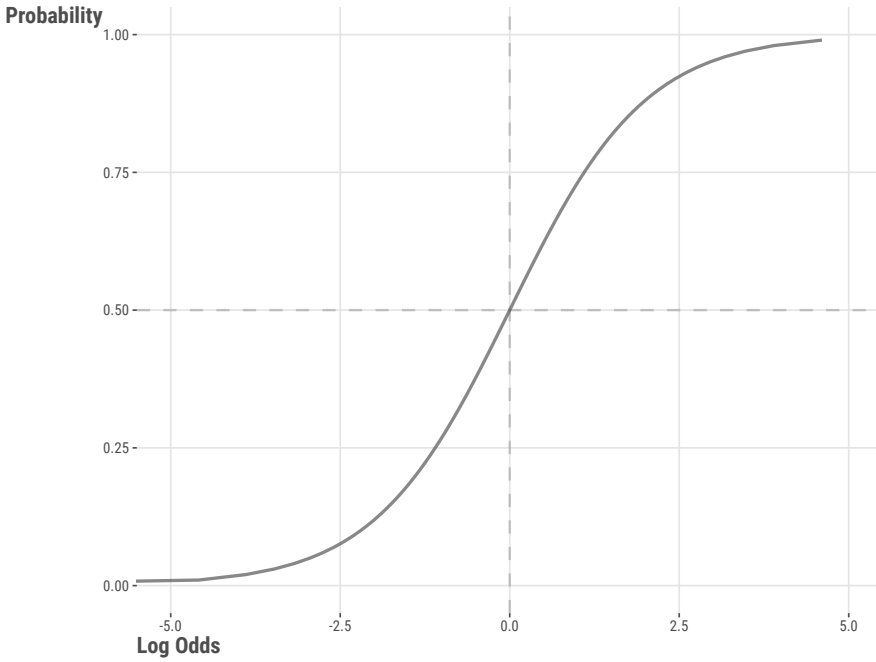


Figure 8.2: Log-odds and probability values.

We can see that the probability of success approaches 0 when the log odds are increasingly negative, and approaches 1 when the log odds are increasingly positive. The shape is something like an S, which also tells us that we are not in linear space when we switch to probabilities.

Log odds have a nice symmetry around 0, where the probability of success is 0.5. Any value above 0 indicates a probability of success greater than 0.5, and any value below 0 indicates a probability of success less than 0.5. However,

don't get too hung up on a .5 probability as being fundamentally important for any given problem.

As mentioned, logistic regression models usually report coefficients on the log-odds scale by default. The coefficients reflect the odds associated with predicted probabilities given the feature at different values one unit apart. Log-odds are not the most intuitive thing to interpret. For additional interpretability, we often convert the coefficients to odds ratios by exponentiating them. In logistic regression models, the odds ratio is the ratio of the odds of the outcome occurring (vs. not occurring) for a one-unit increase in the feature.

The following function will calculate the odds ratio for two probabilities, which we can think of as prediction outcomes for two values of a feature one unit apart.

R

```
calculate_odds_ratio = function(p_1, p_2) {
    odds_1 = p_1 / (1 - p_1)
    odds_2 = p_2 / (1 - p_2)
    odds_ratio = odds_2 / odds_1

    tibble(
        value    = c('1', '2'),
        p        = c(p_1, p_2),
        odds     = c(odds_1, odds_2),
        log_odds = log(odds),
        odds_ratio = c(NA, odds_ratio)
    )
}

result_A = calculate_odds_ratio(.5, .6)
result_B = calculate_odds_ratio(.1, .2)
result_C = calculate_odds_ratio(.9, .8)  # inverse of the .1, .2 example

result_A
```

Python

```
import pandas as pd
import numpy as np

def calculate_odds_ratio(p_1, p_2):
    odds_1 = p_1 / (1 - p_1)
    odds_2 = p_2 / (1 - p_2)
```

```
    odds_ratio = odds_2 / odds_1

    return pd.DataFrame({
        'value': ['1', '2'],
        'p': [p_1, p_2],
        'odds': [odds_1, odds_2],
        'log_odds': [np.log(odds_1), np.log(odds_2)],
        'odds_ratio': [np.nan, odds_ratio]
    })

result_A = calculate_odds_ratio(.5, .6)
result_B = calculate_odds_ratio(.1, .2)
result_C = calculate_odds_ratio(.9, .8)  # inverse of the .1, .2 example

result_A
```

Table 8.1: Odds Ratios for Different Probabilities

	value	p	odds[1]	log_odds	odds_ratio[2]
A	1	0.50	1.00	0.00	NA
	2	0.60	1.50	0.41	1.50
B	1	0.10	0.11	−2.20	NA
	2	0.20	0.25	−1.39	2.25
C	1	0.90	9.00	2.20	NA
	2	0.80	4.00	1.39	0.44

[1]The odds are p / (1 - p).
[2]The odds ratio refers to value 2 versus value 1.

In the table we see that even though each probability difference is the same, the odds ratio is different. Comparing A to B, the difference between a probability of .5 to .6 is not as much of a change on the odds (linear) scale as the difference between .1 to .2. The first setting is a 50% increase in the odds, whereas the second more than doubles the odds. However, the difference between .9 to .8 is the same probability difference as the difference between .1 to .2, as they reflect points that are the same distance from the boundary. The odds ratio for setting C is just the inverse of setting B. The following shows our previous plot with the corresponding settings shaded.

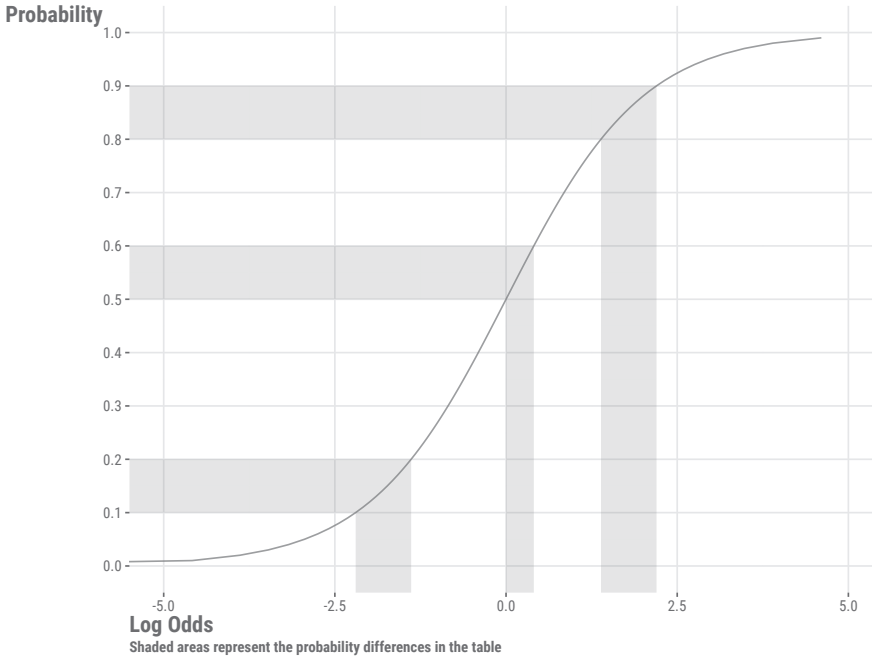


Figure 8.3: Comparison of probability and odds differences.

Odds ratios *might* be more interpretable to some, but since they are ratios of ratios, people have historically had a hard time with those as well. As shown in Table 8.1, knowledge of the baseline rate is *required* for a good understanding of them. Furthermore, doubling the odds is not the same as doubling the probability, so we're left doing some mental calisthenics to interpret them. Odds ratios are often used in academic settings, but elsewhere they are not as common. The take-home message is that we can interpret our result in terms of odds (ratios of probabilities), log odds (linear space), or as probabilities (nonlinear space), but it can take a little more effort than our linear regression setting[1]. Our own preference is to stick with predicted probabilities, but it's good to have familiarity with odds ratios, as well as understand the purely linear aspect of the model.

8.3.3 A logistic regression model

Now let's get our hands dirty and do a classification model using logistic regression. For our model, let's return to the movie review data, but now we'll use the binary `rating_good` ('good' vs. 'bad') as our target. Before we get to modeling, see if you can find out the frequency of 'good' and 'bad' reviews,

[1]For more on interpreting odds ratios, see this article.

and the probability of getting a 'good' review. We examine the relationship of `word_count` and `gender` features with the likelihood of getting a good rating.

R

```
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

model_logistic = glm(
    rating_good ~ word_count + gender,
    data = df_reviews,
    family = binomial
)

summary(model_logistic)
```

```
Call:
glm(formula = rating_good ~ word_count + gender, family = binomial,
    data = df_reviews)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)   1.7124     0.1814    9.44   <2e-16 ***
word_count   -0.1464     0.0155   -9.44   <2e-16 ***
gendermale    0.1189     0.1375    0.86     0.39
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1370.4  on 999  degrees of freedom
Residual deviance: 1257.4  on 997  degrees of freedom
AIC: 1263

Number of Fisher Scoring iterations: 4
```

Python

```
import pandas as pd
import statsmodels.formula.api as smf
import statsmodels.api as sm

df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')
```

```
model_logistic = smf.glm(
    'rating_good ~ word_count + gender',
    data = df_reviews,
    family = sm.families.Binomial()
).fit()

model_logistic.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:            rating_good   No. Observations:                 1000
Model:                            GLM   Df Residuals:                      997
Model Family:                Binomial   Df Model:                            2
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -628.70
Date:                Sat, 10 May 2025   Deviance:                       1257.4
Time:                        11:06:15   Pearson chi2:                 1.02e+03
No. Iterations:                     4   Pseudo R-squ. (CS):             0.1068
Covariance Type:            nonrobust
==============================================================================
                   coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept        1.7124      0.181      9.442      0.000       1.357       2.068
gender[T.male]   0.1189      0.138      0.865      0.387      -0.151       0.388
word_count      -0.1464      0.016     -9.436      0.000      -0.177      -0.116
==============================================================================
"""
```

Now that we have some results, we can see that they aren't too dissimilar from the linear regression output we obtained before. But, let's examine them more closely in the next section.

Binomial Regression

As noted, the binomial distribution is a count distribution. For a binary outcome, we can only have a 0 or 1 outcome for each 'trial', and the 'size' or 'n' for the binomial distribution is 1. In this case, we can also use the Bernoulli distribution (Bern(p)). This does not require the number of trials, since, when the number of trials is 1, the factorial part of Equation 8.3 drops out.

> Many coming from a non-statistical background are not aware that their logistic model can actually handle count and/or proportional outcomes.

8.3.4 Interpretation and visualization

If our modeling goal is not just producing predictions, we need to know what those results mean. The coefficients that we get from our model are in the log-odds scale. Interpreting log odds is difficult, but we can at least get a feeling for them directionally. A log odds of 0 (odds ratio of 1) would indicate no relationship between the feature and target. A positive log odds would indicate that an increase in the feature will increase the log odds of moving from 'bad' to 'good', whereas a negative log odds would indicate that an increase in the feature will decrease the log odds of moving from 'bad' to 'good'. On the log-odds scale, the coefficients are symmetric as well, such that, e.g., a +1 coefficient denotes a similar increase in the log odds, as a -1 coefficient denotes a decrease. As we demonstrated, we can exponentiate them to get the odds ratio for additional interpretability.

Table 8.2: Raw Coefficients and Odds Ratios for a Logistic Regression

Parameter	Coefficient	Exp. Coef (OR)
(Intercept)	1.71	5.54
word_count	−0.15	0.86
gendermale	0.12	1.13

The intercept provides a baseline odds of a 'good' review when word count is 0 and gender is 'female'. From there, we see that we've got a negative raw coefficient and odds ratio of 0.86 for the word count variable. We have a positive raw coefficient and 1.13 odds ratio for the male variable. This means that for every one-unit increase in word count, the odds of a 'good' review decreases by about 14%. Males are associated with an odds of a 'good' review that is 13% higher than females.

We feel it is much more intuitive to interpret things on the probability scale, so we'll get predicted probabilities for different values of the features. The way we do this is through the (inverse) link function, which will convert our log odds of the linear predictor to probabilities. We can then look at specific predictions, calculate marginal effects, or plot these probabilities to see how they change with the features. For the word count feature in the following visualization, we hold gender at the reference group ('female'), and for the gender feature, we hold word count at its mean. In addition we convert the probability to the percentage chance of a 'good' review.

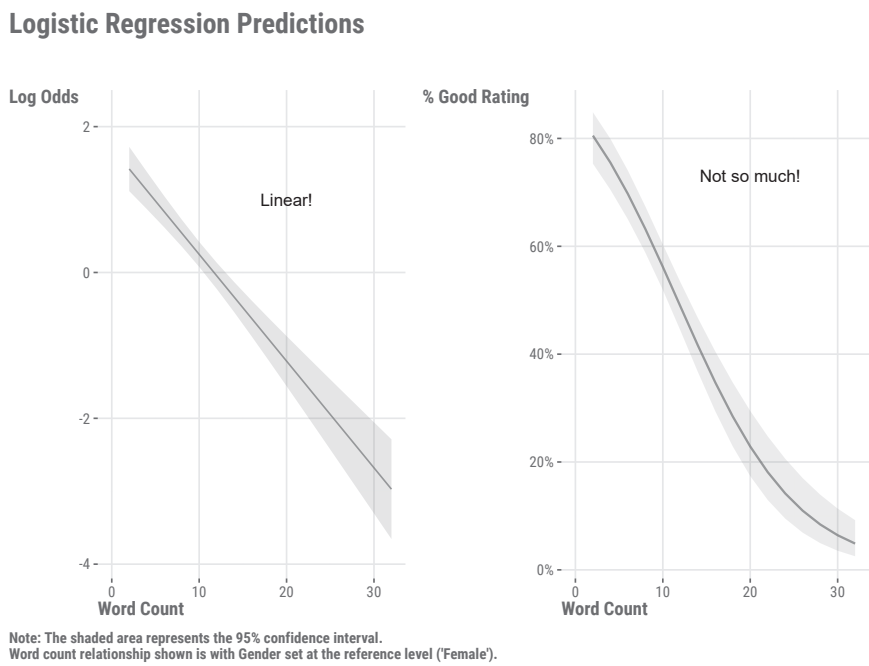


Figure 8.4: Model predictions for word count feature.

In Figure 8.4, we can see a clear negative relationship between the number of words in a review and the probability of being considered a 'good' movie. As we get over 20 words, the predicted probability of being a 'good' movie is less than .2. We also calculated the average marginal effect (Section 5.5.2), or average slope, for word count. It suggests a -0.03 decrease in the probability of a 'good' rating for each additional word in the review (on average).

We also see an increase in the chance for a good rating with males vs. females, but our model results suggest this is not a statistically significant difference.

In the end, whether you think these differences are practically significant is up to you. And you'll still need to do the standard model exploration to further understand the model (Chapter 4 has lots of detail on this). But this is a good start.

> **i** A Note on R^2 for Logistic Regression
>
> Logistic regression does not have an R^2 value in the way that a linear regression model does. Instead, there are pseudo-R^2 values, but they are not the same as the R^2 value that you are used to seeing (UCLA Advanced Research Computing (2023)).

8.4 Poisson Regression

Poisson regression also belongs to the class of generalized linear models, and it is used specifically when you have a count variable as your target. After logistic regression for binary outcomes, Poisson regression is probably the next most common type of generalized linear model you will encounter. Unlike continuous targets, a count starts at 0 and can only be a whole number. Often it is naturally skewed as well, so we'd like a model that is well suited to this situation. Unlike the binomial, there is no concept of number of trials, just the count of events.

8.4.1 The Poisson distribution

The Poisson distribution is very similar to the binomial distribution, because the binomial is also a count distribution, and in fact generalizes the Poisson[2]. The Poisson has a single parameter noted as λ, which makes it the simplest model setting we've seen so far[3]. Conceptually, this rate parameter is going to estimate the expected number of events during a time interval. This can be accidents in a year, pieces produced in a day, or hits during the course of a baseball season.

Let's see what the particular distribution might look like for different rates. We can see that for low count values, the distribution is skewed to the right, but note how the distribution becomes more symmetric and bell-shaped as the rate increases[4]. You might also be able to tell that the variance increases along with the mean, and in fact, the variance is equal to the mean for the Poisson distribution.

[2]If your binomial setting has a very large number of trials relative to the number of successes, which amounts to very small proportions p, you would find that the binomial distribution would converge to the Poisson distribution.

[3]Neither the binomial nor the Poisson has a variance parameter to estimate, as the variance is determined by the mean. This is in contrast to a normal distribution model, where the variance is an estimated parameter. For the Poisson, the variance is equal to the mean, and for the binomial, the variance is equal to $n * p * (1 - p)$. The Poisson assumption of equal variance rarely holds up in practice, so people often use the **negative binomial** distribution instead.

[4]From a modeling perspective, for large mean counts you can just go back to using the normal distribution if you prefer, without losing much predictively and possibly gaining interpretability.

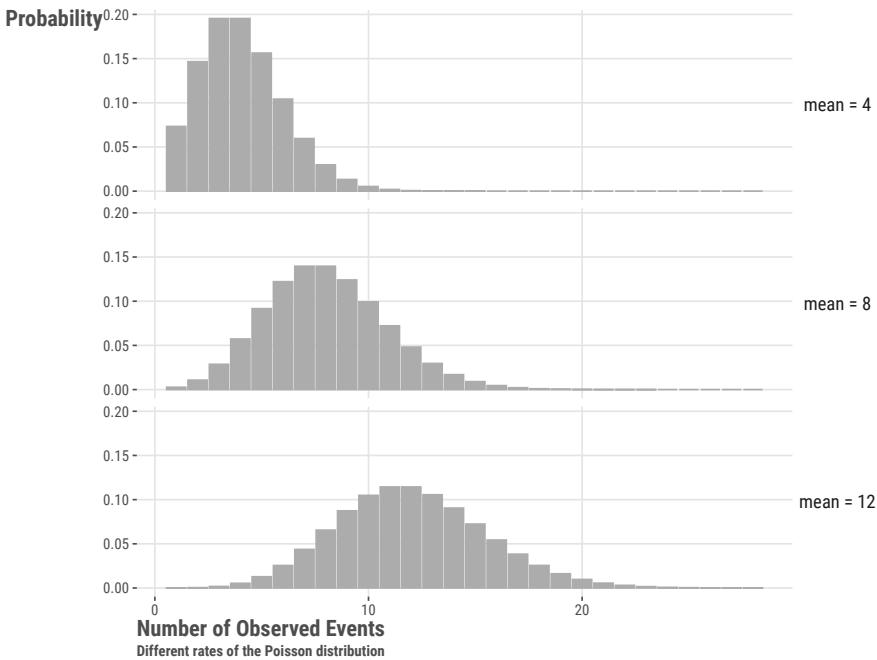


Figure 8.5: Poisson distributions for different rates.

> **More Poisson**
>
> A cool thing about these distributions is that they can deal with different *exposure* rates. They can also be used to model inter-arrival times and time-until-events.

Let's make a new variable that will count the number of times a person uses a personal pronoun word in their review. We'll use it as our target variable and see how it relates to the number of words and gender in a review as we did before.

R

```
df_reviews$poss_pronoun = stringr::str_count(
    df_reviews$review_text,
    '\\bI\\b|\\bme\\b|\\b[Mm]y\\b|\\bmine\\b|\\bmyself\\b'
)

hist(df_reviews$poss_pronoun)
```

Python

```
df_reviews['poss_pronoun'] = (
    df_reviews['review_text']
    .str.count('\\bI\\b|\\bme\\b|\\b[Mm]y\\b|\\bmine\\b|\\bmyself\\b')
)

df_reviews['poss_pronoun'].hist()
```

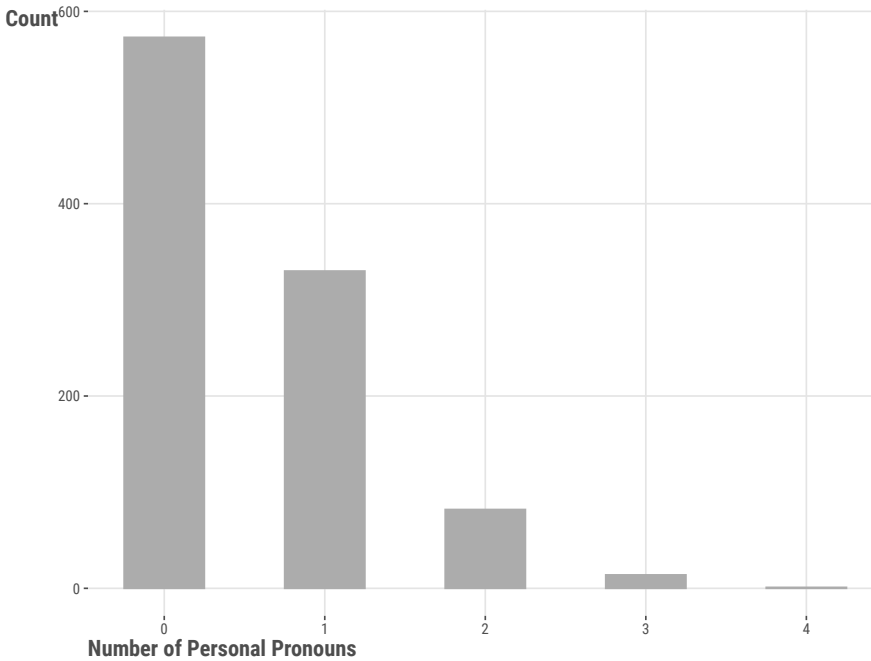


Figure 8.6: Distribution of personal pronouns seen across reviews.

8.4.2 A Poisson regression model

Recall that GLM specific distributions have a default link function. The Poisson distribution uses a log link function:

$$y^* \sim \text{Poisson}(\lambda) \tag{8.5}$$

$$\log(\lambda) = \alpha + X\beta$$

Using the log link keeps the outcome non-negative when we use the inverse of it. For model fitting with standard functions, all we have to do is switch the family from 'binomial' to 'poisson'. As the default link is the 'log', we don't have to specify it explicitly[5].

So in this model we'll predict the number of personal pronouns used in a review, and we'll use word count and gender as our features like we did with the logistic model.

R

```
model_poisson = glm(
    poss_pronoun ~ word_count + gender,
    data = df_reviews,
    family = poisson
)

summary(model_poisson)
```

```
Call:
glm(formula = poss_pronoun ~ word_count + gender, family = poisson,
    data = df_reviews)

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.88767    0.10851  -17.40   <2e-16 ***
word_count   0.10365    0.00646   16.05   <2e-16 ***
gendermale   0.07980    0.08806    0.91     0.36
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

    Null deviance: 996.21  on 999  degrees of freedom
Residual deviance: 775.38  on 997  degrees of freedom
AIC: 1701

Number of Fisher Scoring iterations: 5
```

```
# exponentiate the coefficients to get the rate ratio
# exp(model_poisson$coefficients)
```

[5]It is not uncommon in many disciplines to use different link functions for logistic models, but the log link is always used for Poisson models.

Python

```
model_poisson = smf.glm(
    formula = 'poss_pronoun ~ word_count + gender',
    data = df_reviews,
    family = sm.families.Poisson()
).fit()

model_poisson.summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:           poss_pronoun   No. Observations:                 1000
Model:                            GLM   Df Residuals:                      997
Model Family:                 Poisson   Df Model:                            2
Link Function:                    Log   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -847.43
Date:                Sat, 10 May 2025   Deviance:                       775.38
Time:                        11:06:17   Pearson chi2:                     717.
No. Iterations:                     5   Pseudo R-squ. (CS):             0.1981
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      -1.8877      0.109    -17.395      0.000      -2.100      -1.675
gender[T.male]  0.0798      0.088      0.906      0.365      -0.093       0.252
word_count      0.1036      0.006     16.053      0.000       0.091       0.116
==============================================================================
"""
```

```
# exponentiate the coefficients to get the rate ratio
# np.exp(model_poisson.params)
```

8.4.3 Interpretation and visualization

Now let's check out the results more deeply. Like with logistic, we can exponentiate the coefficients to get what's now referred to as the rate ratio. This is the ratio of the rate of the outcome occurring for a one-unit increase in the feature.

Table 8.3: Rate Ratios for a Poisson Regression

Parameter	Coefficient	Exp. Coef.
(Intercept)	−1.89	0.15
word_count	0.10	1.11
gendermale	0.08	1.08

For this model, an increase in one review word leads to an expected log count increase of ~0.1. We can exponentiate this to get 1.11, and this tells us that every added word in a review gets us a 11% increase in the number of possessive pronouns. This is probably not a surprising result – wordier stuff has more types of words! In addition, the average marginal effect (Section 5.5.2), or average slope, for word count suggested a 0.06 increase in the number of possessive pronouns per word on average. A similar, though slightly smaller, increase is seen for males relative to females, but, as with our previous model, this is not statistically significant.

But as usual, the visualization tells the better story. Here is the relationship for word count. Notice that the predictions are not discrete like the raw count, but continuous. This is because predictions here are the same as with our other models, and reflect the expected, or average, count that we'd predict with this data.

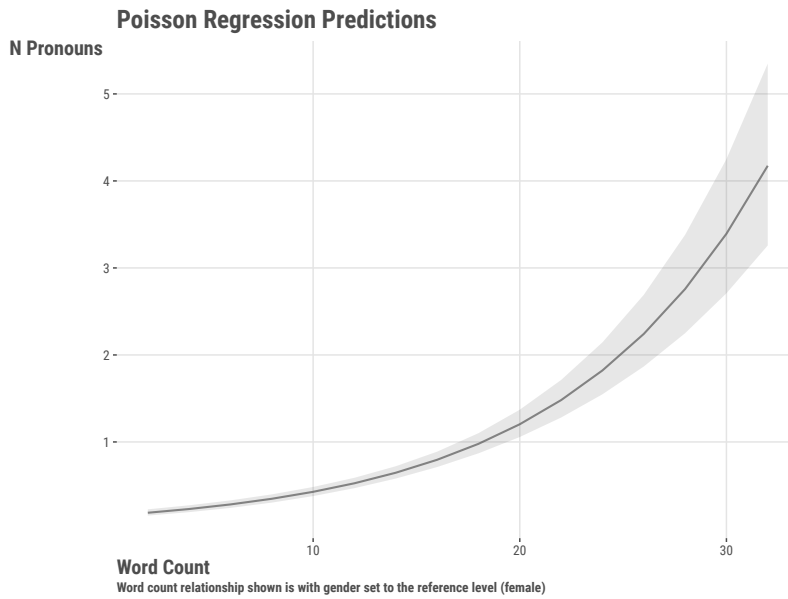


Figure 8.7: Poisson model predictions for word count feature.

With everything coupled together, we have an interpretable coefficient for `word_count`, a clear plot, and adequate model fit. Therefore, we might conclude that there is a positive relationship between the number of words in a review and the number of times a person uses a personal possessive.

> **Nonlinear Linear Models?**
>
> You'll note again that our effects for word count in the logistic (Figure 8.4) and Poisson (Figure 8.7) models were not exactly the straightest of lines. Once we're on the probability and count scales, we're not going to see the same linear relationships that we might expect from a linear regression model due to the transformation. If we plot the effect on the log-odds or log-count scale, we're back to straight lines, as demonstrated with the logistic model. This is a first taste in how the linear model can be used to get at nonlinear relationships depending on the scale we focus on. More explicit nonlinear relationships are the focus of Chapter 9.

8.5 DIY

If we really want to demystify the modeling process for GLMs, let's create our own function to estimate the coefficients. We can use maximum likelihood estimation to estimate the parameters of our model, which is the approach used by standard package functions. Feel free to skip this part if you only wanted the basics, but for even more information on maximum likelihood estimation, see Section 6.7 where we take a deeper dive into the topic and with a similar function. The following code is a simple and conceptual version of what goes on behind the scenes with 'glm' type functions.

R

```
glm_simple = function(par, X, y, family = 'binomial') {
    # add a column for the intercept
    X = cbind(1, X)

    # Calculate the linear predictor
    mu = X %*% par # %*% is matrix multiplication

    # get the likelihood for the binomial or poisson distribution
    if (family == 'binomial') {
        # Convert to a probability ('logit' link/inverse)
        p = 1 / (1 + exp(-mu))
```

```
        L = dbinom(y, size = 1, prob = p, log = TRUE)
    }
    else if (family == 'poisson') {
        # Convert to a count ('log' link/inverse)
        p = exp(mu)
        L = dpois(y, lambda = p, log = TRUE)
    }

    # return the negative sum of the log-likelihood (for minimization)
    value = -sum(L)

    return(value)
}
```

Python

```
from scipy.stats import poisson, binom

def glm_simple(par, X, y, family = 'binomial'):
    # add a column for the intercept
    X = np.column_stack((np.ones(X.shape[0]), X))

    # Calculate the linear predictor
    mu = X @ par  # @ is matrix multiplication

    # get the likelihood for the binomial or poisson distribution
    if family == 'binomial':
        p = 1 / (1 + np.exp(-mu))
        L = binom.logpmf(y, 1, p)
    elif family == 'poisson':
        lambda_ = np.exp(mu)
        L = poisson.logpmf(y, lambda_)

    # return the negative sum of the log-likelihood (for minimization)
    value = -np.sum(L)

    return value
```

Now that we have our objective function, we can fit our models, starting with the logistic model. We will use the `optim` function in R and the `minimize` function in Python. We'll convert our data to matrix format for this purpose as well.

R

```
X = df_reviews |>
    select(word_count, male = gender) |>
    mutate(male = ifelse(male == 'male', 1, 0)) |>
    as.matrix()

y = df_reviews$rating_good

init = rep(0, ncol(X) + 1)

names(init) = c('intercept', 'b1', 'b2')

our_logistic = optim(
    par = init,
    fn = glm_simple,
    X = X,
    y = y,
    control = list(reltol = 1e-8)
)

our_logistic$par
```

Python

```
import numpy as np
from scipy.optimize import minimize

# for the 'by-hand' option later
X = (
    df_reviews[['word_count', 'gender']]
    .rename(columns = {'gender': 'male'})
    .assign(male = np.where(df_reviews[['gender']] == 'male', 1, 0))
)

y = df_reviews['rating_good']

init = np.zeros(X.shape[1] + 1)

our_logistic = minimize(
    fun = glm_simple,
    x0 = init,
    args = (X, y),
    method = 'BFGS'
```

```
)

our_logistic['x']
```

And here is our comparison table of the raw coefficients. It looks like our little function worked well!

Table 8.4: Comparison of Coefficients

Parameter	Ours	Standard
Intercept	1.7122	1.7124
Word Count	−0.1464	−0.1464
Male	0.1189	0.1189

Similarly, we can also use our function to estimate the coefficients for the Poisson model. Just like the GLM function we might normally use, we can change the family option to specify the distribution we want to use.

R

```
our_poisson = optim(
    par = c(0, 0, 0),
    fn = glm_simple,
    X = X,
    y = df_reviews$poss_pronoun,
    family = 'poisson'
)

our_poisson$par
```

Python

```
our_poisson = minimize(
    fun = glm_simple,
    x0 = init,
    args = (
        X,
        df_reviews['poss_pronoun'],
        'poisson'
    )
)

our_poisson['x']
```

And once again we're able to get the same results (raw coefficients shown).

Table 8.5: Comparison of Coefficients

Parameter	Ours	Standard
Intercept	−1.8876	−1.8877
Word Count	0.1036	0.1036
Male	0.0800	0.0798

This goes to show that just a little knowledge of the underlying mechanics can go a long way in understanding how many models work.

8.6 Wrapping Up

So at this point you have standard linear regression with the normal distribution for continuous targets, logistic regression for binary/proportional ones via the binomial distribution, and Poisson regression for counts. These models combine to provide much of what you need for starting out in the linear modeling world, and all serve well as baseline models for comparison when using more complex methods (Section 11.4). However, what we've seen is just a tiny slice of the potential universe of distributions that you could use. Here is a brief list of some that are still in the GLM family proper, and others that technically aren't GLMs but can be similarly useful[6]:

Other Core GLM (available in standard functions):

- **Gamma**: For continuous, positive targets that are skewed.
- **Inverse Gaussian**: For continuous, positive targets that are skewed and have a long tail.

Others (some fairly common):

- **Beta**: For continuous targets that are bounded between 0 and 1.
- **Log-Normal**: For continuous targets that are skewed. Essentially what you get with linear regression and logging the target[7].
- **Tweedie**: Generalizes several core GLM family distributions.

In the ballpark:

- **Negative Binomial**: For count targets that are 'overdispersed'.
- **Multinomial**: Typically used for categorical targets with more than two categories, but like the binomial, it is actually a more general (multivariate) count distribution.

[6] There is no strict agreement about what qualifies for being in the GLM family.
[7] But there is a variance issue to consider.

- **Student t**: For continuous targets that are conditionally distributed similar to normal, but with heavier tails.
- **Categorical/Ordinal**: For categorical targets with more than two categories, or ordered categories. In the former case, it's a different distribution than the multinomial but is applied to the same setting.
- **Quasi ***: For example Quasi-Poisson. These 'quasi-likelihoods' served a need at one point that is best served by other approaches these days.

You'll typically need separate packages to fit some of these, but most often the tools keep to a similar functional approach. The main thing is to know that certain distributions might help your model fit the data a bit better than others, and that you can use both the same basic framework and mindset to conduct the analysis, and maybe get a little closer to the answer you seek about your data!

8.6.1 The common thread

GLMs extend your standard linear model as a powerful tool for modeling a wide range of data types. They are a great way to get started with more complex models, and even allow us to linear models in a not so linear way. It's best to think of GLMs more broadly than the strict statistical definition, and consider many models like ordinal regression, ranking models, survival analysis, and more as part of the same extension.

8.6.2 Choose your own adventure

At this point you have a pretty good sense of what linear models have to offer, but there's even more! You can start to look at more complex models that build on these ideas, like mixed models, generalized additive models and more in Chapter 9. You can also feel confident heading into the world of machine learning (Chapter 10), where you'll find additional ways to think about your modeling approach.

8.6.3 Additional resources

If you are itching for a textbook, there isn't any shortage of those that focus on GLMs out there, and you can essentially take your pick. Most purely statistical treatments are going to be a bit dated at this point, but still accurate and maybe worth your time.

- *Generalized Linear Models* (McCullagh (2019)) is a classic text on the subject, but it is a bit dense and not for the faint of heart, or even Nelder and Wedderburn (1972), which is a very early treatment.

For more accessible fare that doesn't lack on core details either:

- *An Introduction to Generalized Linear Models* is generally well regarded (Dobson and Barnett (2018)).
- *Generalized Linear Models* is another accessible text (Hardin and Hilbe (2018)).
- R*Beyond Multiple Linear Regression*, is available for free (Roback and Legler (2021)).
- *Applied Regression Analysis and Generalized Linear Models*, Fox (2015).
- *Generalized Linear Models with Examples in R*, Dunn and Smyth (2018).
- *Extending the Linear Model with R* is a great resource for moving beyond the basics with R (J. J. Faraway (2016)).

8.7 Guided Exploration

You have two options here, both using the fish data (Section C.4).

- Conduct a Poisson regression to predict the number of fish caught based on the other features like how many people were on the trip, how many children, whether live bait was used (or any others). Interpret your model results in terms of the coefficients/rate ratios. Inspect your model more generally to see how well it fits the data, do you spot any issues?
- Using the same data, binarize the count target variable for whether any fish were caught, and proceed similarly as you would have with the Poisson regression.

R

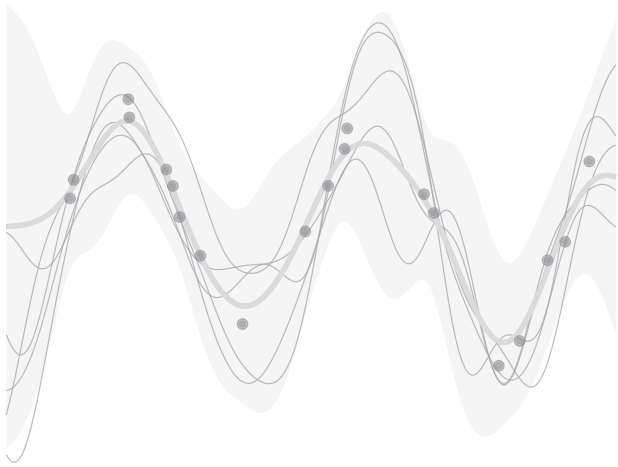
```
# Load the data
df_fish = read_csv('https://tinyurl.com/fishcountdata')
df_fish$any_catch = ifelse(df_fish$count > 0, 1, 0)

model = glm(count ~ ???)
model = glm(any_catch ~ ???)
summary(model)
exp(coef(model))  # IRR/OR
```

Python

```
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf

# Load the data
df_fish = pd.read_csv('https://tinyurl.com/fishcountdata')

df_fish['any_catch'] = (df_fish['count'] > 0).astype(int)

model = smf.glm(
    formula = 'count ~ ???',
    data = df_fish,
    family = ???,
).fit()

model.summary()

np.exp(model.params)  # IRR/OR
```

9

Extending the Linear Model



With just linear and generalized linear models, we have a very solid foundation for modeling, and we've seen how there is a notable amount we can do with a conceptually simple approach. We've also seen how we can extend the linear model to handle different types of target distributions to help us understand and make some inferences about the relationships between our features and target.

In this chapter, we will extend our linear models with additional common and valuable modeling tools. These methods provide good examples of how we can think about our data and modeling approach in different ways, and they can serve as a foundation for exploring more advanced techniques in the future. A thread that binds these techniques together is the ability to use a linear model to investigate explicitly nonlinear relationships!

9.1 Key Ideas

- The standard and generalized linear models are great and powerful starting points for modeling, but there's even more we can do!
- Linear models can be used to model nonlinear feature-target relationships!
- While these seem like different approaches, we can still use our linear model concepts and approach at the core, take similar estimation steps, and even have similar interpretation. However, we'll have even more results to explore and interpret.

9.1.1 Why this matters

The linear model is a great starting point for modeling. It is a simple approach that can be used to model a wide variety of relationships between features and targets, and it's also a great way to get a feel for how to think about modeling. But linear and generalized models are just the beginning, and the models depicted here are common extensions used in a variety of disciplines and industries. More generally, the following techniques allow for nonlinear relationships while still employing a linear model approach. This is a very powerful tool to have in your toolkit, and it's a great way to start thinking about how to model more complex relationships in your data.

9.1.2 Helpful context

While these models are extensions of the linear model, they are not significantly more complicated in terms of how they are implemented or how they are interpreted. However, like anything new, it can take a bit more effort to understand. You likely want to be comfortable with standard linear models at least before you start to explore these extensions.

9.2 Interactions

Things can be quite complex in a typical model with multiple features, but just adding features may not be enough to capture the complexity of the relationships between features and target. Sometimes, we need to consider how features interact with each other to better understand how they correlate with the target. A common way to add complexity in linear models is through **interactions**. This is where we allow the effect of a feature to vary depending on the values of another feature, or even itself!

As a conceptual example, we can think about a movie's rating being different for movies from different genres. For example, maybe by default, ratings are higher for kids' movies, and lower for horror movies. But, genre and season might work together in some way to affect rating, e.g., action movies get higher ratings in the summer. Or maybe having kids in the home might also interact with genre ratings by naturally resulting in higher ratings for kids' movies. As a different example, we might also consider that the length of a movie might positively relate to rating, but plateau or even have a negative effect on rating after a certain point. In other words, it would have a **curvilinear** effect where really long movies aren't as highly rated as those of shorter length.

All of these are types of interactions we can explore. Interactions allow us to incorporate nonlinear relationships into the model, and so greatly extend the linear model's capabilities. We basically get to use a linear model in a nonlinear way!

With that in mind, let's explore how we can add interactions to our models. Going with one of our examples, let's see how having kids impacts the relationship between genre and rating. We'll start with a standard linear model, and then add an interaction term. Using a formula approach makes it very straightforward to add an interaction term. We just need to add a `:` between the two features we want to interact, or a `*` to denote both main effects and the interaction. As elsewhere, we present simplified results in the next table.

R

```
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

model_baseline = lm(rating ~ children_in_home + genre, data = df_reviews)
model_interaction = lm(rating ~ children_in_home * genre, data = df_reviews)

summary(model_interaction)
```

Python

```
import pandas as pd
import statsmodels.formula.api as smf

df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

model_baseline = smf.ols(
    formula = 'rating ~ children_in_home + genre',
    data = df_reviews
).fit()
```

```
model_interaction = smf.ols(
    formula = 'rating ~ children_in_home * genre',
    data = df_reviews
).fit()

model_interaction.summary()
```

Here is a quick look at the model output for the interaction vs. no interaction interaction model. Starting with the baseline model, the coefficients look like what we've seen before, but we have several coefficients for genre. The reason is that genre is composed of several categories and converted to a set of dummy variables (refer to Section 3.5.2 and Section 14.2.2). In the baseline model, the intercept tells us what the mean is for the reference group, in this case Action/Adventure, and the genre coefficients tell us the difference between the mean for that genre and the reference genre. For example, the mean rating for Action/Adventure is 2.76, and the difference between that genre rating for the drama genre is 0.55. Adding the two gives us the mean for drama movies 2.76 + 0.55 = 3.32. We also have the coefficient for the number of children in the home, and this does not vary by genre in the baseline model.

Table 9.1: Model Coefficients with Interaction

feature	coef_base	coef_inter
(Intercept)	2.764	2.764
children_in_home	0.142	0.142
genreComedy	0.635	0.637
genreDrama	0.554	0.535
genreHorror	0.129	0.194
genreKids	−0.199	−0.276
genreOther	0.029	0.084
genreRomance	0.227	0.298
genreSci-Fi	−0.123	−0.109
children_in_home:genreComedy		−0.006
children_in_home:genreDrama		0.053
children_in_home:genreHorror		−0.127
children_in_home:genreKids		0.231
children_in_home:genreOther		−0.106
children_in_home:genreRomance		−0.124
children_in_home:genreSci-Fi		−0.029

But in our other model, we have an interaction between two features: 'children in the home' and 'genre'. So let's start with the coefficient for children. It is 0.14, which means that for every additional child, the rating for any movie increases by that amount. But because of the interaction, we now interpret that as the *effect of children when genre is the reference group Action/Adventure.*

Now let's look at the interaction effect for children and the kids' genre. It is 0.23, which means that for the kids' genre, the *effect of having children in the home increases* by that amount. So our actual effect for an additional child for the kids' genre is $0.14 + 0.23 = 0.37$ increase in the expected review rating. In other words, the effect of children in the home is stronger for kids' movies than for other genres.

We can also interpret this interaction in the reverse fashion. It is also correct to say that the difference in rating between the kids genre and the reference group Action/Adventure when there are no kids in the home is -0.28. This means kids' movies are generally rated worse if there are no kids in the home. But, with an increase in children, the difference in rating between the kids' genre and Action/Adventure increases by 0.23. In other words, it is a **difference in differences**[1].

When we talk about differences in coefficients across values of features, it can get a little bit hard to follow. To combat this, we believe that in every case that you employ an interaction, you should look at the interaction visually for interpretation. Here is a plot of the predictions from the interaction model. We can see that the effect of children in the home is stronger for kids' movies than for other genres, which makes a whole lot of sense! In other genres, the effect of having children seems to produce little difference, and in others it still has a positive effect, but not as strong as for kids' movies.

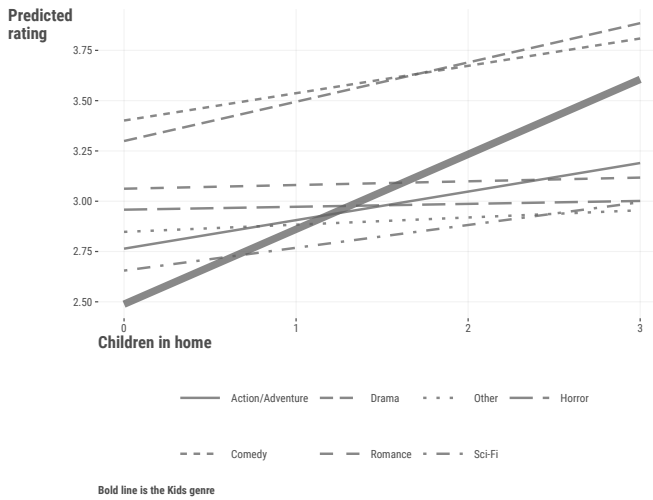


Figure 9.1: Interaction plot.

[1]Some models that employ an interaction that investigates categorical group differences like this actually call their model a difference-in-difference model.

Table 9.2: Average Marginal Effects of Children in the Home

term	estimate	std.error	statistic	p.value	conf.low	conf.high
children_in_home	0.152	0.03	5.68	0.00	0.10	0.20

So we can see that interactions can allow a linear effect to vary depending on the values of another feature. But the real take-home message from this is that *the general effect is actually not just a straight line*! The linear effect *changes* depending on the setting. Furthermore, the effect for children when the interaction is present only applies when 'genre' is at its default group, or when other features are at their default or zero.

So, when we have interactions, we can't talk about a feature's relationship with a target without considering the other features it interacts with. Some might see this as a downside, but it's actually how most feature relationships work, namely, that they don't exist in isolation. Interactions let us model these complex relationships, and they're used a lot in real-world situations.

9.2.1 Summarizing interactions

So what is *the* effect of children in the home? Or a particular genre, for that matter? This is a problematic question, because the effect of one feature depends on the setting of the other feature. We can summarize interactions, and we show two ways in which to do so. But we need to be very careful about summarizing a single feature's effects when we know it interacts with another feature.

9.2.2 Average effects

One thing we can do is get an average effect for a feature. In other words, we can say what the effect of a feature is *on average* across the settings of the other features. This is called the **average marginal effect**, something we've talked about in Section 5.5.2[2]. Here is the average effect of children in the home across all genres.

So-called **marginal effects**, and related approaches such as SHAP values (see Section 5.7), attempt to boil down the effect of a feature to a single number. Here we see the average coefficient for children in the home is 0.15, with a range from 0.1 to 0.2.

But this is difficult even in the simpler GLM settings, and can be downright misleading in interaction models. We saw from Table 9.1 that this average is slightly larger than what we would estimate in the baseline model, and we saw in Figure 9.1 it's actually near zero (flat) for some genres in the interaction

[2]These results are provided by the marginaleffects R package, which is a great tool.

model. So what is the average effect really telling us? Consider a more serious case of drug effects across demographic groups, where the effect of the drug is much stronger for some groups than others. Would you want your doctor to prescribe you a drug based on the average effect across all groups, or the specific group to which you belong?

9.2.3 ANOVA

A common method for summarizing categorical effects in linear models is through **analysis of variance** (ANOVA), something we touched on earlier in Section 3.5.2. ANOVA breaks down the variance in a target attributable to different features or their related effects such as interactions. It's a bit beyond the scope here to get into all the details, but we demonstrate it here, as it's also used to summarize interactions. It also summarizes the random effects and spline terms we'll see in the coming sections on mixed and generalized additive models.

R

```
anova(model_interaction)
```

Python

```
sm.stats.anova_lm(model_interaction)
```

Table 9.3: ANOVA Table for an Interaction Model

feature	df	sum_sq	mean_sq	f	p
children_in_home	1.00	6.45	6.45	21.25	0.00
genre	7.00	86.17	12.31	40.55	0.00
children_in_home:genre	7.00	3.75	0.54	1.76	0.09
Residuals	984.00	298.69	0.30		

In this case, it doesn't appear that the general interaction effect is statistically significant if we use the typical .05 cut-off. We know the effect of children in the home varies across genres, but this result suggests maybe it's not as much as we might think. However, we also saw that the estimate for the children effect more than doubled for the kids genre, so maybe we don't want to ignore it. That's for you to decide.

The ANOVA approach can be generalized to provide a statistical test to compare models. For example, we can compare the baseline model to the interaction model to see if the interaction model is a better fit. However, it's entirely consistent with just looking at the interaction's statistical result in the

ANOVA for the interaction model alone, so it doesn't provide any additional information in this case. Note that the only models that can be compared with ANOVA must be *nested*, i.e., one model is an explicit subset of the other.

> **ANOVA Is Not a Model**
>
> It's worth noting that ANOVA is often confused with being a model itself. When people use it this way, it is just a linear regression with only categorical features, something that is usually only seen within strict experimental designs. It's pretty difficult to think of a linear regression setting where no continuous features would be of theoretical interest, but back when people were doing this stuff by hand, they just categorized everything to enable doing an ANOVA, which was tedious arithmetic but more manageable. It's a bit of a historical artifact, but might be useful for exploratory purposes of feature relationships. For model comparison however, other approaches are more general than ANOVA and preferred. An example is using AIC, or the other methods employed that we discuss in the machine learning chapters, like cross-validation error.

9.2.4 Interactions in practice

When dealing with interactions in a model, it's best to consider how the feature-target relationship changes based on the values of other features it interacts with. Visualizing these effects is important in helping us understand how the relationships change. It's also helpful to consider what the predicted outcome is at important feature values, and how this changes with different feature values. This is the approach we've used with interactions, and it's a good strategy overall.

9.3 Mixed Models

9.3.1 Knowing your data

As much fun as modeling is, knowing your data is far more important. You can throw any model you want at your data, from simple to fancy, but you can count on disappointment if you don't fundamentally know the structure that lies within your data. Let's take a look at the following visualizations. In Figure 9.2, we see a positive relationship between the length of the movie and ratings.

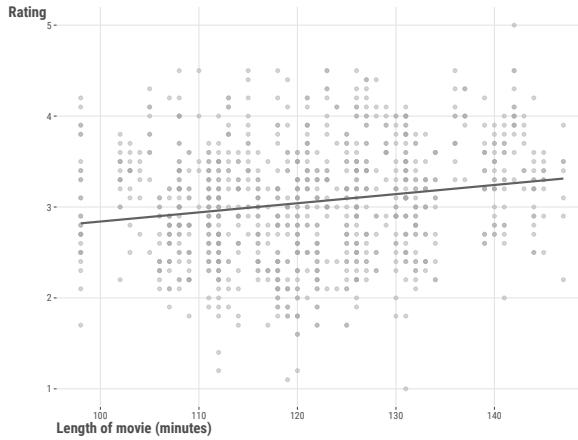


Figure 9.2: Linear relationship between length of movie and rating.

We could probably just stop there, but given what we just saw with interaction, we might think to ourselves to be ignoring something substantial within our data: genre. We might want to ask a question, "Does this relationship work the same way across the different genres?"

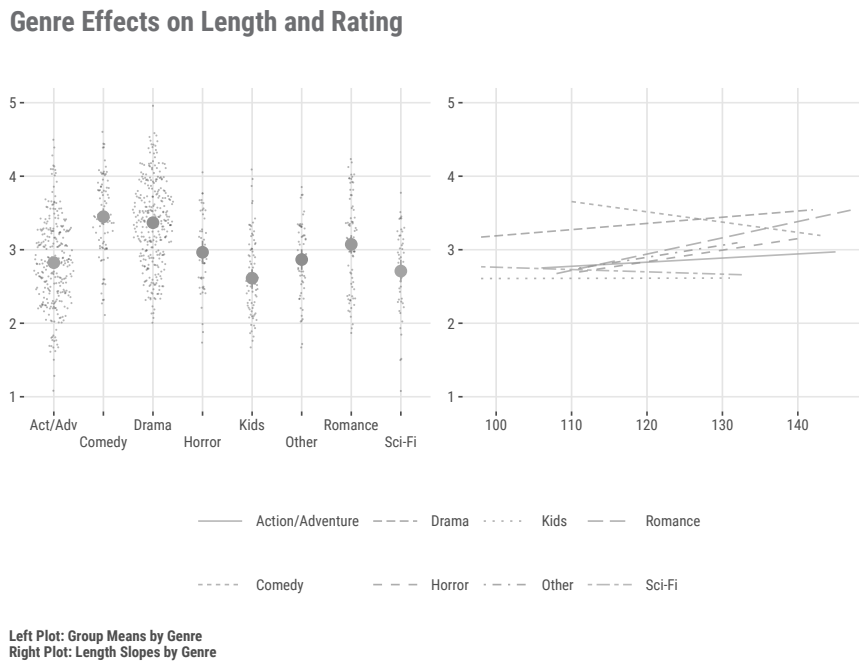


Figure 9.3: Genre effects on length and rating.

A very quick examination of Figure 9.3 might suggest that the rating varies by genre, and that the relationship between length and rating varies significantly over the different genres. The group means in the right panel show variability across genres. In addition, in the left panel, some genres show a strong positive relationship, some show less of a positive relationship, a couple even show a negative relationship, and one even looks flat. We can also see that they would have different intercepts. This is a very important thing to know about your data! If we had just run a model with length as a feature and nothing else, we would have missed this important information.

Now consider something a bit more complicated. Here is a plot of the relationship between the length of a movie and the rating, but across release year. Again we might think there is notable variability in the effect across years, as some slopes are positive, some very strongly positive, and some are even negative. How can we account for this when there are so many group effects to consider?

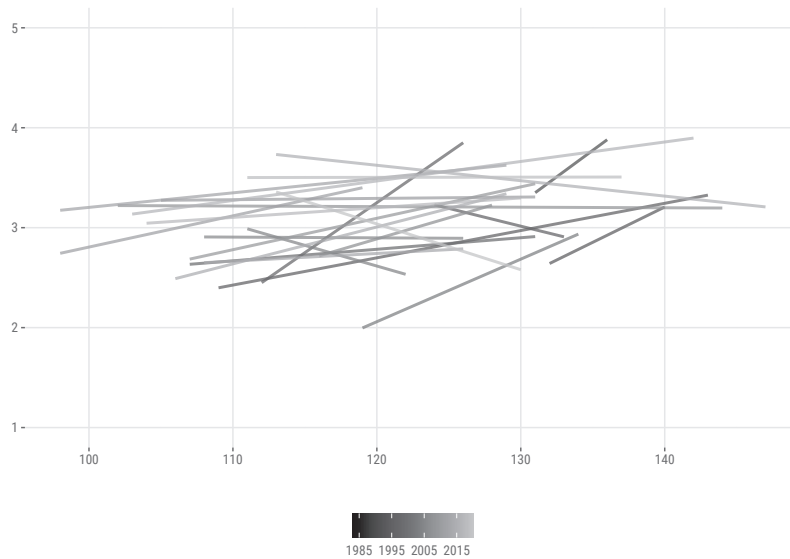


Figure 9.4: Release year effects on length and rating.

9.3.2 Overview of mixed models

What we've just seen might initially bring to mind an interaction effect, and that's the right way to think about it! A **mixed model** can be used to incorporate that type of relationship into our model, which we can think of as a group interaction, without much hassle and additional explainability. But

it's actually a quite flexible class that can also allow for more complicated but related types.

Before going too much further, the term *mixed model* is as vanilla as we can possibly make it, but you might have heard of different names such as *hierarchical linear models*, or *multilevel models*, or maybe *mixed-effects models*. Maybe you've even been exposed to ideas like *random effects* or *random slopes*. These are in fact all instances of what we're calling a *mixed model*.

What makes a model a *mixed* model? The mixed model is characterized by the idea that a model can have **fixed effects** and **random effects**. Fortunately, you've already encountered *fixed* effects – those are the features that we have been using in all of our models so far! We are assuming a single true parameter (e.g., coefficient/weight) to estimate for each of those features, and that parameter is *fixed*.

In mixed models, a *random effect* instead comes from a specific distribution, and this is almost always a normal distribution. This random effect adds a unique source of variance in the target variable. This distribution of effects can be based on a grouping variable (such as genre), where we let those parameters, i.e., coefficients (or weights), vary across the groups, creating a distribution of values.

Let's take our initial example with movie length and genre. Formally, we might specify something like this:

$$\text{Rating} = b_{\text{int[genre]}} + b_{\text{length}} * \text{length} + \epsilon$$

In this formula, we are saying that genre has its own unique effect for this model in the form of specific intercepts for each genre[3]. This means that whenever an observation belongs to a specific genre, it will have an intercept that reflects that genre, and that means that two observations with the same movie length but from different genres would have different predictions.

We also posit, and this a key point, that those group effects come from a *random* distribution. We can specify that as:

$$b_{\text{int[genre]}} \sim \text{N}(b_{\text{intercept}}, \sigma_{\text{int\_genre}})$$

This means that the random intercepts will be normally distributed and the overall intercept is just the mean of those random intercepts, and with its own variance. It also means our model will have to estimate that variance along with our residual variance. Another very common depiction is as follows, which makes explicit the addition of a random effect to the intercept:

[3]The error term ϵ is still assumed normal with mean zero and variance σ^2.

$$\text{re}_{[\text{int\_genre}]} \sim \text{N}(0, \sigma_{\text{int\_genre}})$$

$$b_{\text{int[genre]}} = b_{\text{intercept}} + \text{re}_{[\text{int\_genre}]}$$

The same approach would apply with a random slope, where we would have a random slope for each group, and that random slope would be normally distributed with its own variance. Compared to our first formulation, it would look like this:

$$b_{\text{length[genre]}} \sim \text{N}(b_{\text{length}}, \sigma_{\text{length\_genre}})$$

A simple random intercept and slope is just the start. As an example, we can let the intercepts and slopes correlate with one another, and we could have multiple grouping factors, as well as allowing multiple features and even interactions themselves to vary by group! So mixed models can get quite complex, but the basic idea is still the same: we are allowing parameters to vary across groups, and we are estimating the variance of those parameters.

9.3.3 Using a mixed model

One of the key advantages of a mixed model is that we can use it when the observations within a group are not independent. This is a very common situation in many fields, and it's a good idea to consider this when you have grouped data. As an example we'll use the happiness data for all available years, and we'll consider the country as a grouping variable. In this case, observations within a country are likely to be more similar to each other than to observations from other countries. Such **longitudinal data** is a classic example of when to use a mixed model. This is also a case where we wouldn't just throw in 'country' as a feature like any other factor, since there are 164 countries in the data. We need an easier way to handle so many groups!

In general, to use mixed models, we have to specify a random effect pertaining to the categorical feature of focus, but that's the primary difference from our previous approaches used for linear or generalized linear models. For our example, we'll look at a model with a random intercept for the country feature, and one that adds a random coefficient for the yearly trend across countries. This means that we are allowing the intercepts and slopes to vary across countries, and the intercepts and slopes can correlate with one another. Furthermore, by recoding year to start at zero, the intercept will represent the happiness score at the start of the data. In addition, to see a more reasonable effect, we also divide the yearly trend by 10, so the coefficient provides the change in happiness score per decade.

R

We'll use the lme4 package in R which is the most widely used package for mixed models.

```
library(lme4)

df_happiness_all = read_csv("https://tinyurl.com/worldhappinessallyears")

df_happiness_all = df_happiness_all |>
    mutate(decade_0 = (year - min(year))/10)

# random intercepts are specified by a 1
model_ran_int = lmer(
    happiness_score ~ decade_0 + (1| country),
    df_happiness_all
)

model_ran_slope = lmer(
    happiness_score ~ decade_0 + (1 + decade_0 | country),
    df_happiness_all
)

# not shown
summary(model_ran_int)
summary(model_ran_slope)
```

Python

As with our recommendation with GAMs later, R is the better tool for mixed models, as the functionality is overwhelmingly better there for modeling and post-processing. However, you can use statsmodels in Python to fit them as well[4].

```
import statsmodels.api as sm

df_happiness_all = pd.read_csv("https://tinyurl.com/worldhappinessallyears")

df_happiness_all = (
    df_happiness_all
    .assign(decade_0 = lambda x: (x['year']- x['year'].min())/10)
```

[4]One of your authors worked for several years with the key developer of the mixed models functionality in statsmodels. As such, we can say there is zero doubt about the expertise going into its development, as there are few in the world with such knowledge. Even so, you probably won't find the functionality is as mature or as expansive as what you get in R.

```
)

model_ran_int = sm.MixedLM.from_formula(
    "happiness_score ~ decade_0",
    df_happiness_all,
    re_formula='1',
    groups=df_happiness_all["country"]
).fit()

model_ran_slope = sm.MixedLM.from_formula(
    "happiness_score ~ decade_0",
    df_happiness_all,
    re_formula='1 + decade_0',
    groups=df_happiness_all["country"]
).fit()

# not shown
# model_ran_int.summary()
# model_ran_slope.summary()
```

Table 9.4 shows some typical output from a mixed model, focusing on the random slope model. The fixed effect part (Fixed) is your basic GLM result and interpreted the same way. Nothing is new there, and we can see a slight positive decade trend in happiness, though maybe not a strong one. But the random effects (Random) are where the action is! We can see the standard deviation of the random effects, i.e., the intercepts and slopes. We can also see the correlation between the random intercepts and random slopes. And finally we also have the residual (observation level) standard deviation, which is interpreted the same as with standard linear regression. Depending on your modeling tool, the default result may be in terms of variances and covariances rather than standard deviations and correlations, but it is generally similar.

Table 9.4: Mixed Model Results

Parameter	Coefficient	SE	CI_low	CI_high	Group
Fixed					
(Intercept)	5.34	0.09	5.16	5.52	
decade_0	0.06	0.05	-0.04	0.16	
Random					
SD (Intercept)	1.15				country
SD (decade_0)	0.57				country
Cor (Intercept~decade_0)	-0.38				country
SD (Observations)	0.34				Residual

In this case, we can see notable variability attributable to the random effects. How do we know this? Well, if our happiness score is on a 1 to 8 scale, and we have a standard deviation of happiness of 1.13 before accounting for anything else, then we might surmise that having an effect of that size (roughly 1.15) is a relatively notable amount, as it suggests we would move around about a standard deviation of happiness just going from country to country (on average).

The trend variability is also notable. Although the overall effect is small, the coefficient standard deviation shows that it ranges from a notable negative trend to a notable positive trend ($\pm$ 0.57) depending on which country we're looking at. In addition, we can also see that the correlation between the random intercepts and random slopes is negative, which means that the groups with higher starting points have more negative slopes.

Now let's look at the estimates for the random effects for the model with both intercepts and slopes[5].

R

```
estimated_RE = ranef(model_ran_slope)
```

Python

```
estimated_RE = pd.DataFrame(model_ran_slope.random_effects)
```

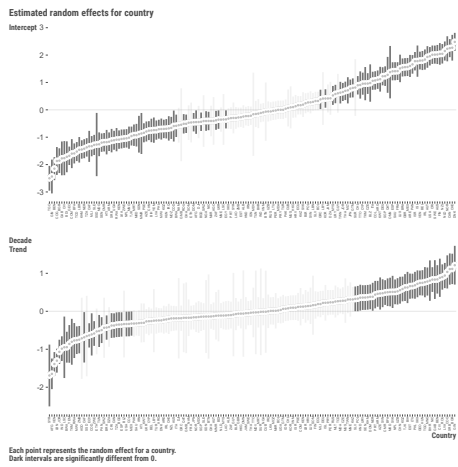


Figure 9.5: Estimated random effects.

[5]MC provides a package for mixed models in R called mixedup. It provides a nice way to extract random effects and summarize such models (link).

How do we interpret these *deviations*? For starters, they are deviations from the fixed effect for the intercept and decade trend coefficient. Here that means anything negative is an intercept or slope below the corresponding fixed effect value, and anything positive is above that value. If we want the specific effect for a country, we just add its random effect to the fixed effect, and we can refer to those as **random coefficients**.

For example, if we wanted to know the effects for the US, we would add its random effects to the population level fixed effects. This is exactly what we did in the previous section in our interpretation with the interaction model. However, you can typically get these from the package functionality directly. The result shows that the US starts at a very high happiness score, but actually has a negative trend over time[6].

R

```
# coef(model_ran_slope) stored here
ranef_usa = estimated_RE$country
ranef_usa = ranef_usa |>
    rownames_to_column('country') |>
    filter(country == 'United States')

ranef_usa[1, c('(Intercept)',  'decade_0')] + fixef(model_ran_slope)
```

```
  (Intercept) decade_0
1       7.296  -0.2753
```

Python

```
ranef_usa = estimated_RE['United States'].rename({'Group': 'Intercept'})
ranef_usa + model_ran_slope.fe_params
```

```
Intercept    7.296
decade_0    -0.275
dtype: float64
```

> **Averages in Mixed Models**
>
> In the linear mixed effect model setting with a random intercept, the fixed effects can be seen as the (population) average effects, but this is not exactly what you are getting from the mixed model. To make the distinction clear, consider family groups and a gender effect for males vs. females. The linear regression and some other types of models

[6] Life expectancy in the US has actually declined recently.

(e.g., estimated via generalized estimating equations) would give you the average effect male-female difference across all families. The mixed model actually tells you the male-female difference as if they were in the same family (e.g., siblings). Again, in the simplest mixed model setting these are the same. Beyond that, when we start dealing with random slopes and non-Gaussian distributions, they are not.
In general, if we set the random effect to 0 to get a prediction, that tells us what the prediction would be for a typical group, in this case, a typical country. Often we want to get something more like the average slope or prediction across countries that we would have with linear regression. This gets us back to the idea of the **marginal effect** we discussed earlier. While the mechanics are not straightforward for mixed models, the tool use generally takes no additional effort.

Let's plot those random coefficients together to see how they relate to each other.

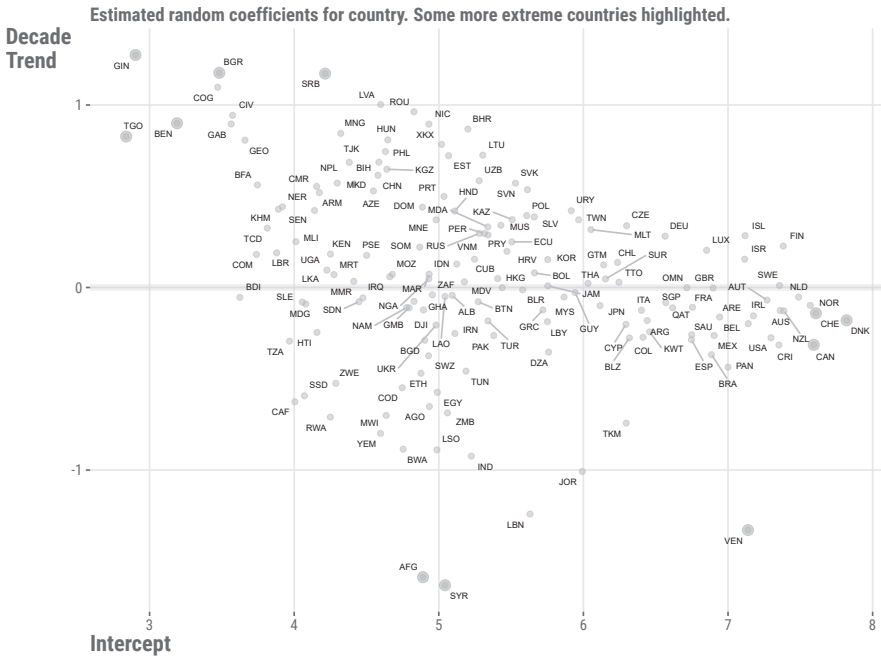


Figure 9.6: Random effects for a mixed model.

From this plot, we can sense why the estimated random effect correlation was negative. For individual country results, we can see that recently war-torn regions like Syria and Afghanistan have declined over time even while they

started poorly as well. Some like Guinea and Togo started poorly but have improved remarkably over time. Many western countries started high and mostly stayed that way, though generally with a slight decline. Perhaps there's only one direction to go when you're already starting off well!

> **Always Scale Features for Mixed Models**
>
> Your authors have run a lot of these models. Save yourself some trouble and standardize or otherwise scale your features before fitting the model. Just trust us, or at least don't be surprised when your model doesn't converge.

9.3.4 Mixed model summary

For a model with just one feature, we certainly had a lot to talk about! And this is just a glimpse of what mixed models have to offer, and the approach can be even richer than what we've just seen. But you might be asking: Why don't I just put genre or country in the model like other categorical features? In the case of genre for movie reviews where there are few groups, that's okay. But doing that with features with a lot of levels would typically result in estimation issues due to having so many parameters to estimate. In general mixed models provide several advantages for the data scientist:

- Any coefficient can be allowed to vary by groups, including other random effects. It actually is just an interaction in the end as far as the linear predictor and conceptual model is concerned.
- The group-specific effects are *penalized*, which shrinks them toward the overall mean and makes this a different approach from just adding a 'mere interaction'. This helps avoid overfitting, and that penalty is related to the variance estimate of the random effect. In other words, you can think of it as running a penalized linear model where the penalty is applied to the group-specific effects (see Section 6.8).
- Unlike standard interaction approaches, we can estimate the covariance of the random effects, and specify different covariance structures for observations within groups. Standard interactions implicitly assume independence.
- Because of the way they are estimated, mixed models can account for lack of independence of observations[7], which is a common issue in many datasets. This is especially important when you have repeated measures, or when you have a hierarchical structure in your data, such as students within schools, or patients within hospitals.

[7] Independence of observations is a key assumption in linear regression models, and when it's violated, the standard errors of the coefficients are biased, which can lead to incorrect inferences. Rather than hacking a model (so-called 'fixed effects' models) or 'correcting' the standard error (e.g., with some 'sandwich' or estimator), mixed models can account for this lack of independence through the model itself.

- Standard modeling approaches can estimate these difficult models very efficiently, even with thousands of groups and millions of observations.
- The group effects are like a very simplified **embedding** (Section 14.2.2), where we have taken a categorical feature and turned it into a numeric one, like those shown in Figure 9.5. This may help you understand other embedding techniques that are used in other places like deep learning.
- When you start to think about random effects and/or distributions for effects, you're already thinking like a Bayesian (Section 7.6), who is always thinking about the distributions for various effects. Mixed models are the perfect segue from standard linear model estimation to Bayesian estimation, where everything is random.
- The random effect is akin to a latent variable of 'unspecified group causes'. This is a very powerful idea that can be used in many different ways, but importantly, you might want to start thinking about how you can figure out what those 'unspecified' causes may be!
- Group effects will almost always improve your model's predictive performance relative to not having them, especially if you weren't including those groups in your model because of how many groups there were.

In short, mixed models are a fun way to incorporate additional interpretive color to your model, while also getting several additional benefits to help you understand your data!

9.4 Generalized Additive Models

9.4.1 When straight lines aren't enough

Linear models, as their name implies, generally assume a linear relationship between the features and the target by default. But fitting a line through your data isn't always going to be the best approach. Not every relationship is linear and not every relationship is monotonic. Sometimes, you need to be able to model a relationship that has a fair amount of nonlinearity – they can appear as slight curves, waves, and any other type of wiggle that you can imagine.

In other words, we can go from the straight line in Figure 9.7:

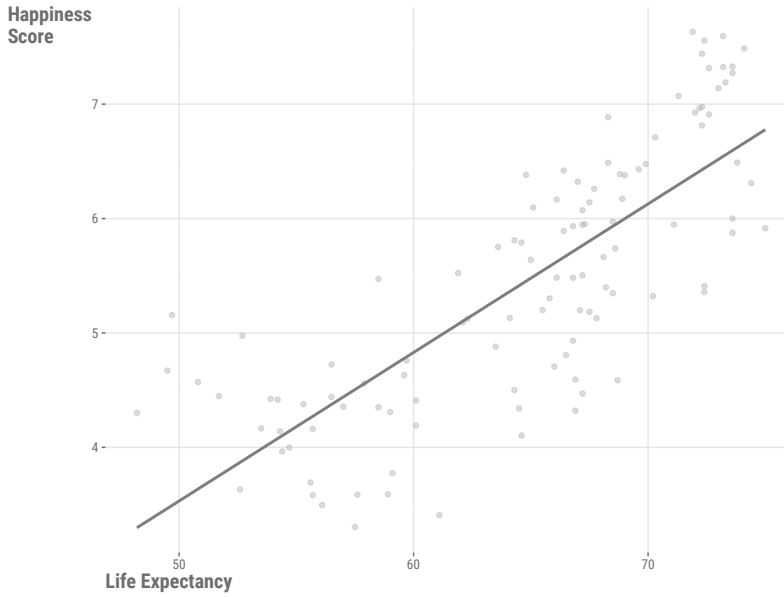


Figure 9.7: Standard linear relationship.

To the curve seen in Figure 9.8:

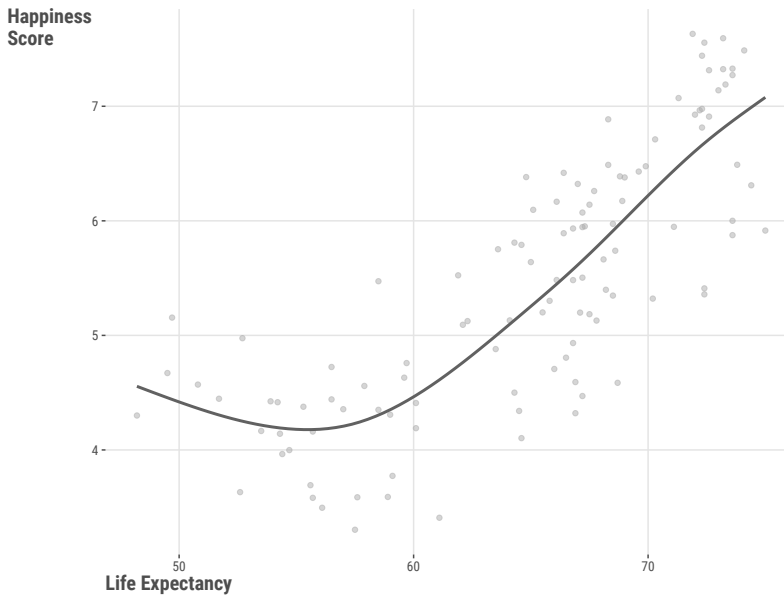


Figure 9.8: Curvilinear relationship.

There are many ways to go about this, and common approaches include using polynomial regression, using feature transformations, or using a nonlinear regression model. The curved line in Figure 9.8 is the result of what's called a **spline**. It is created by a feature and expanding it to multiple columns, each of which is a function of the original feature. We then fit a linear model to that data as usual. What this ultimately means is that we can use a linear model to fit a curve through the data. While this might not give us the same tidy explanation that a typical linear approach would offer, we will certainly get better predictions if it's appropriate, and a better understanding of the reality and complexity of the true relationship. It's also useful for exploratory purposes, and visualization tools can make it easy. Here is some demonstration code (result not shown).

R

```
x = rnorm(1000)
y = sin(x)

tibble(x, y) |>
    ggplot(aes(x = x, y = y)) +
    geom_smooth(method = 'gam', se = FALSE)
```

Python

```
import plotly.graph_objects as go
import numpy as np

x = np.random.normal(size = 1000)
y = np.sin(x)

fig = go.Figure()
fig.add_trace(
    go.Scatter(
        x = x,
        y = y,
        line_shape = 'spline'
    )
)
```

Models incorporating this type of effect belong to a broad group of *generalized additive models* (**GAM**s). When we explored interactions and mixed models, we explored how the feature-target relationship varies with another feature. There we focused on our feature and its relationship to the target at different values of other features. When we use a GAM, our initial focus is on a specific

feature and how its relationship with the target changes at different values of that feature. How are we going to do this, you might ask? Conceptually, we will have a model that looks like this:

$$y = f(X) + \epsilon$$

This isn't actually any different than what you've seen, it really isn't! It's just shorthand for the input X being fed into a function $f()$ of some kind, exactly as we saw in Section 2.3.1. That function is very flexible and can be anything you want.

What we're going to do with the function now is expand the feature x in some way, which on a practical level means it will actually become multiple columns of input instead of just one. Some approaches to creating the expansion can be quite complex, with the goal of tackling spatial, temporal, or other aspects of the data. But, in the end, it's just extra columns in the **model matrix** that go into the model-fitting function like any other feature. This enables the model to explore a more complex representation space, ultimately helping us capture nonlinear patterns in our data.

At this point, you might be asking yourself, "Why couldn't I just use some type of polynomial regression or even a nonlinear regression?". Of course you could, but both have limitations relative to a GAM. If you are familiar with polynomial regression, where we add columns that are squares, cubes, etc. of the original feature, you can think of GAMs as a more general approach, and very similar in spirit. But that polynomial approach assumes a specific form of nonlinearity and has no regularization. This means that it tends to overfit the data you currently have, and *you* are forcing curves to fit through the data, rather than exploring what naturally may arise.

Another approach besides polynomials is to use a **nonlinear regression model**. In this setting, you need to know what the underlying functional form is. An example is a logistic growth curve model for population growth. Without taking extra steps, such models can also overfit. Furthermore, outside of well-known physical, chemical, or biological processes, it's rarely clear what the underlying functional form should be. At the very least, we wouldn't know a formula for life expectancy and happiness!

GAMs are better in such settings because they fit the data well without needing to know the underlying form. They also prevent overfitting in smaller data and/or more complex settings by using a penalized estimation approach. We can use them for multiple features at once, and even include interactions between features. We also can use different types of splines, which you can think of as different functions to apply to features, to capture different types of nonlinearities. Here is another formal definition of a GAM that makes it more clear we can deal with multiple features.

$$\hat{y} = \sum X_j \beta_j$$

In this case, each X_j is a matrix of the feature and its **basis expansion**, which is created by the spline function. The β_j are the coefficients for each of those basis expansion columns. But a specific X_j could also just be a single feature and it's coefficient to model a linear relationship. You can pick and choose, which features to use it on, and how to use it.

The nice thing is that you don't have to worry about the details of the basis expansion. The package you choose will take care of that for you. You'll have different options, and often the default is fine, but sometimes you'll want to adjust the technique and how 'wiggly' you want the curve to be.

9.4.2 A standard GAM

Now that you have some background, let's give this a shot! In most respects, we can use the same sort of approach as we did with our other linear model examples. For our example here, we'll use the model that was depicted in Figure 9.8, which looks at the relationship between life expectancy and happiness score from the world happiness data (2018). Results are simplified in the subsequent table.

R

We'll use the very powerful `mgcv` package in R. The `s` function will allow us to use a spline approach to capture the nonlinearity. The `bs` argument specifies the type of spline, and here we use a B-spline, but there are many options. The `k` argument provides a means to control the complexity of the spine, historically thought of as 'knots' where bends in the curve can occur.

```
library(mgcv)

df_happiness_2018 = read_csv('https://tinyurl.com/worldhappiness2018')

model_gam = gam(
    happiness_score ~ s(healthy_life_expectancy_at_birth, bs = 'bs'),
    data = df_happiness_2018
)

summary(model_gam)
```

Python

We can use the statsmodels package in Python to fit a GAM, or alternatively, pygam, and for consistency with previous models, we'll choose the former.

Honestly though, you should use R's mgcv, as these require notably more work without having some of the basic functionality available there. In addition, there is an ecosystem of R packages to further extend mgcv, while these are not as mature in Python. Here we must create the spline explicitly first, a so-called B-spline, and add it to the model.

```
from statsmodels.gam.api import GLMGam, BSplines

df_happiness_2018 = pd.read_csv('https://tinyurl.com/worldhappiness2018')

bs = BSplines(
    df_happiness_2018['healthy_life_expectancy_at_birth'],
    df = 9,
    degree = 3
)

gam_happiness = GLMGam.from_formula(
    'happiness_score ~ healthy_life_expectancy_at_birth',
    smoother = bs,
    data = df_happiness_2018
)

model_gam = gam_happiness.fit()

model_gam.summary()
```

Table 9.5: GAM Model Results

Component	Term	Estimate	Std.Error	t.value	p.value
parametric coefficients	Intercept	5.44	0.06	92.73	0
		EDF	Ref.df	F.value	p.value
smooth terms	s(Life Exp.)	5.55	6.49	40.11	0

When you look at the model output, what you get will depend a lot on the tool you use, and the details are mostly beyond the scope we want to present here (check out Michael Clark's (2022b) document on GAMs for a lot more). But in general, the following information will be provided as part of the summary ANOVA table or as an attribute of the model object:

- **Coefficients**: The coefficients for each of the features in the model. For a GAM, these are the coefficients for the basis expansion columns, as well as standard linear feature effects.
- **Global test of a feature**: Some tools will provide a statistical test of the significance of the entire feature's basis expansion, as opposed to just the

individual coefficients. In the table we have the intercept and the summarized smooth term.

- **EDF/EDoF**: Effective degrees of freedom. This is a measure of 'wiggle' in the relationship between the feature and the target. The higher the value, the more wiggle you have. If you have a value close to 1, then you have a linear relationship. With our current result, we can be pretty confident that a nonlinear relationship gives a better idea about the relationship between a country's life expectancy and happiness than a linear one.
- **R-squared**: Adjusted/Pseudo R^2 or *deviance explained*. This is a measure of how much of the variance in the target is explained by the model. The higher the value, the better the model. Deviance explained is an analog to the unadjusted R^2 value for a Gaussian model that is used in the GLM setting. It's fine as a general assessment of prediction-target correspondence, and in this case, we might be feeling pretty good about the model.

Far more important than any of these is the visual interpretation, and we can get plots from GAMs easily enough.

R

```
# not shown
plot(model_gam)
```

Python

```
# not shown
model_gam.plot_partial(0, cpr=True)
```

Unfortunately, the default package plots are not pretty, and sadly they also aren't provided in the same way we'd expect for interpretation. But they're fine for a quick look at your wiggly result. We provide a better looking one here[8]. The main interpretation is that there is not much relationship between life expectancy and happiness score until you get to about 60 years of life expectancy, and then it increases at a faster rate. Various tools are available to easily plot the derivatives for more understanding.

[8]We used the see package in R for a quick plot. We also recommend the functionality via the gratia package to visualize the derivatives, which will show where the feature effect is changing most.

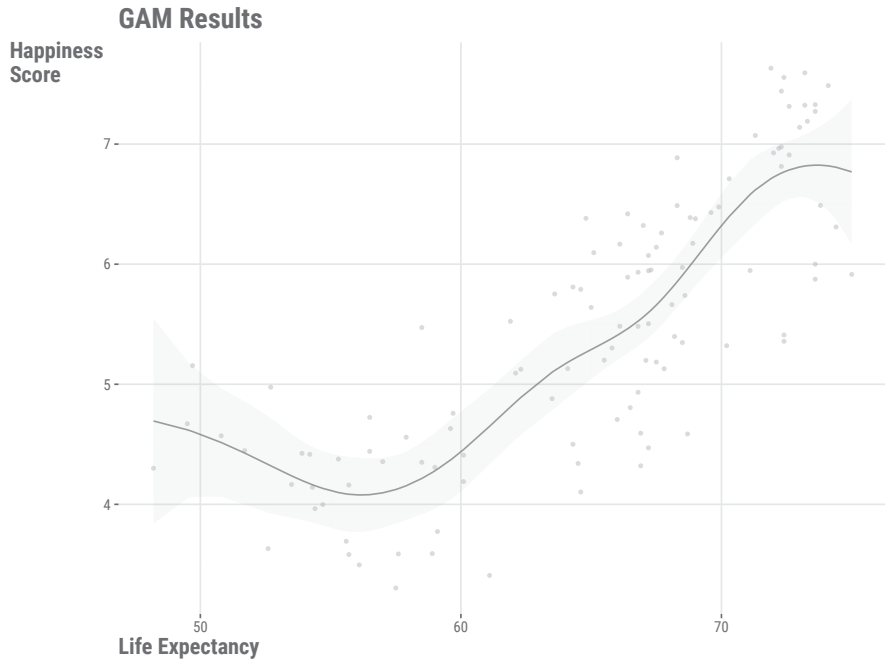


Figure 9.9: Visualizing a GAM.

9.4.3 GAM Summary

To summarize, we can use a GAM to model nonlinear relationships with a linear model approach. We can use splines to capture those nonlinearities, and we can use a penalized approach to control the amount of wiggle in our model. What's more, we can interact the wiggle with other categorical and numeric features to capture even more complexity in our data. This allows us to model spatial, temporal, and other types of data that have complex relationships.

GAMs are a *very* powerful modeling tool that take us a step toward more complex models, but without the need to go all the way to a neural network or similar. Plus they still provide standard statistical inference information. In short, they're a great tool for modeling!

> **i** GAMs Are Random Effects Models
>
> It turns out that GAMs have a very close relationship to mixed models, where splines can be thought of as random effects (see Gavin Simpson's post also), and GAMs can even incorporate the usual random effects for categorical variables. So you can think of GAMMs, or generalized additive mixed models, as a way to combine the best of both GAM and

mixed model worlds into one extremely powerful modeling that covers a lot of ground. It's also a great baseline and sanity check model for when you're trying boosting or deep learning models, where a well-specified GAMM can be hard to beat for tabular data and still can be highly interpretable, while coming with built-in uncertainty estimates and better visualization tools.

9.5 Quantile Regression

People generally understand the concept of the arithmetic mean, or 'average'. You see it some time during elementary school, it gets tossed around in daily language, and it is statistically important. After all, so many distributions depend on it! Why, though, do we feel so tied to it from a regression modeling perspective? Yes, it has handy features, but it can limit the relationships we can otherwise model effectively. Here we'll show you what to do when the mean betrays you – and trust us, the mean will betray you at some point!

9.5.1 When the mean breaks down

In a perfect data world, we like to assume the mean is equal to the middle observation of the data: the *median*. But that is only when things are symmetric though, and usually our data comes loaded with challenges. Skewness and even just a few extreme scores in your data may cause a rift between the median and the mean.

Let's say we take the integers between 1 and 10, and find the mean.

$$\frac{1+2+3+4+5+6+7+8+9+10}{10} = 5.5$$

The middle value in that vector of numbers would also be 5.5.

What happens if we replace the 1 with a more extreme value, like -10?

$$\frac{-10+2+3+4+5+6+7+8+9+10}{10} = 4.5$$

With just one dramatic change, our mean went down by a whole point. The median observation, though, is still 5.5. In short, the median is invariant to wild swings out in the tails of your numbers.

You might be saying to yourself, "Why should I care about this central tendency chicanery?". Let us tell you why! The least squares approach to the standard

linear model dictates that the regression line needs to be fit through the means of the variables. If you have extreme scores that influence the mean, then your regression line will also be influenced by those extreme scores.

Consider the following regression line:

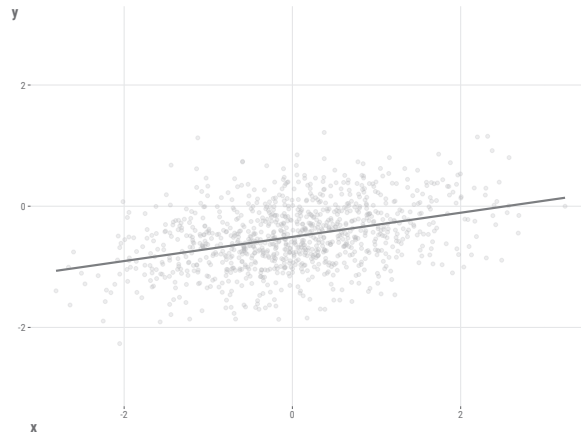


Figure 9.10: Linear relationship without extreme scores.

Now, what would happen if we replaced a few of our observations with extreme scores?

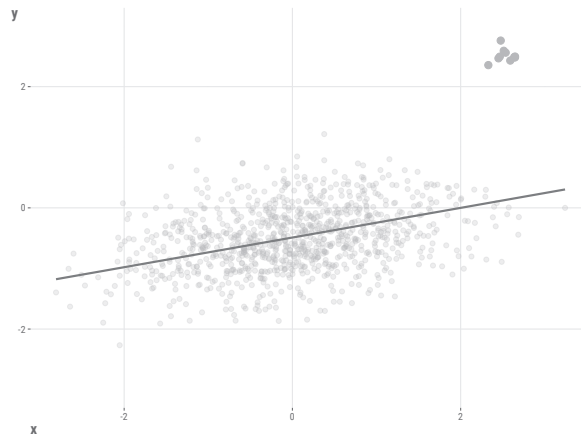


Figure 9.11: Linear relationship with extreme scores.

With just a casual glance, it doesn't look like our two regression lines are that different. They both look like they have a similar positive slope, so all should be good. To offer a bit more clarity, let's put those lines in the same space:

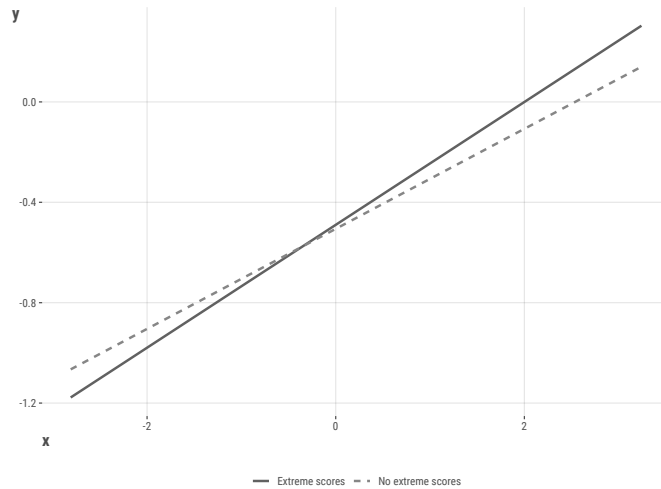


Figure 9.12: Linear relationships with and without extreme scores.

With 1000 observations, we see that having just 10 relatively extreme scores is enough to change the regression line, even if just a little. But that little bit can mean a huge difference for predictions and the conclusions we come to.

There are a few approaches we could take here, with common approaches being dropping those observations, winsorizing them, or doing some transformation. Throwing away data because you don't like the way it behaves is almost statistical abuse, and winsorization is just replacing those extreme values with numbers that you like a little bit better. Transformations done independently of the model rarely work for this purpose either. So let's try something else!

9.5.2 A standard quantile regression

A better answer to this challenge might be to try a median-based approach instead. This is where a model like **quantile regression** becomes handy. Quantile regression is a type of regression that allows us to model the relationship between the features and the target at different quantiles of the target. For example, we can examine models at the 10th, 25th, 50th, 75th, and 90th percentiles of the target. This is very cool, as it allows us to model the relationship between the features and the target in a way that is robust to outliers and extreme scores. It's also a way to understand a type of nonlinearity that

is not captured by a standard linear model, as the feature target relationship may change at different quantiles of the target.

To demonstrate this type of model, let's use our movie reviews data. Let's say that we are curious about the relationship between the `word_count` and `rating` to keep things simple. To make it even more straightforward, we will use the standardized (scaled) version of the feature. In our default approach, we will start with a median regression.

R

```
library(quantreg)

model_median = rq(rating ~ word_count_sc, tau = .5, data = df_reviews)

summary(model_median)
```

Python

```
model_median = smf.quantreg('rating ~ word_count_sc',  data = df_reviews)
model_median = model_median.fit(q = .5)

model_median.summary()
```

Table 9.6: Quantile Regression Model Results

feature	coef	conf.low	conf.high
(Intercept)	3.09	3.05	3.26
word_count_sc	−0.29	−0.40	−0.20

Fortunately, our interpretation of this result isn't all that different from a standard linear model. The rating should decrease by -0.29 for every bump in a standard deviation for the number of words, which in this case is about 5 words. However, this concerns the expected *median* rating, not the mean, as would be the case with standard linear regression.

Quantile regression is not a one-trick pony though – being able to compute a median regression is just the default. As mentioned, we can also explore different quantiles, and this gives us the ability to answer brand new questions such as, "Does the relationship between word count and their ratings change at different quantiles of rating?". Very cool!

Let's now examine the trends within five different quantiles of the data: .1, .3 .5, .7, and .9[9]. We aren't limited to just those quantiles though, and you can examine any of them that you might find interesting. Here is a plot of the results of these models.

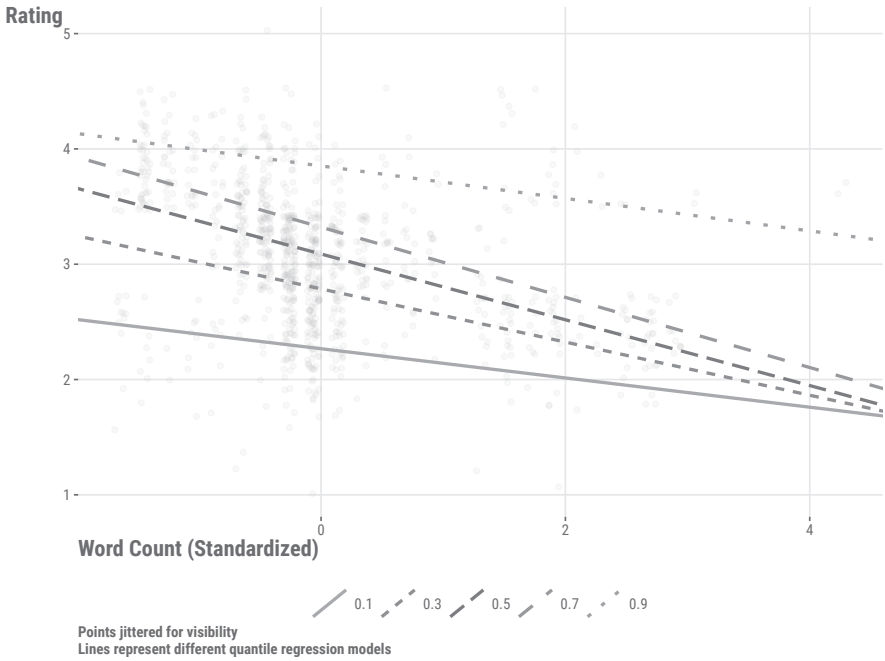


Figure 9.13: Quantile regression lines.

To interpret our visualization, we could start by saying that all of the model results suggest a negative relationship. The 10th and 90th quantiles seem weakest, while those in the middle show a notably stronger relationship. We can also see that the 90th percentile model is better able to capture those values that would otherwise be deemed as outliers using other standard techniques. The following table shows the estimated coefficients for each of the quantiles, and it suggests that all word count relationships are statistically significant, since the confidence intervals do not include zero.

[9]The R function can take a vector of quantiles, while the Python function can only take a single quantile, so you would need to loop through the quantiles.

Table 9.7: Quantile Regression Model Results

feature	coef	SE	CI_low	CI_high	quantile
(Intercept)	2.27	0.03	2.21	2.33	tau (0.1)
word_count_sc	−0.13	0.03	−0.19	−0.07	tau (0.1)
(Intercept)	2.79	0.03	2.73	2.84	tau (0.3)
word_count_sc	−0.23	0.02	−0.27	−0.19	tau (0.3)
(Intercept)	3.09	0.02	3.06	3.12	tau (0.5)
word_count_sc	−0.29	0.01	−0.31	−0.26	tau (0.5)
(Intercept)	3.32	0.02	3.28	3.36	tau (0.7)
word_count_sc	−0.30	0.02	−0.34	−0.27	tau (0.7)
(Intercept)	3.85	0.05	3.76	3.95	tau (0.9)
word_count_sc	−0.14	0.06	−0.25	−0.03	tau (0.9)

95% confidence intervals are shown.

9.5.3 The quantile loss function

Formally, the objective function for the quantile regression model can be expressed as:

$$\text{Value} = \Sigma \left((\tau - 1) \sum_{y_i < \hat{y}} (y_i - \hat{y}) + \tau \sum_{y_i \geq \hat{y}} (y_i - \hat{y}) \right) \tag{9.1}$$

Compared to a standard linear regression, we are given an extra parameter for the model: τ. It's a number between 0 and 1 representing the desired quantile (e.g., 0.5 for the median)[10]. The objective function treats positive residuals differently than negative residuals. If the residual is positive, then we multiply it by τ. If the residual is negative, then we multiply it by $\tau - 1$. The increased penalty for over-predictions ensures that the estimated quantile $\hat{y}$ is such that τ proportion of the data falls below it, balancing the total loss and providing a robust estimate of the desired quantile.

9.5.4 DIY

Given how relatively simple the objective function is, let's demystify this model further by creating our own quantile regression model and see if we can get the same results. We'll start by creating a loss function that we can use to fit our model.

[10]This is equivalent to using the least absolute deviation objective.

R

```
quantile_loss = function(par, X, y, tau) {
    y_hat = X %*% par

    residual = y - y_hat

    loss = ifelse(
        residual < 0,
        (tau - 1) * residual,
        tau * residual
    )

    sum(loss)
}
```

Python

```
def quantile_loss(par, X, y, tau):
    y_hat = X.dot(par)

    residual = y - y_hat

    loss = np.where(
        residual < 0,
        (tau-1)*residual,
        tau*residual
    )

    return sum(loss)
```

This code is just the embodiment of the formula shown for the quantile objective function. Compared to some of our other approaches demonstrated, we add an argument for τ, but we otherwise proceed very similarly. We calculate the residuals or errors in prediction, and then we sum the loss function based on those errors.

Now that we have our data and our loss function, we can fit the model almost exactly like our standard linear model. Again, note the τ value, which we've set to .5 to represent the median.

R

```
X = cbind(1, df_reviews$word_count_sc)
y = df_reviews$rating


optim(
    par = c(intercept = 0, word_count_sc = 0),
    fn  = quantile_loss,
    X   = X,
    y   = y,
    tau = .5
)$par
```

```
 intercept word_count_sc
    3.0886       -0.2852
```

Python

```
from scipy.optimize import minimize
import numpy as np

X = pd.DataFrame(
    {'intercept': 1,
    'word_count_sc': df_reviews['word_count_sc']}
)
y = df_reviews['rating']

minimize(quantile_loss, x0 = np.array([0, 0]), args = (X, y, .5)).x
```

```
array([3.0901, -0.2842])
```

Let's compare this to our previous result, and the OLS results as well. As usual, our simple code does what we need it to do! We also see that the linear regression model would produce a relatively smaller coefficient.

Table 9.8: Comparison of Quantile Regression Models

feature	OLS	QReg	Ours
intercept	3.051	3.089	3.089
word_count_sc	−0.216	−0.285	−0.285

Yet Another Interaction

One way to interpret this result is that we have a nonlinear relationship between the word count and the rating, in the same way we had an interaction previously. In this case, our effect of word count interacts with the target! In other words, we have a different word count effect for different ratings. This is a bit of a mind bender to process, but it's another good example of how a linear approach can be used to model quirky relationships!

9.6 Wrapping Up

The standard linear model is useful across many different data situations. It does, unfortunately, have some issues when data becomes a little bit more "real". When you have extreme scores or relationships that a standard model might miss, you don't necessarily need to abandon your linear model in favor of something more exotic. Instead, you might just need to think about how you are actually fitting the line through your data.

9.6.1 The common thread

The models discussed in this chapter are all linear models, but they add flexibility in how they model the relationship between the features and the target, and provide a nonlinear aspect to the otherwise linear model. Furthermore, with tools like mixed models, GAMs, and quantile regression, we generalize our GLMs to handle even more complex data settings.

9.6.2 Choose your own adventure

No matter how much we cover in this book, there is always more to learn. Hopefully you've got a good grip on linear models and related topics, so feel free to try out some machine learning in Chapter 10!

9.6.3 Additional resources

There is no shortage of great references for mixed effects models. If you are looking for a great introduction to mixed models, we would recommend starting with yet another tutorial by one of your fearless authors! Michael Clark's *Mixed Models with R* (2023) is a great introduction to mixed models and is available for free. For a more comprehensive treatment, you can't go wrong with Gelman & Hill's *Data Analysis Using Regression and Multilevel/Hierarchical*

Models (2006), and their more recent efforts in *Regression and Other Stories* (2020), which will soon have an added component for mixed models: *Advanced Regression and Multilevel Models* (2025).

If you want to dive more into the GAM world, we would recommend that you start with the **Moving Beyond Linearity** chapter in *An Introduction to Statistical Learning* (James et al. 2021). Not only do they have versions for both R and Python, but both have been made available online. If you are wanting something more technical, you can't beat Simon Wood's book, *Generalized Additive Models: An Introduction with R* (2017), or a more digestible covering of the same content by one of your own humble authors (Clark 2022b).

For absolute depth on quantile regression, we will happily point you to the OG of quantile regression, Roger Koenker. His book, *Quantile Regression* (2005) is a must read for anyone wanting to dive deep into quantile regression, or just play around with his R package `quantreg`. *Galton, Edgeworth, Frisch, and Prospects for Quantile Regression in Econometrics* is another of his. And finally, you might also consider Fahrmeir et al. (2021), which takes an even more generalized view of GLMs, GAMs, mixed models, quantile regression, and more (very underrated).

9.7 Guided Exploration

These models are so much fun, so you should feel comfortable just swapping any feature(s) in and out of the models. For example, for the mixed model, try using GDP per capita or life expectancy instead of (just a) trend effect. For the GAM, try using several features with nonlinear effects and see what shakes out. For quantile regression, try a different feature like movie length and use different quantiles.

R

```
library(lme4)
library(mgcv)
library(quantreg)

df_happiness = read_csv('https://tinyurl.com/worldhappinessallyears')

model_mixed = lmer(
    happiness_score ~ ??? + (1 | country),
    data = df_happiness_2018
)
```

```
model_gam = gam(
    happiness_score ~ s(???)...,
    data = df_happiness
)

model_quant = rq(
    happiness_score ~ ???,
    tau = ???,
    data = df_happiness
)
```

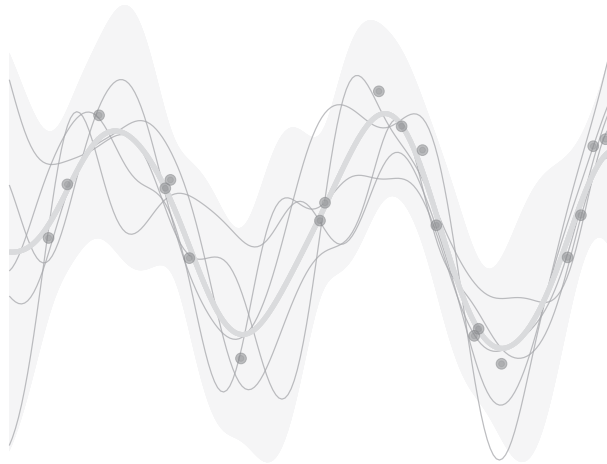
Python

```
import pandas as pd
import statsmodels.formula.api as smf
from statsmodels.gam.api import GLMGam, BSplines

df_happiness = pd.read_csv('https://tinyurl.com/worldhappinessallyears')

model_mixed = smf.mixedlm(
    'happiness_score ~ ???',
    data = df_happiness,
    groups = df_happiness['country']
).fit()

bs = BSplines(
    df_happiness['???'],
    df = 9,  # fiddle with these if you like
    degree = 3
)

gam_happiness = GLMGam.from_formula(
    'happiness_score ~ ???',
    smoother = bs,
    data = df_happiness
).fit()

model_quant = smf.quantreg(
    'happiness_score ~ ???',
    data = df_happiness['???']
).fit(q = ???)
```

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

10

Core Concepts in Machine Learning



Machine learning (ML) is used everywhere, and it allows us to do things that would have been impossible just a couple of decades ago. It is used in everything from self-driving cars, to medical diagnosis, to predicting the next word in your text message. The ubiquity of it is such that machine learning and related adventures like artificial intelligence are used as buzzwords, and it is not always clear what is meant by the one speaking about them. In this chapter we hope to demystify what machine learning is, and how it can be used in your own work.

At its core, machine learning is a form of data analysis with a primary focus on predictive performance. Honestly, that's pretty much it from a practical standpoint. It is *not* a subset of particular types of models, it does not prohibit using statistical models, it doesn't mean that a program spontaneously learns without human involvement[1], it doesn't require any 'machines' outside of a

[1]The description of ML as machines learning 'without being programmed' can be misleading to the newcomer. In fact, many of the most common models used in machine learning are not capable of learning 'on their own' at any level, and require human intervention to provide processed data, specify the model, its parameters, set up the search through that parameter space, analyze the results, update the model, etc. We only very recently, post-2020, have developed models that appear to be able to generalize well to new tasks

laptop, and it doesn't even mean that the model used is particularly complex. Machine learning is a set of tools and a modeling approach that attempts to improve and generalize model performance[2].

This is a *different focus* than statistical modeling approaches that put much more emphasis on interpreting coefficients and uncertainty. But these two approaches can work together. Some implementations of machine learning include models that have their basis in traditional statistics, while others are often sufficiently complex that they are barely interpretable at all. However, even after you conduct your modeling via machine learning, you may still fall back on statistical analysis for further exploration of the results.

Here we will discuss some of the key ideas in machine learning, such as model assessment, loss functions, and cross-validation. Later we'll demonstrate common models used, but if you want to dive in, you can head there now!

> **ML by Any Other Name…**
>
> AI, statistical learning, data mining, predictive analytics, data science, and BI… there are a lot of names used alongside or even interchangeably with machine learning. It's mostly worth noting that using 'machine learning' without context makes it very difficult to know what tools have actually been employed, so you may have to do a bit of digging to find out the details.

10.1 Key Ideas

Here are the key ideas we'll cover in this chapter:

- Machine learning is an approach that prioritizes making accurate predictions using a variety of tools and methods.
- Models used in machine learning are typically more complex and difficult

as if they have learned them without human involvement, but we still can't ignore all the hands-on work that went into the development of those models, which never could have such capabilities otherwise. When you see this 'learning without being programmed', it is an odd way to say that we don't have to guess the parameters ourselves (aside from the first guess). That said, it does feel like the worlds of *The Matrix*, *Star Trek*, and the rest are just around the corner though, doesn't it?

[2]Generalization in statistical analysis is more about generalizing from our sample of data to the population from which it's drawn. In order to do that well or precisely, one needs to meet certain assumptions about the model. In machine learning, generalization is more about how well the model will perform on new data, and it is often referred to as 'out-of-sample' performance. These are not mutually exclusive, but the connotation is different.

to interpret than those used in standard statistical models. However, *any model can be used with ML.*

- There are many performance metrics used in machine learning, and care should be taken to choose the appropriate one for your situation. You can also use multiple performance metrics to evaluate a model.
- Objective functions likewise should be chosen for the situation, and they are often different from the performance metric.
- Regularization is a general approach to penalize complexity in a model, and it is typically used to improve generalization.
- Cross-validation is a technique that helps us choose parameters for our models and compare different models.

10.1.1 Why this matters

Machine learning applications help define the modern world and how we interact with it. There are few aspects of modern society that have not been touched by it in some way. With a basic understanding of the core ideas behind machine learning, you will better understand the models and techniques that are used in ML applications and be able to apply them to your own work. You'll also be able to understand the limitations of these models, and not think of machine learning as 'magic'.

10.1.2 Helpful context

To dive into applying machine learning models, you really only need a decent grasp of linear models as applied to regression and classification problems (Chapter 3 and Chapter 8). It would also be good to have an idea behind how they are estimated (Chapter 6), as the same basic logic serves as a starting point here.

10.2 Objective Functions

We've implemented a variety of objective functions in other chapters, such as mean squared error for numeric targets and log loss for binary targets (Chapter 6). The objective function is what we used to estimate model parameters, but it's not necessarily the same as the performance metric we ultimately use to select a model. For example, we may use log loss as the objective function, but then use accuracy as the performance metric. In that setting, the log loss provides a 'smooth' objective function to search the parameter space over, while accuracy is a straightforward and more interpretable metric for stakeholders. In this case, the objective function is used to optimize the model, while the performance metric is used to evaluate the model. In some cases,

the objective function and performance metric are the same (e.g., (R)MSE), and even if not, they might have selected the same 'best' model, but this is not always the case. The following breakdown shows some commonly used objective functions in machine learning for regression and classification tasks.

Regression

- **Mean Squared Error (MSE)**: Average of the squared differences between the predicted and actual values.
- **Mean Absolute Error (MAE)**: Average of the absolute differences between the predicted and actual values.
- **Huber Loss**: A robust approach that is less sensitive to outliers than MSE.
- **Log Likelihood**: Maximizes the likelihood of the data given the model parameters.

Classification

- **Binary Cross-Entropy / Log-Likelihood (Loss)**: Used for binary classification problems. Same as log-likelihood.
- **Categorical Cross-Entropy**: Binary approach extended to multiclass classification problems.

Of course, there are many more objective functions than these, and they can be quite complex. The choice of objective function will depend on the type of problem you are trying to solve, and the characteristics of your data.

10.3 Performance Metrics

When discussing how to understand our model (Section 4.2), we noted there are many performance metrics used in machine learning. Care should be taken to choose the appropriate one for your situation. Usually we have a standard set we might use for the type of predictive problem. For example, for numeric targets, we typically are interested in (R)MSE and MAE. For classification problems, many metrics are based on the **confusion matrix**, which is a table of the predicted classes versus the observed classes. From that we can calculate things like accuracy, precision, recall, AUROC, etc. (see Table 4.1).

As an example, and as a reason to get our first taste of machine learning, let's get some metrics for a movie review model. We'll do this for both numeric and classification targets to demonstrate the different types of metrics we can obtain. As we start our journey into machine learning, we'll show Python code

first, as it's the dominant tool for ML. Here we'll model the target in both numeric and binary form with corresponding metrics.

Python

In Python, we can use the `sklearn.metrics` module to get a variety of metrics.

```
from sklearn.metrics import (
    mean_squared_error, root_mean_squared_error,
    mean_absolute_error, r2_score,
    accuracy_score, precision_score, recall_score,
    roc_auc_score, roc_curve, auc, confusion_matrix
)

from sklearn.linear_model import LinearRegression, LogisticRegression

import pandas as pd

df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

X = df_reviews[
    [
        'word_count',
        'age',
        'review_year',
        'release_year',
        'length_minutes',
        'children_in_home',
        'total_reviews',
    ]
]

y = df_reviews['rating']
y_class = df_reviews['rating_good']

model_lin_reg = LinearRegression().fit(X, y)

# note that sklearn uses regularization by default for logistic regression
model_log_reg = LogisticRegression().fit(X, y_class)

y_pred_linreg = model_lin_reg.predict(X)
y_pred_logreg = model_log_reg.predict(X)

# regression metrics
rmse = root_mean_squared_error(y, y_pred_linreg)
```

```
mae = mean_absolute_error(y, y_pred_linreg)
r2 = r2_score(y, y_pred_linreg)

# classification metrics
accuracy = accuracy_score(y_class, y_pred_logreg)
precision = precision_score(y_class, y_pred_logreg)
recall = recall_score(y_class, y_pred_logreg)
```

R

In R, we can use mlr3measures, which has a variety of metrics.

```
library(mlr3measures)

# convert rating_good to factor for some metric inputs
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata') |>
    mutate(rating_good = factor(rating_good, labels = c('bad', 'good')))

model_lin_reg = lm(
    rating ~
        word_count
        + age
        + review_year
        + release_year
        + length_minutes
        + children_in_home
        + total_reviews,
    data = df_reviews
)

model_log_reg = glm(
    rating_good ~
        word_count
        + age
        + review_year
        + release_year
        + length_minutes
        + children_in_home
        + total_reviews,
    data = df_reviews,
    family = binomial(link = 'logit')
)

y_pred_linreg = predict(model_lin_reg)
```

```
y_pred_logreg = predict(model_log_reg, type = 'response')
y_pred_logreg = factor(ifelse(y_pred_logreg > .5, 'good', 'bad'))


# regression metrics
rmse_val = rmse(df_reviews$rating, y_pred_linreg)
mae_val  = mae(df_reviews$rating, y_pred_linreg)
r2_val   = rsq(df_reviews$rating, y_pred_linreg)

# classification metrics
accuracy  = acc(df_reviews$rating_good, y_pred_logreg)
precision = precision(df_reviews$rating_good, y_pred_logreg, positive = 'good')
recall    = recall(df_reviews$rating_good, y_pred_logreg, positive = 'good')
```

We put them all together in the following table. Now we know how to get them, and it was easy! But as we'll see later, there is a lot more to think about before we use these for model assessment.

Table 10.1: Example Metrics for Linear and Logistic Regression Models

Metric	Value
Linear Regression	
RMSE	0.52
MAE	0.41
R-squared	0.32
Logistic Regression	
Accuracy	0.71
Precision	0.72
Recall	0.79

10.4 Generalization

So getting metrics is easy enough, but how will we use them? One of the key differences separating ML from traditional statistical modeling approaches is the assessment of performance on unseen or future data, a concept commonly referred to as **generalization**. The basic idea is that we want to build a model that will perform well on *new* data, and not just the data we used to train the model. This is because ultimately data is ever evolving, and when we are concerned with making predictions, we don't want to be beholden to a

particular set of data that we just happened to have at a particular time and context.

But how do we do this? As a starting point, we can simply split (often called **partitioning**) our data into two sets, a **training set** and a **test set**, or, **holdout set**. The test set is typically a smaller subset, say 25% of the original data, but this amount is arbitrary and will reflect the data situation. We **fit** or **train** the model on the training set, and then use the model to make predictions on, or **score**, the test set. This general approach is also known as the **holdout method**.

Consider a simple linear regression. We can fit the linear regression model on the training set, which provides us coefficients, etc. We can then use that model result to predict on the test set, and then compare the predictions to the observed target values in the test set. We can calculate metrics on both the training and test sets. Here we demonstrate this with our simple linear model.

Python

```
from sklearn.model_selection import train_test_split

X = df_reviews[[
        'word_count',
        'age',
        'review_year',
        'release_year',
        'length_minutes',
        'children_in_home',
        'total_reviews',
    ]]

y = df_reviews['rating']

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=123
)

model_linreg_train = LinearRegression().fit(X_train, y_train)

# get predictions
y_pred_train = model_linreg_train.predict(X_train)
```

```
y_pred_test = model_linreg_train.predict(X_test)

# get RMSE
rmse_train = root_mean_squared_error(y_train, y_pred_train)
rmse_test = root_mean_squared_error(y_test, y_pred_test)

pd.DataFrame(
    dict(
        prediction = ['Train', 'Test'],
        rmse = [rmse_train, rmse_test]
    )
).round(3)
```

R

```
# create a train and test set
library(rsample)

set.seed(123)

split = initial_split(df_reviews, prop = .75)

X_train = training(split)
X_test  = testing(split)

model_linreg_train = lm(
    rating ~
        word_count
        + age
        + review_year
        + release_year
        + length_minutes
        + children_in_home
        + total_reviews,
    data = X_train
)

# get predictions
y_train_pred = predict(model_linreg_train, newdata = X_train)
y_test_pred  = predict(model_linreg_train, newdata = X_test)

# get RMSE
rmse_train = rmse(X_train$rating, y_train_pred)
```

```
rmse_test  = rmse(X_test$rating, y_test_pred)

tibble(
    prediction = c('Train', 'Test'),
    rmse = c(rmse_train, rmse_test)
)
```

Table 10.2: RMSE for Linear Regression Model on Train and Test Sets

prediction	rmse
Train	0.515
Test	0.530

So there you have it – you just did some machine learning! And now we have a model that we can use to predict with any new data that comes along with ease. But as we'll soon see, there are limitations to doing things this simply. But conceptually this is an important idea, and one we will continue to return to.

> **i** Split Results
>
> A reminder: the results of the split can vary depending on the random seed used, and whether you are using R or Python. So your results may look slightly different from the presented tables.

10.4.1 Using metrics for model evaluation and selection

As we've seen elsewhere, there are many performance metrics to choose from to assess model performance, and the choice of metric depends on the type of problem (Section 4.2). It also turns out that assessing the metric on the data we used to train the model does not give us the best assessment of that metric. This is because the model will do better on the data it was trained on than on data it wasn't trained on, and we can generally always improve that metric in training by making the model more complex. However, in many modeling situations, this complexity comes at the expense of generalization. So what we really want to ultimately say about our model will regard performance on the test set with our chosen metric, and not the data we used to train the model. At that point, we can also compare multiple models to one another given their performance on the test set, and select the one that performs best.

In the previous section you can compare our results on the tests vs. training set. Metrics are notably better on the training set on average, and that's what we see here. But since we should be more interested in how well the model will do on new data, we should focus on the test set result.

10.4.2 Understanding test error and generalization

This part gets into the weeds a bit. If you are not so inclined, skip to the summary of this section.

In the following discussion, you can think of a standard linear model scenario, for example, with squared-error loss function, and a dataset where we split some of the observations in a random fashion into a training set, for initial model fitting, and a test set, which will be kept separate and independent, and used to measure generalization performance. We note **training error** as the average loss over all the training sets we could create in this process of random splitting. The **test error** is the average prediction error obtained when a model fitted on the training data is used to make predictions on the test data.

Generalization in the classical regime

So what result should we expect in this scenario? Let's look at the following visualization, inspired by Hastie, Tibshirani, and Friedman (2017).

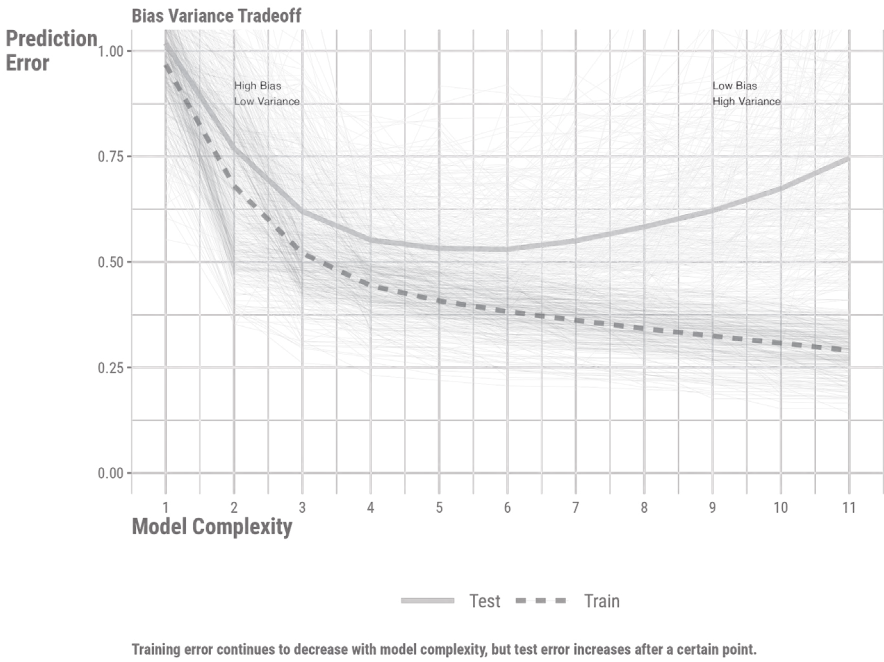


Figure 10.1: Bias-variance tradeoff.

Prediction error on the test set is a function of several components, and two of these are **bias** and **variance**.

A key idea is that as the model complexity increases, we potentially capture more of the data variability. This reduces **bias**, which is the difference in our average prediction and the *true* model prediction. But this only works for training error, where eventually our model can potentially fit the training data perfectly!

For test error though, as the model complexity increases, the bias decreases, but the **variance** eventually increases. This variance reflects how much our prediction changes with different data. If our model gets too cozy with the training data, it will do poorly when we try to generalize beyond it, and this will be reflected in increased variance. This is traditionally known as the **bias-variance tradeoff** – we can reduce one source of error in the test set at the expense of the other, but not both at the same time indefinitely. In other words, we can reduce bias by increasing model complexity, but this will eventually increase variance in our test predictions. As seen in Figure 10.1[3], we can reduce variance by reducing model complexity, but this will increase bias. One additional thing to note is that even if we had the 'true' model given the features specified correctly, for the vast majority of cases there would still be prediction error due to the random data generating process (**noise**). This can potentially be reduced using additional valid features, getting better measurements, etc., but it will still be there to some extent in practice, and so will limit test set performance.

The ultimate goal is to find the sweet spot. We want a model that's complex enough to capture the data, but not so complex that it overfits the training data.

Generalization in deep learning

It turns out that with lots of data and very complex models, or maybe even in most settings, the 'classical' understanding just described doesn't hold up. In fact, it is possible to get a model that fits the training data perfectly, and yet ultimately still generalizes well to new data!

This phenomenon is encapsulated in the notion of **double descent**. The idea is that, with very complex and flexible models such as those employed with deep learning, we can get to the point of interpolating the data exactly. But as we continue to increase the complexity of the model, we actually start to generalize better again, as the model continues to explore potential options for fitting the data. This is a fascinating and somewhat counterintuitive result, and visually this displays as a 'double descent' in terms of test error. We see an initial decrease in test error as the model gets better in general. After a while, it begins to rise as we would expect in the classical regime (Figure 10.1). Eventually it peaks at the point where we have as many parameters as data

[3]For those viewing the pdf, we recommend turning off the 'enhance thin lines' option in 'Preferences' for this and the next plot.

points. Beyond that however, as we get even more complex with our model, we can possibly see a decrease in test error again[4]. Crazy!

We can demonstrate this on the classic `mtcars` dataset[5], which has only 32 observations! We repeatedly trained a model to predict miles per gallon on only 10 of those observations, and assess test error on the rest. The model we used is a form of ridge regression[6], but we implemented splines for the car's weight, horsepower, and displacement, i.e., we GAMed it up (Section 9.4). We trained increasingly complex models, and in what follows we visualize the error as a function of model complexity as we did previously.

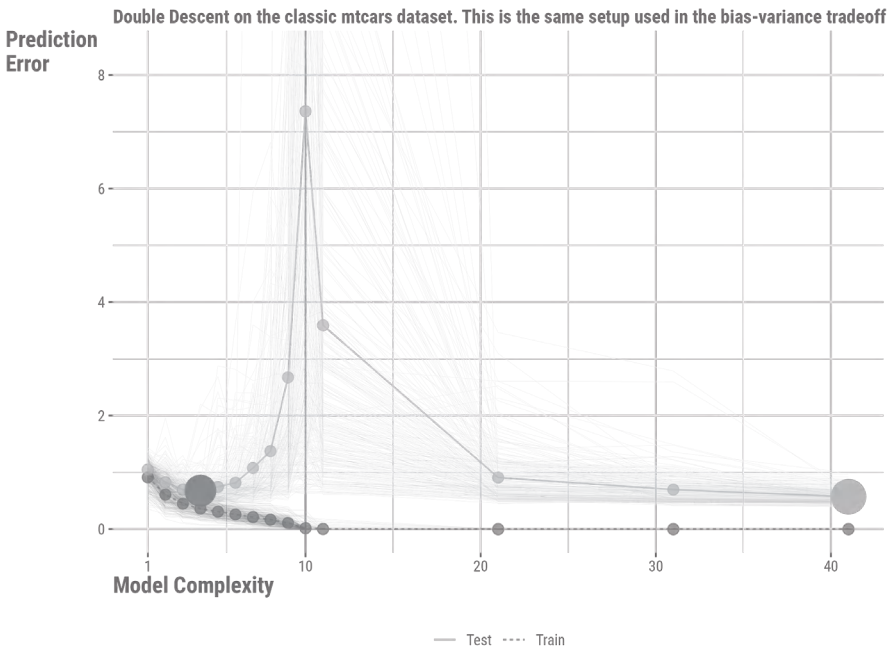


Figure 10.2: Double descent on the classic mtcars dataset.

On the left part of the Figure 10.2, we see that the test error dips as we get a better model. Our best test error is noted by the large dot on the left. Eventually though, the test error rises as expected, even as training error gets better. Test error eventually hits a peak when the number of parameters equals

[4]A similar phenomenon is found in the idea of **grokking** in deep learning. In this case, even after seemingly doing as well as the model can on training and validation, the model 'spontaneously' starts to improve on validation with continued iterations. See Power et al. (2022) for more on this.

[5]If not familiar, the `mtcars` object is a dataset that comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-74 models).

[6]It's actually called *ridgeless* regression.

the number of training observations. But then we keep adding to the model complexity, and the test error starts to decrease again! By the end we have essentially perfect training prediction, and our test error is as good as it was with the simpler models (large dot on the right). This is the double descent phenomenon with one of the simplest datasets around. Cool!

Generalization summary

The take-home point is this: our primary concern is generalization error. We can reduce this error by increasing model complexity, but this may eventually cause test error to increase. However, with enough data and model complexity, we can get to the point where we can fit the training data perfectly and yet still generalize well to new data. In many standard, or at least smaller, data and model settings, you can maybe assume the classical regime holds. But when employing deep learning with massive data and billions of parameters, you can worry less about the model's complexity. But no matter what, we should use tools to help make our model work better, and we prefer smaller and simpler models that can do as well as more complex ones, even if those 'smaller' models are still billions of parameters!

10.5 Regularization

We now are very aware that a key aspect of the machine learning approach is having our model to work well with new data. One way to improve generalization is through the use of **regularization**, which is a general approach to penalize complexity in a model, and is typically used to prevent **overfitting**. Overfitting occurs when a model fits the training data very well but does not generalize well to new data. This usually happens when the model is too complex and starts fitting to random noise in the training data. We can also have the opposite problem, where the model is too simple to capture the patterns in the data, and this is known as **underfitting**[7].

In the following demonstration, the first plot shows results from a model that is probably too complex for the data setting. The curve is very wiggly as it tries as much of the data as possible, and it is an example of overfitting. The second plot shows a straight line fit as we'd get from linear regression. It's too

[7]Underfitting is a notable problem in many academic disciplines, where the models are often too simple to capture the complexity of the underlying process. Typically, the model employed assumes linear relationships without any interactions, and the true data generating process may be anything but. These models are chosen for their simplicity and interpretability, rather than how well they can explain the phenomenon in question. However, one could make the argument that 'understanding' an unrealistic result is not very useful either, and that the goal should be to understand the true process however we can, and not just choose a model that's convenient.

simple for the underlying feature-target relationship, and it is an example of underfitting. The third plot shows a model that is a better fit to the data, and it is an example of a model that is complex enough to capture the nonlinear aspect of the data, but not so complex that it capitalizes on a lot of noise.

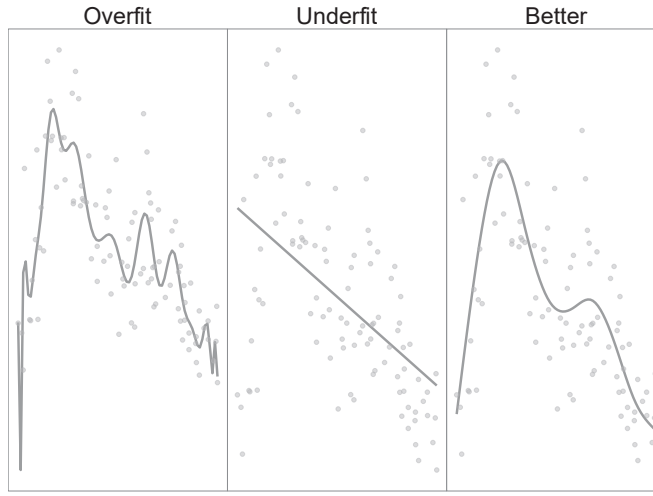


Figure 10.3: Overfitting and underfitting.

As a demonstration, let's examine generalization performance in this type of setting[8] with the following table that represents the test set RMSE. We see that the overfit model does best on training data, but relatively very poorly on test, with nearly a 20% increase in the RMSE value. The underfit model doesn't change as much in test performance because it was poor to begin with and is the worst performer for both. Our 'better' model wasn't best on training but was best on the test set.

Table 10.3: RMSE for Each Model on Test Data

Data	Over	Under	Better
Train	1.97	3.05	2.18
Test	2.34	3.24	2.19
% inc. RMSE	19.09	6.08	0.56

A fairly simple example of regularization can be seen with a ridge regression model (Section 6.8), where we add a penalty term to the objective function. The penalty is a function of the size of the coefficients and helps keep the

[8]The data is based on a simulation (using `mgcv::gamSim`), with training sample of 200 and scale of 1, so the test data is just more simulated data points.

model from getting too complex. It is also known as **L2 regularization** due to squaring the coefficients (formally, the L_2 norm). Another type is the **L1 penalty**, used in the 'lasso' model, which is based on the absolute values of the coefficients (L_1 norm). Yet another common approach combines the two, called **elastic net**. There, we use both L1 and L2 penalties with different weights, and use cross-validation to find the best balance. L1 and/or L2 penalties are applied in many other models such as gradient boosting, neural networks, and others, and they are a key aspect of machine learning.

Regularization is used in many modeling scenarios. Here is a quick rundown of some examples.

- GAMs use penalized regression for estimation of the coefficients for the basis functions (typically with L2). This keeps the 'wiggly' part of the GAM from getting too wiggly, as in the overfit model in Figure 10.3. This shrinks the feature-target relationship toward a linear one.
- Similarly, the variance estimate of a random effect in mixed models, e.g., for the intercept or slope, is inversely related to an L2 penalty on the effect estimates for that group effect. The more penalization applied, the less random effect variance, and the more the random effect is shrunk toward the overall mean[9].
- Still another form of regularization occurs in the form of priors in Bayesian models. There we use priors to control the influence of the data on the final model. A small variance on the prior shrinks the model toward the prior mean. If large, there is little influence of the prior on the posterior. In regression models, there is correspondence between ridge regression and using a normal distribution prior for the coefficients in Bayesian regression, where the L2 penalty is related to the variance of that prior. Even in deep learning, there is usually a 'Bayesian interpretation' of the regularization approaches employed.
- As a final example of regularization, **dropout** is a technique used in deep learning to prevent overfitting. Feel free to return to this discussion after seeing neural networks in action in the next chapter (Section 11.7), as that will provide the appropriate context. But the gist is that dropout works by randomly removing some of the nodes in intervening/hidden layers in the network during training. This tends to force the network to learn more robust features, allowing for better generalization[10].

[9] One more reason to prefer a random effects approach over so-called fixed effects models, as the latter are not penalized at all and as a result are more prone to overfitting.

[10] **Batch normalization**, another common layer application in deep learning, also has a regularizing effect by adding a bit of noise to the network. This can help with generalization and also helps with training stability.

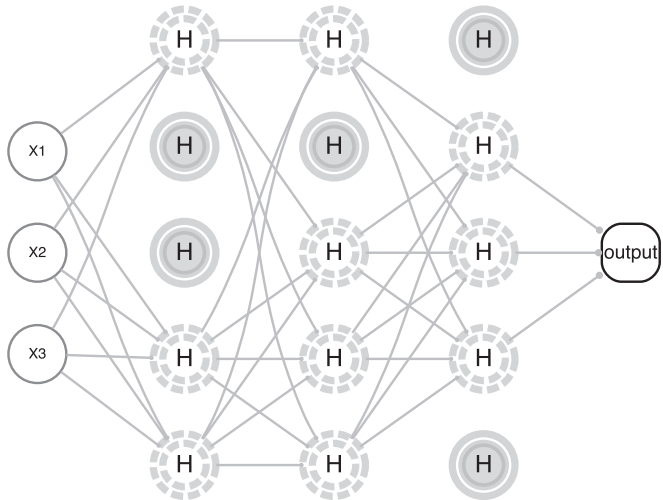


Figure 10.4: Neural network with dropout.

In the end, regularization comes in many forms across the modeling landscape, and it is a key aspect of machine learning and traditional statistical modeling alike. The primary goal is to decrease model complexity in the hopes of increasing our ability to generalize the selected model to new data scenarios.

> **Regularization with Large Data**
>
> For the linear model and related models for typical tabular data, a very large dataset can often lessen the need for regularization. This is because the model can learn the patterns in the data without overfitting, and the penalty ultimately is overwhelmed by the other parts of the objective function. However, regularization is still useful in many cases, and can help with model stability and speed of convergence.

10.6 Cross-validation

We've talked a lot about generalization, so now let's think about some ways to go about a general process of selecting parameters for a model and assessing performance in a way that helps us generalize to new data better.

We previously used a simple approach where we split the data into training and test sets. We then fit the model on the training set and subsequently assessed performance on the test set. This is fine, but the test set error, or any

other metric, has uncertainty. It could be slightly different with any random training-test split we came up with.

We'd also like to get better model assessment when searching the hyperparameter space, because we have no way of guessing the value beforehand, and we'll need to try out different ones. An example would be the penalty parameter in lasso regression. In this case, we need to figure out the best parameters *before* assessing a final model's performance.

One way to do this is to split the training data into different partitions, which we now call **validation sets**. We fit the model on the training set, and then assess performance on the validation set(s). We then repeat this process for many different splits of the data into training and validation sets, and average the results. This is known as **K-fold cross-validation**. It's important to note that we still want a test set to be held out that is in no way used during the training process. The validation sets are used to help us choose the best model based on some metric, and the test set is used to assess the chosen model's performance.

Here is a visualization of how 3-fold cross-validation works. We split the data such that two-thirds of it will be used for training, and one-third for validation. We then do this for a total of three times, so that the validation set represents a different part of the data each time. Ultimately, all observations are used for both training and validation by the end of the process. We then average the results of any metric across the validation sets. Note that in each case here, there is no overlap of data between the training and validation sets.

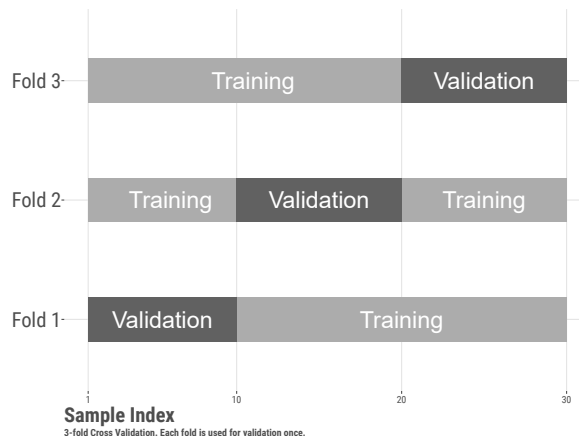


Figure 10.5: Three-fold cross-validation.

The idea is that we are trying to get a better estimate of the error by averaging over many different validation sets. The number of folds, or splits, is denoted

by K. The value of K can be any number, but typically is 10 or less. The larger the value of K, the more accurate the estimate of the metric, but the more computationally expensive it is, and in application, you generally don't need much to get a good estimate. With smaller datasets, one can even employ a **leave-one-out** approach, where K is equal to the number of observations in the data. In this case, a validation prediction is made for each observation, which we can then accumulate to calculate the metric of choice.

So cross-validation provides a better measure of the metric we use to choose our model. When comparing a model with different parameter settings, we can look at the (average) metric each has from the validation process, and select the model parameter set that has the best metric value. This process is typically known as **model selection**. This works for choosing a model across different sets of hyperparameter settings, for example, with different penalty parameters for regularized regression. But it can also aid in choosing a model from a set of different model types, for example, standard linear model approach vs. boosting. In that case we apply the cross-validation approach for each model, and the 'winner' is the one with the best average metric value on the test set.

Now how might we go about this for modeling purposes? Very easily with modern packages. In the following we demonstrate cross-validation with a logistic regression model.

Python

```
from sklearn.linear_model import LogisticRegressionCV

X = df_reviews.filter(regex='_sc$') # grab the standardized features
y = df_reviews['rating_good']

# Cs is the (inverse) penalty parameter
model_logistic_l2 = LogisticRegressionCV(
    penalty='l2',      # penalty type
    Cs=[1],            # penalty parameter value
    cv=5,
    max_iter=1000,
    verbose=False
).fit(X, y)

# model_logistic_l2.scores_  # show the accuracy score for each fold

# print the average accuracy score
model_logistic_l2.scores_[1].mean()
```

```
0.671
```

R

For R, we prefer mlr3 for our machine learning demonstrations, as we feel it is more like sklearn in spirit, and offers computational advantages when you need it most[11]. The tidymodels ecosystem is also a good option.

```
library(mlr3)
library(mlr3learners)

X = df_reviews |>
    select(matches('_sc|good'))  # grab the standardized features/target

# Define task
task_lr_l2 = TaskClassif$new('movie_reviews', X, target = 'rating_good')

# Define learner (alpha = 0 is ridge/l2 regression)
learner_lr_l2 = lrn('classif.cv_glmnet', alpha = 0, predict_type = 'response')

# set the penalty parameter to some value
learner_lr_l2$param_set$values$lambda = c(.1, .2)

# Define resampling strategy
model_logistic_l2 = resample(
    task = task_lr_l2,
    learner = learner_lr_l2,
    resampling = rsmp('cv', folds = 5),
    store_models = TRUE
)

# show the accuracy score for each fold
# model_logistic_l2$score(msr('classif.acc'))

model_logistic_l2$aggregate(msr('classif.acc'))
```

```
classif.acc
      0.671
```

From the five validation sets, we end up with five separate accuracy values, one for each fold. Our final assessment of the model's accuracy is the average of these five values, which is shown. This is a better estimate of the model's

[11] In this particular case, we're using glmnet for the logistic regression. To say that it is a confusing implementation of a model function compared to standard R approaches is an understatement. While it's hard to argue with the author of the lasso itself (who is an author of the package), it's not the most user-friendly package in the world and has confused most who've used it. Our example does actually set the penalty parameter, but it's not the most straightforward thing to do.

accuracy than if we had just used a single test of the model, and in the end, it is still based on the entire training data.

10.6.1 Methods of cross-validation

There are different approaches we can take for cross-validation that we may need for different data scenarios. Here are some of the more common ones.

- **Shuffled**: Shuffling prior to splitting can help avoid data ordering having undue effects.
- **Grouped/Stratified**: In cases where we want to account for the grouping of the data, e.g., for data with a hierarchical structure.
  - Grouped: We may want groups to appear in training *or* test, but not both. This allows us to generalize to new groups.
  - Stratified: We may want to ensure group proportions are consistent across training and test sets. This is especially useful in unbalanced target settings to ensure all class labels are present in training and test.
- **Time-based**: For time series data, where we only want to assess error on future values.
- **Combinations.: For example, grouped and time-based

Here are images based on the scikit-learn library documentation (Pedregosa et al. (2011)) depicting some different cross-validation approaches. In general, the type we use will be based on our data needs[12].

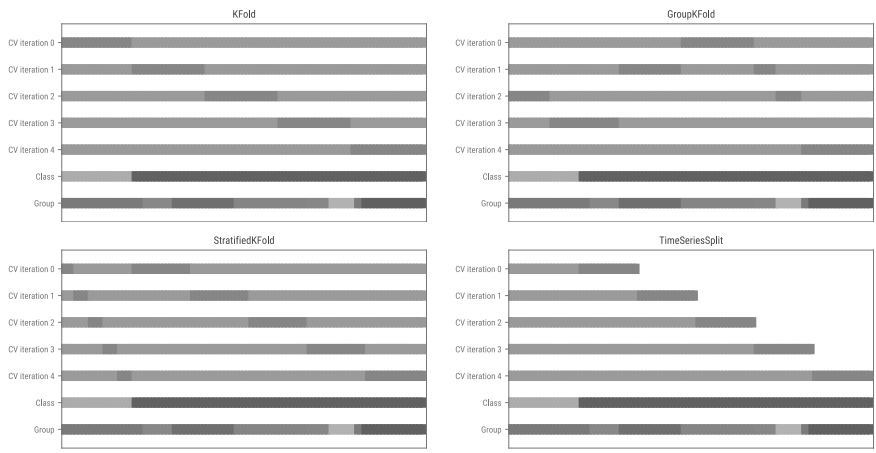


Figure 10.6: Comparison of cross-validation strategies.

[12]These images likely don't work well for black-white printing, but there are multiple groups and using a bunch of different patterns would make the figure worse. We invite you to examine the pdf or web version.

> **i** Stratified Cross-Validation
>
> It's generally always useful to use a stratified approach to cross-validation, especially with classification problems, as it helps ensure a similar balance of the target classes across training and test sets. You can also use this with numeric targets, enabling you to have a similar distribution of the target across training and test sets.

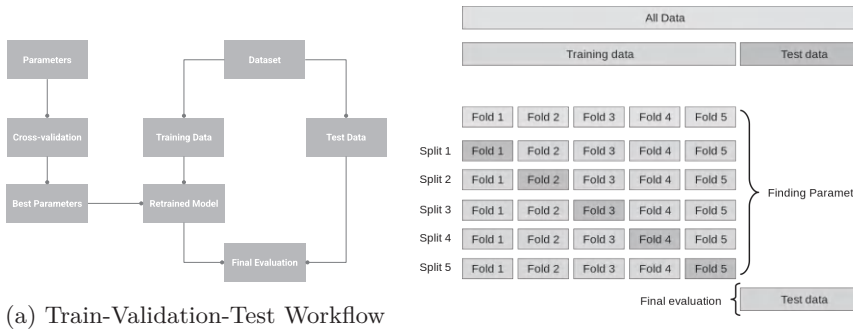
10.7 Tuning

One problem with the previous ridge logistic model we just used is that we set the penalty parameter to a fixed value. We can do better by searching over a range of values instead, and picking a 'best' value based on which model performs best with a specific penalty value. This is generally known as **hyperparameter tuning**, or simply **tuning**. It is one aspect of machine learning that distinguishes it from traditional statistical modeling, where we usually don't have hyperparameters to consider. For this example with penalized regression, we can tune our model with k-fold cross-validation to assess the error for each proposed value of the penalty parameter. We then select the value of the penalty parameter for which the associated model gives the lowest average error. This is a form of model selection.

Another potential point of concern is that we are using the same data to both select the model and assess its performance. This is a type of **data leakage**, and it may result in an overly optimistic assessment of performance. One solution is to do as we've discussed before, which is to split the data into three parts: training, validation, and test. We use the training set(s) to fit the models, assess their performance on the validation set(s), and select the best model. Then we use the test set to assess the best model's performance. So the validation approach is used to select the model, and the test set is used to assess that model's performance. The following visualizations from the scikit-learn documentation illustrate the process.

> **i** Nested Cross-Validation
>
> As the performance on test is not without uncertainty, we can actually nest the entire process within a validation approach, where we have an inner loop of k-fold cross-validation and an outer loop to assess the model's performance on multiple hold-out sets. This is known as **nested cross-validation**. It is a more computationally expensive approach, and

generally would require more data, but it would result in a more robust assessment of performance.



(a) Train-Validation-Test Workflow

(b) 5-fold Cross-Validation

Figure 10.7: A tuning workflow.

10.7.1 A tuning example

While this may start to sound complicated, it doesn't have to be, as tools are available to make our generalization journey a lot easier. In the following we demonstrate the approach with the same ridge logistic regression model as before. The specific type of parameter search we'll use is called a **grid search**, where we explicitly step through potential values of the penalty parameter, fitting a model with the selected value through cross-validation. While we only look at one parameter here, for a given modeling approach we could construct a 'grid' of sets of parameter values to search over as well[13]. For each hyperparameter value, we are interested in the average accuracy score across the validation folds to assess the best performance. The final model can then be assessed on the test set[14].

[13]We can use `expand.grid` or `crossing` functions in R, or pandas' `expand_grid` to easily construct these values to iterate over. `scikit-learn`'s `GridSearchCV` function does this for us when we provide the dictionary of values for each parameter.

[14]If you're comparing the Python vs. R approaches, while the name explicitly denotes no penalty, the scikit-learn model by default uses ridge regression. In R we set the value alpha to enforce the ridge penalty, since glmnet by default uses the elastic net, a mixture of lasso and ridge, and we only want the ridge approach. Also, scikit-learn uses the inverse of the penalty parameter, while mlr3 uses the penalty parameter directly. And obviously, no one will agree on what we should name the value, and we have no idea where 'C' comes from, maybe 'complexity'(?), though we have seen λ used in many statistical publications.

Python

With a set of penalty values to explore, we again use the `LogisticRegression` function in `sklearn` to perform k-fold cross-validation to select the best performing penalty parameter. We then apply the chosen model to the test set and calculate accuracy.

```
from sklearn.model_selection import GridSearchCV

# split the dataset from the previous example into
# training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=42
)

# define the parameter values for GridSearchCV
param_grid = {
    'C': [0.1, 1, 2, 5, 10, 20],
}

# perform k-fold cross-validation to select the best penalty parameter
# Note that LogisticRegression by default is ridge regression for scikit-learn
model_logistic_grid = GridSearchCV(
    LogisticRegression(),
    param_grid=param_grid,
    cv=5,
    scoring='accuracy'
).fit(X_train, y_train)

# if you want to inspect the results
best_model = model_logistic_grid.best_estimator_
best_param = model_logistic_grid.best_params_['C']

# apply the best model to the test set and calculate accuracy
acc_train = model_logistic_grid.score(X_train, y_train)
acc_test  = model_logistic_grid.score(X_test, y_test)
```

```
Best C: 2
Accuracy on train set: 0.661
Accuracy on test set: 0.692
```

R

We use the `auto_tuner` function to perform k-fold cross-validation to select the best penalty parameter (`lambda`). We set the mixing parameter (`alpha`) to zero, because glmnet is by default elastic net, or a mixture of ridge and lasso. This ensures we are only using a ridge (L2) penalty. See the glmnet vignette for details.

```
# Load necessary libraries
library(mlr3learners)  # for a choice of modeling approaches
library(mlr3tuning)    # for tuning

X = df_reviews |>
    mutate(rating_good = as.factor(rating_good)) |>
    select(matches('sc|rating_good')) |>
    as.data.table()

# Define task
task = TaskClassif$new(
    'movie_reviews',
    X,
    target = 'rating_good',
    positive = 'good'
)

# split the dataset into training and test sets
splits = partition(task, ratio = 0.75)

# Define learner
learner = lrn('classif.glmnet', alpha = 0, predict_type = 'response')

# Define resampling strategy
cv_k5 = rsmp('cv', folds = 5)

# Define measure
measure = msr('classif.acc')

# Define parameter space
param_set = ParamSet$new(list(
    lambda = p_dbl(lower = 1e-3, upper = 1)
))

# Define tuner
model_logistic_grid = auto_tuner(
    learner = learner,
```

```
    resampling = cv_k5,
    measure = measure,
    search_space = param_set,
    tuner = tnr('grid_search', resolution = 10),
    terminator = trm('evals', n_evals = 10)
)

# Tune hyperparameters
model_logistic_grid$train(task, row_ids = splits$train)

# Get best hyperparameters
best_param = model_logistic_grid$model$learner$param_set$values

# Use the best model to predict and get metrics
pred_train = model_logistic_grid$predict(task, row_ids = splits$train)
pred_test = model_logistic_grid$predict(task, row_ids = splits$test)
acc_train = pred_train$score(measure)
acc_test  = pred_test$score(measure)
```

```
Best lambda: 0.556
Accuracy on train set: 0.674666666666667
Accuracy on test set: 0.684
```

So there you have it. We searched the parameter space, chose the best hyperparameter via k-fold cross-validation, and got an assessment of generalization error. Neat!

10.7.2 Parameter spaces

In the previous example, we used a grid search to search over a range of values for the penalty parameter. It is a quick and easy way to get started, but generally we want something that can search a better space of parameter values rather than a limited grid. It can also be computationally expensive with many hyperparameters, as we might have with boosting methods. We can do better by using more efficient approaches. For example, we can use a **randomized search**, where we randomly sample from the parameter space. This is generally faster than a grid search, and can be just as effective. Other methods are available that better explore the space and do so more efficiently.

> **i** Tuning and Overfitting
>
> A word of caution. Cross-validation is not a perfect solution, and you can still overfit the model selection process. This is especially true when you have a large number of parameters and other model aspects to search

over. It may help to use more sophisticated approaches to search the parameter space, such as **Bayesian optimization**, **hyperband**, or **genetic algorithms**, along with the nested cross-validation mentioned before (e.g., Cawley and Talbot (2010)).

10.8 Pipelines

For **production-level** work, or just for **reproducibility**, it is useful to create a **pipeline** for your modeling work. In essence, a pipeline is just a series of steps that are performed in a particular order. For example, we might want to perform the following steps:

- Impute missing values
- Transform features
- Create new features
- Split the data into training and test sets
- Fit the model on the training set with cross-validation
- Assess the model's performance on the test set
- Compare the model with others put through the same process
- Save the 'best' model
- Use the model for prediction on future data, sometimes called **scoring**
- Redo the whole thing on a regular basis

We can create a pipeline that performs all of these steps in sequence. This is useful for a number of reasons:

1. Using a pipeline makes it far easier to reproduce the results as needed. Running the pipeline means you are running each of the same exact steps in the same exact order.
2. It is relatively easy to change the steps in the pipeline. For example, we might want to try a different imputation method, or add a new model. The pipeline is already built to handle these steps, so any modification is straightforward and more easily applied.
3. It is relatively easy to use the pipeline with new data. We can just start with the new data, and it will perform all of the steps in sequence.
4. Having a pipeline facilitates model comparison, as we can ensure that the models are receiving the same data process.
5. We can save the pipeline for later use. We just save the pipeline as a file, and then load it later when we want to use it again.

While pipelines are useful for any modeling work, they are especially useful for machine learning, where we often have many steps to perform, and where we are often trying to compare many different models. You don't have to have a formal pipeline, but it is a good practice to have a script that performs all of the steps in sequence, and that can be run at any time to reproduce the results. Formal pipeline tools make it easier to manage the process, and the following demonstrates how that might look.

Python

Here is an example of a pipeline in Python. We use the `make_pipeline` function from scikit-learn. This function takes a series of steps as arguments, and then performs them in sequence. We can then use the pipeline to fit the model, assess its performance, and save it for later use.

```
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler

# create pipeline
logistic_cv_pipeline = make_pipeline(
    SimpleImputer(strategy='mean'),
    StandardScaler(),
    LogisticRegressionCV(penalty='l2', Cs=[1], cv=5, max_iter=1000),
)

# Fit the pipeline
logistic_cv_pipeline.fit(X_train, y_train)

# Assess the pipeline on test
y_pred = logistic_cv_pipeline.predict(X_test)
accuracy_score(y_test, y_pred)

# Save the pipeline
# from joblib import dump, load
# dump(logistic_cv_pipeline, 'logistic_cv_pipeline.joblib')
```

```
0.692
```

R

With R, mlr3 works in a similar fashion to scikit-learn. We create a pipeline with the `po`, or pipe operator function, which takes a series of steps as arguments, and then performs them in sequence.

```
# Using task/splits/resampling from tuning section
library(mlr3pipelines)

# Define pipeline
logistic_cv_pipeline =  po('imputemean') %>>%
    po('scale') %>>%
    po(
        'learner',
        lrn('classif.cv_glmnet', predict_type = 'response'),
        alpha = to_tune(1e-04, 1e-1, logscale = TRUE), # mixing parameter
        lambda = c(1e-3, 1e-2, 1e-1, 1)   # penalty
    )

model_logistic_cv_pipeline = AutoTuner$new(
    learner = logistic_cv_pipeline,
    resampling = cv_k5, # defined earlier 5-fold cv
    measure = measure,
    tuner = tnr('grid_search', resolution = 10),
    terminator = trm('evals', n_evals = 10)
)

# Fit pipeline
model_logistic_cv_pipeline$train(task, row_ids = splits$train)

# Assess pipeline on test
preds = model_logistic_cv_pipeline$predict(task, row_ids = splits$test)
preds$score(msr('classif.acc'))

# Save pipeline
# saveRDS(logistic_cv_pipeline, 'pipeline.rds')
```

```
classif.acc
      0.664
```

Development and deployment of pipelines will depend on your specific use case, and it can get notably complicated. Think of a case where your model is the culmination of features drawn from a dozen wildly different databases, and the model itself being a complex ensemble of models, each with its own hyperparameters. Your final modeling approach then produces predictions that are used in a variety of ways, from simple reports to real-time decision-making. Fun stuff!

You can imagine the complexity of the pipeline that would be required to handle all of that, but it is possible, and the entire pipeline from data ingestion to outputs that include reports, dashboards, etc. comprise the essence of **MLOps**

(Google (2024)). But even for your own personal model efforts, pipelines are a great way to organize your modeling work.

10.9 Wrapping Up

When machine learning began to take off, it seemed many in the field of statistics sat on their laurels, and often scoffed at these techniques that didn't bother to test their assumptions[15]! ML was, after all, mostly just a rehash of statistics, right? But the machine learning community, which actually comprised both computer scientists and statisticians, was able to make great strides in predictive performance, and the application of machine learning in myriad domains continues to enable us to push the boundaries of what is possible. Statistical analysis wasn't going to provide ChatGPT or self-driving cars, but it remains vitally important whenever we need to understand the uncertainty of our predictions, make causal statements, or want to make inferences about the data generating process. Eventually, the more general field of **data science** became the way people use traditional statistical analysis *and* machine learning to solve their data challenges. The best data scientists will be able to draw from both, use the best tool for the job, and as importantly, have fun with modeling!

10.9.1 The common thread

If using a model like the lasso or ridge regression, machine learning is simply a different focus to modeling compared to what we see in traditional linear modeling contexts. You could still do standard interpretation and statistical inference regarding the estimated coefficients. However, in traditional statistical application of linear models, we rarely see cross-validation or hyperparameter tuning. It does occur in some contexts though, and definitely *should* be more common.

As we will see, the generality of machine learning's approach allows us to use a wider variety of models than in standard linear model settings, and it incorporates those that are not easily summarized from a statistical standpoint,

[15]Brian Ripley, a core R developer in the early days, said: "To paraphrase provocatively, 'machine learning is statistics minus any checking of models and assumptions' ". Want to know what's even crazier than that statement? It was said by the guy who literally wrote the book on neural networks before anyone was even using them in any practical way! He's also the author of the nnet package in R, which existed even before there was a scikit-learn in Python. Also interesting to note is that techniques like the lasso, random forests, and others associated with machine learning actually came from established statisticians. In short, *there never was a statistics vs. machine learning divide*. Tools are tools, and the best data scientists will have many at their disposal for any project.

such as boosting and deep learning models. The key is that any model, from linear regression to deep learning, can be used with the tools of machine learning we've covered here.

10.9.2 Choose your own adventure

At this point, you're ready to dive in and run some common models used in machine learning for tabular data, so head to Chapter 11!

10.9.3 Additional resources

If looking for a deeper dive into some of these topics, here are some resources to consider:

- A core ML text is **Elements of Statistical Learning** (Hastie, Tibshirani, and Friedman (2017)) which paved the way for modern ML.
- A more recent treatment is **Probabilistic Machine Learning** (Murphy (2023)).

On the more applied side, you might consider courses on Coursera and similar ones, as some are both good and taught by some very well-known folks in machine learning. Michael got his first formal taste of ML from Andrew Ng's course on Coursera back in the day, and it was a great introduction. You can also get overviews on Google's Developer pages (Google (2023)). And if we're being honest, one of the mostly widely used resources for ML is the scikit-learn documentation.

Python resources include:

- **Machine Learning with PyTorch and Scikit-Learn** (Raschka (2022b))
- **An Introduction to Statistical Learning (Python)** (James et al. (2021))

R resources include:

- **An Introduction to Statistical Learning (R)** (James et al. (2021))
- **Applied Machine Learning for Tabular Data** (Kuhn and Johnson (2023))
- **Applied Machine Learning Using mlr3 in R** (Bischl et al. (2024))

Miscellaneous resources related to topics covered:

- Ridge - Bayesian connection
- Bias-Variance tradeoff
- A great thread on double descent using a GAM example by Daniela Witten
- Reconciling modern machine-learning practice and the classical bias-variance trade-off
- Overview of dropout in deep learning

- **Annotated History of Modern AI and Deep Learning** (Schmidhuber (2022))
- **Machine Learning Flashcards** (Albon (2024))

10.10 Guided Exploration

We did not run the pipeline previously, but we think that doing so would be a good way for you to put your new skills to the test.

1. Start by using the non-standardized features from the `movie_reviews` dataset.
2. Split the data into training and test sets.
3. Create a pipeline as we did previously that has at least two steps, e.g., scales the data and fits a model. Try a different model than the logistic regression we fit earlier (your choice).
4. Examine the validation set results.
5. Assess the pipeline's performance on the test set, but use a different metric than accuracy.
6. Bonus: Tune a hyperparameter for the model using a grid search or random search.

You can just modify the previous pipeline. Here is some helper code to get you going.

Python

```
# import the metrics and model you want
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import accuracy_score, roc_auc_score, recall_score
from sklearn.tree import DecisionTreeClassifier

pipeline = make_pipeline(
    SimpleImputer(strategy='mean'),
    StandardScaler(),
    RandomizedSearchCV(
        DecisionTreeClassifier(),
        param_distributions={'max_depth': [2, 5, 7]},
        cv=5,
        scoring='???',  # change to some other metric
    ),
)
```

```
# extract the best model from the pipeline
best_model = pipeline.named_steps['randomizedsearchcv'].best_estimator_

# extract the best parameter from the pipeline
best_model.max_depth

# ???(y_test, y_pred) # use your chosen metric on the test set
```

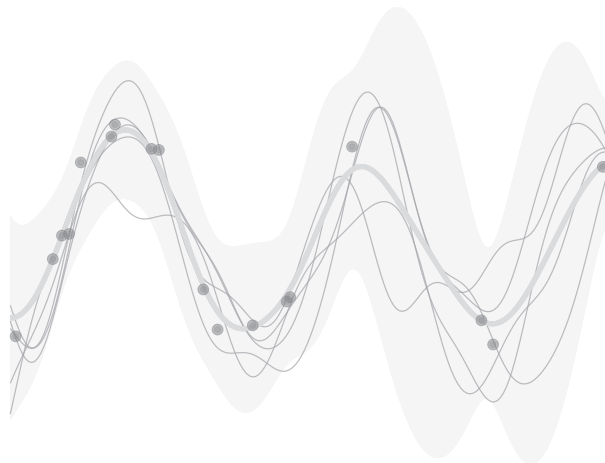
R

```
task = TaskClassif$new('movie_reviews', df_reviews, target = 'rating_good')
split = partition(task, ratio = 0.75) # set train/test split

# Define learner
learner = lrn(
    'classif.rpart',
    predict_type = 'prob', # get predicted probabilities
    cp = to_tune(1e-04, 1e-1, logscale = TRUE)
)

pipeline = ??? # see the text example

at = auto_tuner(
    tuner = tnr('random_search'),
    learner = pipeline,
    resampling = rsmp ('cv', folds = 5),
    measure = msr('classif.???'),  # change ??? e.g., try auc, recall, logloss
    term_evals = 10
)

#
at$train(task, row_ids = split$train)

at$model$learner$param_set$values # get the best parameter

at$predict(task, row_ids = split$test)$score(msr('classif.???')) # change ???
```

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

11

Common Models in Machine Learning



Before really getting into some machine learning models, let's get one thing straight from the outset: **any model may be used in machine learning**, from a standard linear model to a deep neural network. The key focus in ML is on performance, and generally we'll go with what works for the situation. This means that the modeler is often less concerned with the interpretation of the model, and more with the ability of the model to predict well on new data. But, as we'll see, we can do both if desired. In this chapter, we will explore some of the more common machine learning models and techniques.

11.1 Key Ideas

The take-home messages from this section include the following:

- Any model can be used with machine learning.
- A good and simple baseline is essential for interpreting your performance results.

- You only need a small set of tools (models) to go very far with machine learning.

11.1.1 Why this matters

Having the right tools in data science saves time and improves results, and using well-known tools means you'll have plenty of resources for help. It also allows you to focus more on the data and the problem, rather than the details of the model. A simple model might be all you need, but if you need something more complex, these models can still provide a performance benchmark.

11.1.2 Helpful context

Before diving in, it'd be helpful to be familiar with the following:

- Linear models, especially linear and logistic regression (Chapter 3 and Chapter 8)
- Basic machine learning concepts as outlined in Chapter 10
- Model estimation as outlined in Chapter 6

11.2 General Approach

Let's start with a general approach to machine learning to help us get some bearings. Here is an example outline of the process we could typically take. It incorporates some of the ideas we also cover in other chapters, and we'll demonstrate most of this in the following sections.

- Define the problem, including the target variable(s)
- Select the model(s) to be explored, including one baseline model
- Define the performance objective and metric(s) used for model assessment
- Define the search space (parameters, hyperparameters) for those models
- Define the search method (optimization)
- Implement a validation technique and collect the corresponding performance metrics
- Evaluate the chosen model on unseen data
- Interpret the results

Here is a more concrete example:

- Define the problem: predict the probability of heart disease given a set of features
- Select the model(s) to be used: ridge regression (main model), standard regression with no penalty (baseline)
- Define the objective and performance metric(s): RMSE, R-squared

- Define the search space (parameters, hyperparameters) for those models: ridge penalty parameter
- Define the search method (optimization): grid search
- Implement some sort of cross-validation technique: 5-fold cross-validation
- Evaluate the results on unseen data: RMSE on test data
- Interpret the results: the ridge regression model performed better than the baseline model, and the coefficients tell us something about the nature of the relationship between the features and the target

As we go along in this chapter, we'll see most of this in action. So let's get to it!

11.3 Data Setup

For our demonstration here, we'll use the heart disease dataset. This is a popular ML binary classification problem, where we want to predict whether a patient has heart disease, given information such as age, sex, resting heart rate, etc. (Section C.3).

There are two forms of the data that we'll use: one which is mostly in raw form, and one that is purely numeric, where the categorical features are dummy coded and where numeric variables have been standardized (Section 14.2). The purely numeric version will allow us to forgo any additional data processing for some model/package implementations (like penalized regression). We have also dropped the handful of rows with missing values, even though some techniques, like tree-based models, naturally handle missing values. This form of the data will allow us to use any model and make direct comparisons among them later.

Python

For Python we'll go ahead and do all the imports needed for this chapter.

```
# Basic data packages
import pandas as pd
import numpy as np

# Models
from sklearn.linear_model import LogisticRegression
from lightgbm import LGBMClassifier
from sklearn.neural_network import MLPClassifier

# Metrics and more
from sklearn.model_selection import (
```

```
    cross_validate, RandomizedSearchCV, train_test_split
)
from sklearn.metrics import accuracy_score
from sklearn.inspection import PartialDependenceDisplay
```

```
df_heart = pd.read_csv('https://tinyurl.com/heartdiseaseprocessed')
df_heart_num = pd.read_csv('https://tinyurl.com/heartdiseaseprocessednumeric')

# convert appropriate features to categorical
non_num_cols = df_heart.select_dtypes(exclude='number').columns
df_heart[non_num_cols] = df_heart[non_num_cols].astype('category')

X = df_heart_num.drop(columns=['heart_disease']).to_numpy()
y = df_heart_num['heart_disease'].to_numpy()

prevalence = np.mean(y)
majority = np.max([prevalence, 1 - prevalence])
```

R

```
library(tidyverse)

df_heart = read_csv('https://tinyurl.com/heartdiseaseprocessed') |>
    mutate(across(where(is.character), as.factor))

df_heart_num = read_csv('https://tinyurl.com/heartdiseaseprocessednumeric')

# for use with for mlr3
X = df_heart_num |>
    as_tibble() |>
    mutate(heart_disease = factor(heart_disease)) |>
    janitor::clean_names() # remove some symbols

prevalence = mean(df_heart_num$heart_disease)
majority = pmax(prevalence, 1 - prevalence)
```

In this data, roughly 46% suffered from heart disease, so if we're interested in accuracy, we could get 54% correct by just guessing the majority class of no disease. Hopefully we can do better than that!

One last thing, as we go along, performance metrics will vary depending on your setup (e.g., Python vs. R), package versions used, and other things. As such, your results may not look exactly like these, and that's okay! Your results

should still be similar, and the important thing is to understand the concepts and how to apply them to your own data.

11.4 Beat the Baseline

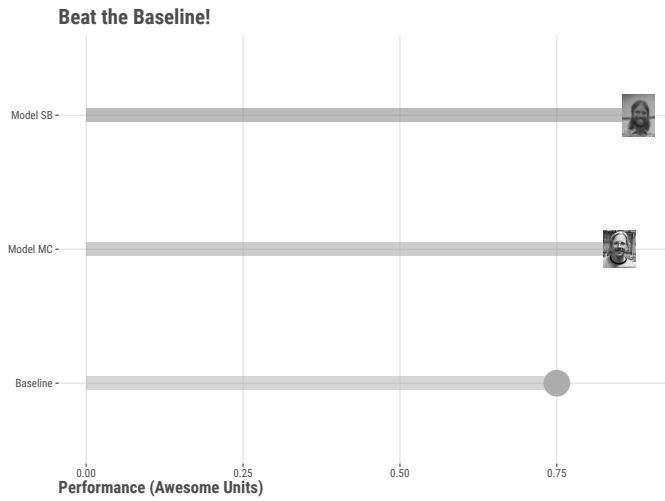


Figure 11.1: Hypothetical model comparison.

Before getting carried away with models, we should have a good reference point for performance – a **baseline model**. The baseline model should serve as a way to gauge how much better your model performs over one that is simpler, probably more computationally efficient, more interpretable, and is still *viable*. It could also be a model that is sufficiently complex to capture something about the data you are exploring, but not as complex as the models you're also interested in.

Take a classification model, for example. In this case we might use a logistic regression as a baseline. It is a viable model to begin answering some questions, and get a sense of performance possibilities, but it is often too simple to be adequately performant for many situations. We should be able to do better with more complex models, or if we can't, there is little justification for using them.

11.4.1 Why do we do this?

Having a baseline model can help you avoid wasting time and resources implementing more complex models, and to avoid mistakenly thinking performance is better than expected. It is probably rare, but sometimes relationships for the selected features and target are mostly or nearly linear and have little interaction. In this case, no amount of fancy modeling will make complex feature targets exist if they don't already. Also, if our baseline is a more complex model that actually incorporates nonlinear relationships and interactions (e.g., a GAMM), you'll often find that the more complex models often don't significantly improve on it. As a last example, in time series settings, a *moving average* can often be a difficult baseline to beat, so it can be a good starting point.

So you may find that the initial baseline model is good enough for your purposes, and you can then move on to other problems to solve, like acquiring data that is more predictive. This is especially true if you are working in a situation with limited time and resources.

11.4.2 How much better?

In many settings, it often isn't enough to merely beat the baseline model. Your model should perform *statistically* better. For instance, if your advanced model accuracy is 75% and your baseline model's accuracy is 73%, that's great. But, it's good to check if this 2% difference is statistically significant. Remember, accuracy and other metrics are *estimates* and come with uncertainty[1]. This means you can get a ranged estimate for them, as well as test whether they are different from one another. Table 11.1 shows an example comparison of 75% vs. 73% accuracy at different sample sizes. If the difference is not statistically significant, then it's possible you should stick with the baseline model, or maybe try a different approach to compete with it. This is because such a result means that the next time you run the model on new data, the baseline may actually perform better, or at least you can't be sure that it won't.

Table 11.1: Interval Estimates for Accuracy Differences

Sample Size	Lower Bound	Upper Bound	p-value
1000	−0.02	0.06	0.31
10000	0.01	0.03	0.00

Statistics regard the difference in proportions of .75 and .73.

[1]There would be far less hype and wasted time if those in ML and DL research simply did this rather than just reporting the chosen metric of their model 'winning' against other models. It'd also be nice if they used a more meaningful baseline than logistic regression, but that's a different story. And one more thing, although many papers also rank the competing models, ranks and mean ranks also have uncertainty, and ranks are typically *very* noisy.

That said, in some situations *any* performance increase is worth it, and even if we can't be certain a result is statistically better, any sign of improvement is worth pursuing. For example, if you are trying to predict the next word in a sentence, and your baseline is 70% accurate, and your new model is 72% accurate, that may be significant in terms of user experience. You should still try and show that this is a consistent increase and not a fluke if possible. In other settings, you'll need to make sure the cost is worth it. Is 2% worth millions of dollars? Six months of research? These are among many of the practical considerations you may have to make as well.

11.5 Penalized Linear Models

So let's get on with some models already! Let's use the classic linear model as our starting point for ML. We show explicitly how to estimate models like lasso and ridge regression in Section 6.8. Those work well as a baseline, and so should be in your ML modeling toolbox.

11.5.1 Elastic net

Another common linear model approach is **elastic net**, which we also saw in Chapter 10. It combines two techniques: lasso and ridge regression. We demonstrate the lasso and ridge penalties in Section 6.8, but all you have to know is that elastic net combines the two penalties: one for lasso and one for ridge, along with a standard objective function for a numeric or categorical target. The relative proportion of the two penalties is controlled by a mixing parameter, and the optimal value for it is determined by cross-validation. So for example, you might end up with a 75% lasso penalty and 25% ridge penalty. In the end though, it's just a slightly fancier logistic regression!

Let's apply this to the heart disease data. We are only doing simple cross-validation here to get a better performance assessment, but you are more than welcome to tune both the penalty parameter and the mixing ratio as we have demonstrated before (Section 10.7). We'll revisit hyperparameter tuning toward the end of this chapter.

Python

```
model_elastic = LogisticRegression(
    penalty = 'elasticnet',
    solver = 'saga',
    l1_ratio = 0.5,
    random_state = 42,
```

```
    max_iter = 10000,
    verbose = False,
)

model_elastic_cv = cross_validate(
    model_elastic,
    X,
    y,
    cv = 5,
    scoring = 'accuracy',
)

# pd.DataFrame(model_elastic_cv) # default output
```

```
Training accuracy:  0.828
Guessing:  0.539
```

R

```
library(mlr3verse)

tsk_elastic = as_task_classif(
    X,
    target = "heart_disease"
)

model_elastic = lrn(
    "classif.cv_glmnet",
    nfolds = 5,
    type.measure = "class",
    alpha = 0.5
)

model_elastic_cv = resample(
    task = tsk_elastic,
    learner = model_elastic,
    resampling = rsmp("cv", folds = 5)
)

# model_elastic_cv$aggregate(msr('classif.acc')) # default output
```

```
Training Accuracy: 0.825
Guessing: 0.539
```

So we're starting off with what seems to be a good model. Our average accuracy across the validation sets is definitely doing better than guessing, with a performance increase of more than 50%!

11.5.2 Strengths and weaknesses

Let's take a moment to consider the strengths and weaknesses of penalized regression models.

Strengths

- Intuitive approach. In the end, it's still just a standard regression model you're already familiar with.
- Widely used for many problems. Lasso/Ridge/ElasticNet would be fine to use in any setting you would use linear or logistic regression.
- A good baseline for tabular data problems.

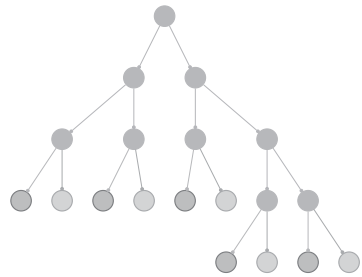
Weaknesses

- Does not automatically seek out interactions and nonlinearity, and as such will generally not be as predictive as other techniques.
- Variables have to be scaled or results will largely reflect data types.
- May have interpretability issues with correlated features.
- Relatively weaker performance compared to other models, especially in high-dimensional settings.

11.5.3 Additional thoughts

Using penalized regression is a very good default method in the tabular data setting, and it is something to strongly consider for more interpretation-focused model settings. These approaches predict better on new data than their standard, non-regularized complements, so they provide a nice balance between interpretability and predictive power. However, in general they are not going to be as strong of a method as others typically used in the machine learning world, and they may not even be competitive without a lot of feature engineering. If prediction is all you care about, you'll likely need something else. Now let's see if we can do better with other models!

11.6 Tree-based Models



Let's move beyond standard linear models and get into a notably different type of approach. Tree-based methods are a class of models that are very popular in machine learning contexts, and for good reason, they work *very* well. To get a sense of how they work, consider the following classification example where we want to predict a binary target as 'Yes' or 'No'.

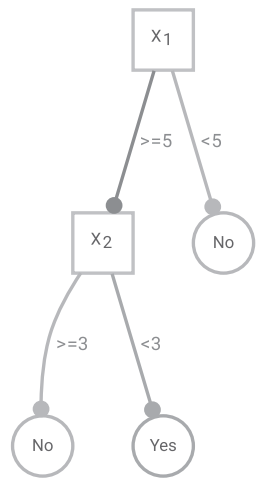


Figure 11.2: Simple classification tree.

We have two numeric features, X_1 and X_2. At the start, we take X_1 and make a split at the value of 5. Any observation less than 5 on X_1 goes to the right with a prediction of *No*. Any observation greater than or equal to 5 goes to the left, where we then split based on values of X_2. Any observation less than 3 goes to the right with a prediction of *Yes*. Any observation greater than or equal to 3 goes to the left with a prediction of *No*. So in the end, we see that an observation that is relatively lower on X_1, or relatively higher on both,

results in a prediction of *No*. On the other hand, an observation that is high on X_1 and low on X_2 results in a prediction of *Yes*.

This is a simple example, but it illustrates the core idea of a tree-based model, where the **tree** reflects the total process, and **branches** are represented by the splits going down, ultimately ending at **leaves** where predictions are made. We can also think of the tree as a series of `if-then` statements, where we start at the top and work our way down until we reach a leaf node, which is a prediction for all observations that qualify for that leaf.

A single tree would likely be the most interpretable model we could probably come up with. Furthermore, it incorporates nonlinearities through multiple branches on a single feature, interactions by branching across different features, and feature selection by excluding features that do not result in useful splits for the objective, all in one.

However, a single tree is not a very stable model unfortunately, and so it does not generalize well. For example, just a slight change in data, or even just starting with a different feature, might produce a very different tree[2]. Even though predictions could be similar, model interpretation would be very different.

The solution to that problem is straightforward though. By using the power of a bunch of trees, we can get predictions for each observation from each tree, and then average the predictions, resulting in a much more stable estimate. This is the concept behind both **random forests** (RF) and **gradient boosting** (GB), which can be seen as different algorithms to produce a bunch of trees. They are also considered types of **ensemble models**, which are models that combine the predictions of multiple models, to ultimately produce a single prediction for each observation. In this case each tree serves as a model.

Random forests and boosting methods are very easy to implement, to a point. However, there are typically several hyperparameters to consider for tuning. Here are just a few to think about:

- Number of trees
- Learning rate (GB)
- Maximum depth of each tree
- Minimum number of observations in each leaf
- Number of features to consider at each tree/split
- Regularization parameters (GB)
- Out-of-bag sample size (RF)

The number of trees is simply how many trees you want to build, and it is a key parameter setting for both RF and GB. For boosting models, the number of trees and learning rate play off of each other. Having more trees allows for

[2]A single regression/classification tree actually could serve as a decent baseline model, especially given the interpretability, and modern methods try to make them more stable.

a smaller rate[3], which might improve the model but will take longer to train. However, it can lead to overfitting if other steps are not taken.

The depth of each tree refers to how many levels we allow the model to branch out and is a crucial parameter. It controls the complexity of each tree, and thus the complexity of the overall model – less depth helps to avoid overfitting, but if the depth is too shallow, you won't be able to capture the nuances of the data. The minimum number of observations required for each leaf is also important for similar reasons. A lower number will allow for more complex trees, while a higher number will result in simpler trees.

It's also generally a good idea to take a random sample of features for each tree (or possibly even each branch), to also help reduce overfitting, but it's not obvious what proportion to take. The regularization parameters[4] are typically less important in practice, but can help reduce overfitting as in other modeling circumstances we've talked about. As with hyperparameters in other model settings, you'll use something like cross-validation to settle on final values.

11.6.1 Example with LightGBM

Here is an example of gradient boosting with the heart disease data. We'll explicitly set some of the parameters, and use 5-fold cross-validation to estimate performance.

Python

Although boosting methods are available in scikit-learn for Python, in general we recommend using the lightgbm or xgboost packages directly for boosting, as both have a sklearn API (as demonstrated). Also, they both provide R and Python implementations of the package, making it easy to not lose your place when switching between languages. We'll use lightgbm here[5].

```
model_boost = LGBMClassifier(
    n_estimators = 1000,
    learning_rate = 1e-3,
    max_depth = 5,
    verbose = -1,
```

[3]For boosting models, the learning rate is a scaling factor for the contribution of each tree to the overall model. A smaller learning rate means that each tree contributes less to the overall model, and so you'll need more trees to get the same performance, all else being equal.

[4]For boosting models, the regularization parameters are basically penalties on the weights of the leaves. For example, a smaller value would reduce the contribution of that leaf to the overall model, and so would help to reduce overfitting.

[5]Some also prefer catboost. Your humble authors have not actually been able to practically implement catboost in a setting where it was more predictive or as efficient/speedy as xgboost or lightgbm to get to the same performance level, but some have had notable success with it.

```
    random_state=42,
)

model_boost_cv = cross_validate(
    model_boost,
    df_heart.drop(columns='heart_disease'),
    df_heart['heart_disease'],
    cv = 5,
    scoring='accuracy',
)

# pd.DataFrame(model_boost_cv)
```

```
Training accuracy:  0.835
Guessing:  0.539
```

R

Note that as of writing, the mlr3 requires one of the extended packages for its implementation of lightgbm, and so we'll use the mlr3extralearners package.

```
library(mlr3verse)

# for lightgbm, you need mlr3extralearners and lightgbm package installed
# it is available from github via:
# remotes::install_github("mlr-org/mlr3extralearners@*release")
library(mlr3extralearners)

set.seed(42)

# Define task
# For consistency we use X, but lgbm can handle factors and missing data
# and so we can use the original df_heart if desired
tsk_boost = as_task_classif(
    df_heart,                   # can use the 'raw' data
    target = "heart_disease"
)

model_boost = lrn(
    "classif.lightgbm",
    num_iterations = 1000,
    learning_rate = 1e-3,
    max_depth = 5
)
```

```
model_boost_cv = resample(
    task = tsk_boost,
    learner = model_boost,
    resampling = rsmp("cv", folds = 5)
)
```

```
Training Accuracy: 0.828
Guessing: 0.539
```

So here we have a model that is also performing well, though not significantly better or worse than our elastic net model. For most tabular data situations, we'd expect boosting to do better, but this shows why we want a good baseline or simpler model for comparison. We'll revisit hyperparameter tuning using this model later.

11.6.2 Strengths and weaknesses

Random forests and boosting methods, though not new, are still 'state of the art' in terms of performance on tabular data like the type we've been using for our demos here. You'll often find that it will usually take considerable effort to beat them.

Strengths

- A single tree is highly interpretable.
- Relatively good prediction out of the box.
- Easily incorporates features of different types, regardless of scale or whether it's categorical.
- Tolerance to irrelevant features.
- Some tolerance to correlated inputs.
- Handling of missing values. Missing values are just another value to potentially split on[6].

Weaknesses

- Honestly few, but like all techniques, it might be relatively less predictive in certain situations. There is no free lunch.
- It does take more effort to tune relative to linear model methods, so this wouldn't be the best choice for a baseline model.
- Predictions, though relatively accurate, will be less smooth relative to some models like GAMs, and a smooth result may be more desirable in some settings.

[6]It's not clear why most model functions still have no default for this sort of thing in 2025. Is it that hard to drop or impute them with an informative message?

11.7 Deep Learning and Neural Networks

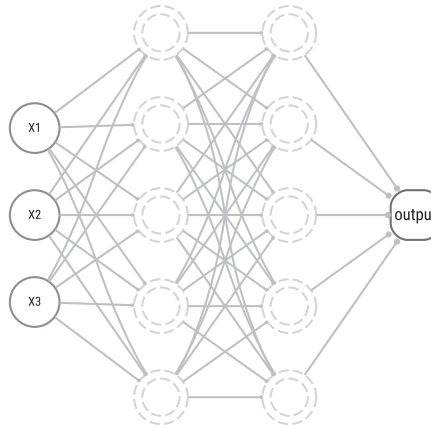


Figure 11.3: Neural network.

Deep learning has fundamentally transformed the world of data science, and, in many ways, the world itself. It has been used to solve problems in image detection, speech recognition, natural language processing, and more, from assisting with cancer diagnosis, to writing entire novels, providing self-driving cars, and even helping the formerly blind see. It is an extremely powerful tool.

For tabular data, however, the story is a bit different. Here, deep learning has consistently struggled to outperform models like boosting and even penalized regression in many cases. But while it is not always the best option, it should be in your modeling toolbox, if only because it potentially can be the most performant model and may well become the dominant model for tabular data in the future. Here we'll provide a brief overview of the key concepts behind neural networks, the underlying approach to deep learning, and then demonstrate how to implement a simple neural network to get things started.

11.7.1 What is a neural network?

Neural networks form the basis of deep learning models. They have actually been around a while – both computationally and conceptually going back decades[7, 8]. Like other models, they are computational tools that help us

[7]Most consider the scientific origin with McCulloch and Pitts (1943).

[8]On the conceptual side, they served as a rudimentary model of neuronal functioning in the brain, and a way to understand how the brain processes information. The models sprung

understand how to get outputs from inputs. However, they weren't quickly adopted due to computing limitations, similar to the slow adoption of Bayesian methods. But now, neural networks, or deep learning more generally, have recently become the go-to method for many problems.

11.7.2 How do they work?

At its core, a neural network can be seen as a series of matrix multiplications and other operations to produce combinations of features, and ultimately a desired output. We've been talking about inputs and outputs since the beginning (Section 2.3), but neural networks like to put a lot more in between the inputs and outputs than we've seen with other models. However, many of the key operations are often no different than what we've done with a basic linear model, and they sometimes even simpler! But the combinations of features they produce can represent many aspects of the data that are not easily captured by simpler models.

One notable difference from models we've been seeing is that neural networks implement *multiple combinations of features*, where each combination is referred to as a hidden **node** or unit[9]. In a neural network, each feature has a weight (or coefficient), just like in a linear model of the type we've used before. These features are multiplied by their weights and then added together. But we actually create multiple such combinations, as depicted in the 'H' or 'hidden' nodes in the following visualization.

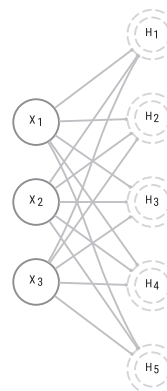


Figure 11.4: The first hidden layer.

from the cognitive revolution, a backlash against the behaviorist approach to psychology, and used the computer as a metaphor for how the brain might operate.

[9]The term 'hidden' is used because these nodes are between the input or output. It does not imply a latent/hidden variable in the sense used in many statistical models, but there is common ground. See the connection with principal components analysis, for example (Section 12.2.1).

The next phase is where things can get more interesting. We take those hidden units and add in nonlinear transformations before moving deeper into the network. The transformations applied are typically referred to as **activation functions**[10]. So, the output of the current (typically linear) part is transformed in a way that allows the model to incorporate nonlinearities. While this might sound new, this is just like how we use link functions in generalized linear models (Section 8.2). Furthermore, these multiple combinations also allow us to incorporate interactions between features.

But we can go even further! We can add more layers, and more nodes in each layer, even different types of layers, to create a **deep neural network**. We can also add components specific to certain types of processing, have some parts of the network only connected to certain other parts, apply specific computations to specific components, and more. The complexity really is only limited by our imagination, *and computational capacity*! This is what helps make neural networks so powerful. Given enough nodes, layers, and components, they can approximate ***any*** function, which could include the true function that connects our features to the target. Practically though, the feature inputs become an output or multiple outputs that can then be assessed in the same ways as other models.

Figure 11.5: Complex neural network.

[10]We have multiple options for our activation functions, and probably the most common one in deep learning is the **rectified linear unit** or ReLU, and its more recent variants. Others used include the sigmoid function, which is the same as what we used in logistic regression, the hyperbolic tangent function, and the linear/identity function, which does not do any transformation at all.

Before getting too carried away, let's simplify things a bit by returning to some familiar ground. Consider a logistic regression model. There we take the linear combination of features and weights, and then apply the sigmoid function (inverse logit) to it, and that is the output of the model that we compare to our observed target and calculate an objective function.

We can revisit a plot we saw earlier (Figure 3.7) to make things more concrete. The input features are X_1, X_2, and X_3, and the output is the probability of a positive outcome of a binary target. The weights are w_1, w_2, and w_3, and the bias[11] is w_0. The hidden node is just our linear predictor which we can create via matrix multiplication of the feature matrix and weights. The sigmoid function is the activation function, and the output is the probability of the chosen label.

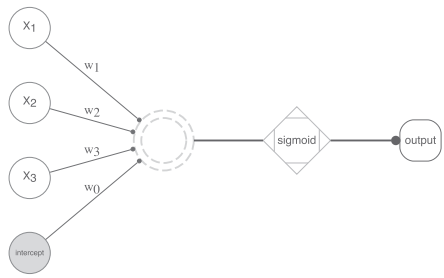


Figure 11.6: Logistic regression as a neural network with a single hidden layer with one node, and sigmoid activation

This shows that we can actually think of logistic regression as a very simple neural network, with a linear combination of the inputs as a single hidden node and a sigmoid activation function adding the nonlinear transformation. Indeed, the earliest **multilayer perceptron** models were just composed of multiple layers of logistic regressions!

> **i** GAMs and Neural Networks
>
> You can think of neural networks as nonlinear extensions of linear models. Regression approaches like GAMs and Gaussian process regression can be seen as approximations to neural networks (see also Rasmussen and Williams (2005)), bridging the gap between the simpler, and more interpretable linear model, and black box of a deep neural network. This brings us back to having a good baseline. If you know some simpler

[11] It's not exactly clear why computer scientists chose to call this the bias, but it's the same as the intercept in a linear model, or conceptually as an offset or constant. It has nothing to do with the word bias as used in every other modeling context.

tools that can approximate more complex ones, you can often get 'good enough' results with the simpler models.

11.7.3 Trying it out

The neural network model we'll use is a **multilayer perceptron** (MLP), which is a model like the one we've been showing. It consists of multiple hidden layers of potentially varying sizes, and we can incorporate activation functions as we see fit.

> **More on Neural Networks for Tabular Data**
>
> Be aware that this would be considered a bare minimum approach for a neural network, and generally you'd need to do more, even for standard tabular data. To begin with, you'd want to tune the **architecture**, or structure of hidden layers. For example, you might want to try more layers, as well as 'wider' layers, or more nodes per layer. Also, we'd usually want to use **embeddings** for categorical features as opposed to the one-hot approach used here (Section 14.2.2)[12].

For our demo, we'll use the numeric heart disease data with one-hot encoded categorical features. For our architecture, we'll use three hidden layers with 200 nodes each. As noted, these and other settings are hyperparameters that you'd normally prefer to tune, but we'll just set them as fixed parameters.

Python

For our demonstration we'll use sklearn's built-in `MLPClassifier`. We set the learning rate to 0.001. We set an **adaptive learning rate**, which is a way to automatically adjust the learning rate as the model trains. The ReLU activation function is default. We'll also use the **nesterov momentum** approach, which is a modification to an SGD variant (Adam). We use a **warm start**, which allows us to train the model in stages, and is useful for allowing the algorithm to stop before the maximum number of iterations. We'll also set the **validation fraction**, which is the proportion of data to use for the validation set. And finally, we'll use **shuffle** to shuffle each batch used during the SGD approach (Section 6.10.3).

```
model_mlp = MLPClassifier(
    hidden_layer_sizes = (200, 200, 200),
```

[12] A really good tool for a standard MLP type approach with automatic categorical embeddings is fastai's tabular learner. For a more flexible, DIY type of approach, consider the recently developed torch_frame package.

```
    learning_rate = 'adaptive',
    learning_rate_init = 0.001,
    shuffle = True,
    random_state = 123,
    warm_start = True,
    nesterovs_momentum = True,
    validation_fraction =  .2,
    verbose = False,
)

# with the above settings, this will take a few seconds
model_mlp_cv = cross_validate(
    model_mlp,
    X,
    y,
    cv = 5
)

# pd.DataFrame(model_mlp_cv) # default output
```

```
Training accuracy:  0.818
Guessing:  0.539
```

R

For R, we'll use mlr3torch, which calls pytorch directly under the hood. We'll use the same architecture as was done with the Python example. It uses the **ReLU** activation function as a default. We'll also use the **Adam** SGD variant as the optimizer, which is a popular choice in deep learning models, and the default for the sklearn approach. We'll use **cross-entropy** as the loss function, which is the same as the log loss objective function used in logistic regression and other ML classification models. We use a **batch size** of 16. Batch size is the number of observations to use for each batch of training. We'll also use **epochs** of 50, which is the number of times to train on the entire dataset (probably way more than necessary). We'll also use **predict type** of **prob**, which is the type of prediction to make. Finally, we'll use both **logloss** and **accuracy** as the metrics to track. As specified, this took over a minute.

```
library(mlr3torch)

learner_mlp = lrn(
    "classif.mlp",
    # defining network parameters
    neurons = c(200, 200, 200),
```

```
    # training parameters
    batch_size = 16,
    epochs = 50,
    # Defining the optimizer, loss, and callbacks
    optimizer = t_opt("adam", lr = 1e-3),
    loss = t_loss("cross_entropy"),
    # Measures to track
    measures_train = msrs(c("classif.logloss")),
    validate = .1,
    measures_valid = msrs(c("classif.logloss", "classif.ce")),
    # predict type (required by logloss)
    predict_type = "prob",
    seed = 123
)

tsk_mlp = as_task_classif(
    x = X,
    target = 'heart_disease'
)

# this will take a few seconds depending on your chosen settings and hardware
model_mlp_cv = resample(
    task = tsk_mlp,
    learner = learner_mlp,
    resampling = rsmp("cv", folds = 5),
)

model_mlp_cv$aggregate(msr("classif.acc")) # default output
```

```
Training Accuracy: 0.842
Guessing: 0.539
```

This model actually did pretty well, and we're on par with our accuracy as we were with the other two models. This is somewhat surprising given the nature of the data, small number of observations with different data types, a type of situation in which neural networks don't usually do as well as others. Just goes to show, you never know until you try!

Deep and Wide

A now relatively old question in deep learning is what is the better approach: deep networks, with more layers, or extremely wide (lots of neurons) and fewer layers? The answer is that it can depend on the problem, but in general, deep networks are more efficient and easier to

train, and will generalize better. Deeper networks have the ability to build upon what the previous layers have learned, basically compartmentalizing different parts of the task to learn. More important to the task is creating an architecture that is able to learn the appropriate aspects of the data, and generalize well.

11.7.4 Strengths and weaknesses

So why might we want to use neural networks for tabular data? The main reason is that they can be the most performant model and can potentially capture the most complex relationships in the data. They can also be used for a wide variety of data types and tasks. However, they are also the most complex model and can be the most difficult to tune and interpret.

Strengths

- Good prediction generally.
- Incorporates the predictive power of different combinations of inputs.
- Some tolerance to correlated inputs.
- Batch processing and parallelization of many operations makes it very efficient for large datasets.
- Can be used for even standard GLM approaches.
- Can be added as a component to other deep learning models (e.g., LLMs that are handling text input).

Weaknesses

- Susceptible to irrelevant features.
- Doesn't consistently outperform other methods that are easier to implement on tabular data.

11.8 Tuned Example

We noted in the chapter on machine learning concepts that there are often multiple hyperparameters we are concerned with for a given model (Section 10.7). We had hyperparameters for each of the models in this chapter also. For the elastic net model, we might want to tune the penalty parameters and the mixing ratio. For the boosting method, we might want to tune the number of trees, the learning rate, the maximum depth of each tree, the minimum number of observations in each leaf, and the number of features to consider at each tree/split. And for the neural network, we might want to tune the number of hidden layers, the number of nodes in each layer, the learning rate,

the batch size, the number of epochs, and the activation function. There is plenty to explore!

Here is an example of a hyperparameter search using the boosting model. We'll tune the number of trees, the learning rate, the minimum number of observations in each leaf, and the maximum depth of each tree. We'll use a **randomized search** across the parameter space to sample from the set of hyperparameters, rather than searching every possible combination as in a **grid search**. This is a good approach when you have a lot of hyperparameters to tune, and/or when you have a lot of data.

Python

```
# train-test split
X_train, X_test, y_train, y_test = train_test_split(
    df_heart.drop(columns='heart_disease'),
    df_heart['heart_disease'],
    test_size = 0.2,
    random_state = 42
)

model_boost = LGBMClassifier(verbose = -1)

param_grid = {
    'n_estimators': [500, 1000],
    'learning_rate': [1e-3, 1e-2, 1e-1],
    'max_depth': [3, 5, 7, 9],
    'min_child_samples': [1, 5, 10],
}

# this will take a few seconds
model_boost_cv_tune = RandomizedSearchCV(
    model_boost,
    param_grid,
    n_iter = 10,
    cv = 5,
    scoring = 'accuracy',
    n_jobs = -1,
    random_state = 42
)

model_boost_cv_tune.fit(X_train, y_train)

test_predictions = model_boost_cv_tune.predict(X_test)
accuracy_score(y_test, test_predictions)
```

```
Test Accuracy 0.8
Guessing:  0.539
```

R

```
set.seed(1234)

tsk_model_boost_cv_tune = as_task_classif(
    df_heart,
    target = "heart_disease",
    positive = "yes"
)

split = partition(tsk_model_boost_cv_tune, ratio = .8)

lrn_lgbm = lrn(
    "classif.lightgbm",
    num_iterations = to_tune(c(500, 1000)),
    learning_rate = to_tune(1e-3, 1e-1, logscale = TRUE),
    max_depth = to_tune(c(3, 5, 7, 9)),
    min_data_in_leaf = to_tune(c(1, 5, 10))
)

model_boost_cv_tune = auto_tuner(
    tuner = tnr("random_search"),
    learner = lrn_lgbm,
    resampling = rsmp("cv", folds = 5),
    measure = msr("classif.acc"),
    terminator = trm("evals", n_evals = 10)
)

model_boost_cv_tune$train(tsk_model_boost_cv_tune, row_ids = split$train)

test_preds = model_boost_cv_tune$predict(
    tsk_model_boost_cv_tune,
    row_ids = split$test
)

test_preds$score(msr("classif.acc"))
```

```
Test Accuracy: 0.831
Guessing: 0.539
```

It looks like we've done a lot better than guessing. Even if we don't do better than our previously untuned model, we should feel better that we've done our due diligence in trying to find the best set of underlying parameters, rather than just going with defaults or what *seems* to work best.

11.9 Comparing Models

We can tune all the models and compare them head to head. For this demo, we'll just describe what we did, as you've seen the code for how to do so throughout this chapter already. We first split the same data into training and test sets (20% test). Then with training data, we tuned each model over different settings:

- Elastic net: penalty and mixing ratio
- Boosting: number of trees, learning rate, and maximum depth, etc.
- Neural network: number of hidden layers, number of nodes in each layer, etc.

After this, we used the tuned values to retrain the model on the complete training dataset. At this stage it's not necessary to investigate in most settings, but we show the results of the 10-fold cross-validation for the already-tuned models, to give a sense of the uncertainty in error estimation with a small sample like this. Even with the 'best' settings, we can see that there is definitely some variability across data splits.

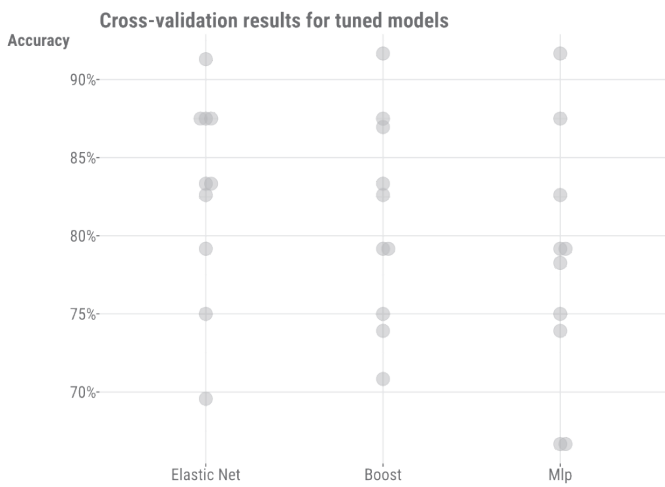


Figure 11.7: Cross-validation results for tuned models.

We now look at the performance on the holdout set with our tuned models in the following table[13]. In this case, we see something that might surprise you – the simplest model does really well! In this case, we'd probably declare it the winner given the combination of ease of use and interpretability. Again, your results may vary depending on whether you used a seed, R vs. Python, and possibly other aspects of your modeling environment.

Table 11.2: Metrics for Tuned Models on Holdout Data

model	acc	tpr	tnr	f1	ppv	npv
Elastic Net	0.88	0.83	0.92	0.85	0.87	0.89
Boost	0.85	0.92	0.81	0.83	0.76	0.94
MLP	0.80	0.83	0.78	0.77	0.71	0.88

It's important to note that, for each metric, none of the model results are *statistically different* from each other. As an example, the elastic net model had an accuracy of 0.88, but the interval estimate for such a small holdout sample is very wide – from 0.77 to 0.95. The interval estimate for the *difference* in accuracy between the elastic net and boosting models is from -0.1 to 0.17[14]. Again, we shouldn't take this result too far, as we're dealing with a small dataset and it is difficult to detect potentially complex relationships in such a setting. In addition, we could have done more to explore the parameter space of the models, but we'll leave that for another time. But this was a good example of the importance of having an adequate baseline, and where complexity didn't really help much, though all our approaches did reasonably well.

Test Metrics Better than Training?

Some may wonder how the holdout results can be better than training, which you might have seen in playing around with the models for this data. This can definitely happen and, at least in this case, would probably just reflect the small sample size. The holdout set is a random sample of 20% of the complete data, which is 59 examples. Just slightly different predictions could result in a several percentage point difference in accuracy. In general though, you'd expect the holdout results to be a bit, or even significantly, worse than the training results.

[13]This table was based on Python with randomized CV search, but the R approach produced similar results. However, they can both vary quite a bit even with just a random seed change due to the small sample size.

[14]We just used the `prop.test` function in R for these values with the key question of whether these proportions are different. A lot of the metrics people look at from confusion matrices are proportions.

11.10 Interpretation

When it comes to machine learning, many models we use don't have an easy interpretation, like with coefficients in a linear regression model. However, that doesn't mean we can't still figure out what's going on. Let's use the boosting model as an example.

11.10.1 Feature importance

The default importance metric for a lightgbm model is the number of splits in which a feature is used across trees, and this will depend a lot on the chosen parameters of the best model. For the table below, we show the top 4 features from the tuned model and values rescaled to be between 0 and 1 for easier comparison. But there are other ways to think about what importance means that will be specific to a model, data setting, and the ultimate goal of the modeling process.

Python

```
# Get feature importances
best_model = model_boost_cv_tune.best_estimator_
best_model.feature_importances_ # seriously, no feature names?

# if it's not obvious which of these values belongs to which feature, do this:
pd.DataFrame({
    'Feature': best_model.feature_name_,
    'Importance': best_model.feature_importances_
}).sort_values('Importance', ascending=False)
```

R

R shows the proportion of splits in which a feature is used across trees rather than the raw number.

```
# Get feature importances
model_boost_cv_tune$learner$importance()
```

Table 11.3: Top 4 Features from a Tuned LGBM model

Feature	Importance
age	1.00
cholesterol	0.97
max_heart_rate	0.83
resting_bp	0.68

Now let's think about a visual display to aid our understanding. Here we show a partial dependence plot (Section 5.8) to see the effects of cholesterol and being male. From this we can see that males are expected to have a higher probability of heart disease, and that cholesterol has a positive relationship with heart disease, though this occurs mostly after midpoint for cholesterol (shown by vertical line). The plot shown is a prettier version of what you'd get with the following code, but the model predictions are the same.

Python

```
PartialDependenceDisplay.from_estimator(
    model_boost_cv_tune,
    df_heart.drop(columns='heart_disease'),
    features=['cholesterol', 'male'],
    categorical_features=['male'],
    percentiles=(0, .9),
    grid_resolution=75
)
```

R

For R we'll use the iml package.

```
library(iml)

prediction = Predictor$new(
    model_boost_cv_tune$model$learner,
    data = df_heart,
    type = 'prob',
    class = 'yes'
)

# interaction plot, select a single feature for a single feature plot
effect_dat = FeatureEffect$new(
    prediction,
    feature = c('cholesterol', 'male'),
```

```
    method = "pdp",
)

effect_dat$plot(show.data = TRUE)
```

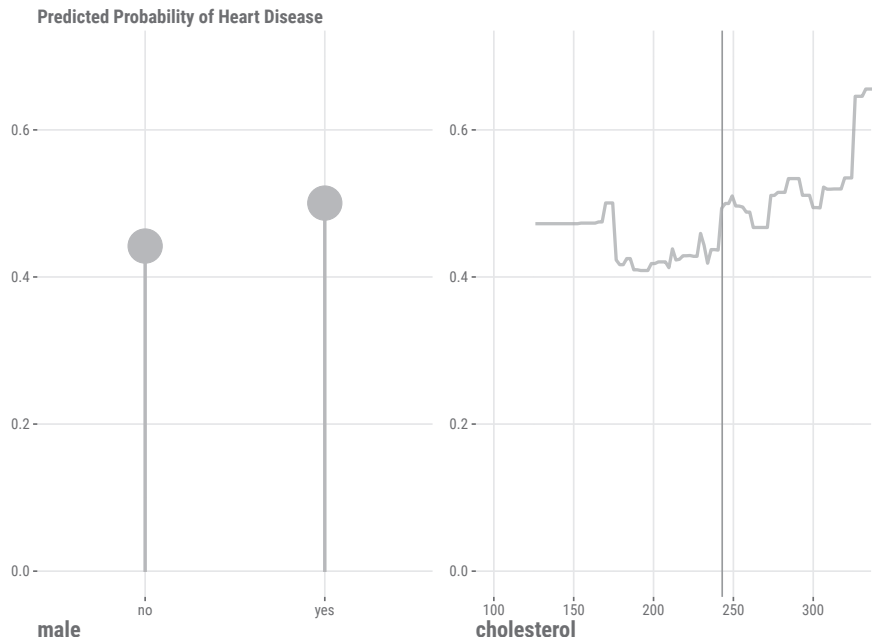


Figure 11.8: Partial dependence plot for cholesterol.

11.11 Other ML Models for Tabular Data

When you research classical machine learning models for the kind of data we've been exploring, you'll find a variety of methods. Popular approaches from the past include k-nearest neighbors regression, principal components regression, support vector machines (SVM), and more. You don't see these used in practice as much though for several reasons:

- Some, like k-nearest neighbors regression, generally don't predict as well as other models.
- Others, like linear discriminant analysis, make strong assumptions about how the data is distributed.

- Some models, like SVM, tend to work well only with 'clean' and well-structured data of the same type.
- Many of these models' standard approach is computationally demanding, making them less practical for large datasets.
- Lastly, some of these models are less interpretable, making it hard to understand their predictions without an obvious gain in performance.

While some of these classical models might still work well in unique situations, when you have tools that can handle a lot of data complexity and predict very well (and usually better) like tree-based methods, there's not much reason to use the historical alternatives. If you're interested in learning more about them or think one of them is just 'neat', you could potentially use it as a baseline model. Alternatively, you could maybe employ them as part of an ensemble or **stacked** model, where you combine the predictions of multiple models to produce a single prediction. This is a common approach in machine learning and is often used in Kaggle competitions.

There are also other methods that are more specialized, such as those for text, image, and audio data. We will provide an overview of these elsewhere (Chapter 12). Currently, the main research effort for new models for tabular data regards deep learning methods like large language models (LLMs). While typically used for text data, they can be adapted for tabular data as well. They are very powerful but also computationally expensive. The issue is primarily whether a model can be devised that can consistently beat boosting and other approaches that already do very well. While it hasn't happened yet, there is a good chance it will in the near future. For now, the best approach is to use the best model that works for your data and to be open to new methods as they come along.

> **i** SOTA Deep Learning for Tabular Data
>
> As of this writing, the current state of the art (SOTA) for deep learning on tabular data appears to be techniques like TabR (Gorishniy et al. (2023)) and Modern NCA (Ye, Yin, and Zhan (2024)). These are very new and not yet widely used, but they are showing promise in some benchmarks.

11.12 Wrapping Up

In this chapter we've provided a few common and successful models you can implement with much success in machine learning. You don't really need much beyond these for tabular data unless your unique data condition somehow

requires it. But here are some things are worth mentioning before moving on from models in machine learning:

Thinking hard about the problem and the data is more important than the model choice.

Feature engineering will typically pay off more in performance than the model choice.

The best model is simply the one that works best for your situation.

You'll always get more payoff by coming up with better features to use in the model, as well as just using better data that's been 'fixed' because you've done some good exploratory data analysis. Thinking harder about the problem means you will waste less time going down dead-ends. You also can find better data to use to solve the problem by thinking more clearly about the question at hand. And finally, it's good to not be stuck on one model and be willing to use something new to get the job done.

11.12.1 The common thread

When it comes to machine learning, you can use any model you feel like, and this could be standard statistical models like we've covered elsewhere. Both boosting and neural networks, like GAMs and related techniques, can be put under a common heading of *basis function models*. GAMs with certain types of smooth functions are approximations of Gaussian processes, and Gaussian processes are equivalent to a neural network with an infinitely wide hidden layer (Neal (1996)). Even the most complicated deep learning model typically has components that involve feature combinations and transformations that we use in far simpler models like linear regression.

11.12.2 Choose your own adventure

If you haven't had much exposure to statistical approaches, we suggest heading to any chapter before Chapter 10. Otherwise, consider an overview of more machine learning techniques (Chapter 12), data-specific considerations (Chapter 14), or causal modeling (Chapter 13).

11.12.3 Additional resources

Additional resources include those mentioned in Section 10.9.3, but here are some more to consider:

- Interpretable ML (Molnar (2023))
- Interpretable Machine Learning with Python (Masis (2023))
- Machine Learning Q & AI (Raschka (2023b))

- Google's Course on Decision Forests

For deep learning specifically:

- Common activation functions
- An overview of deep learning applications for tabular data by Michael (see Clark 2021b, 2022a)
- Dive into Deep Learning (Zhang et al. (2023))
- Fast AI course (Howard (2024))

11.13 Guided Exploration

Tune a model of your choice to predict whether a movie is good or bad with the movie review data. Use the categorical target, and use one-hot encoded features if needed. Make sure you use a good baseline model for comparison!

Python

```
df_reviews = pd.read_csv('https://tinyurl.com/moviereviewsdata')

df_reviews_sub = df_reviews[[
    'review_year',
    'age',
    'children_in_home',
    'education',
    'work_status',
    'genre',
    'release_year',
    'word_count',
    'rating_good'
]]


X_train, X_test, y_train, y_test = train_test_split(
    df_reviews_sub.drop(columns='rating_good'),
    df_heart_num['rating_good'],
    test_size = ???,
    random_state = 42
)

model_boost = LGBMClassifier(
    verbose = -1
```

```
)

param_grid = {
    'n_estimators': ???,
    'learning_rate': ???,
    'max_depth': ???,
    'min_child_samples': ???,
}

# this will take a few seconds
model_boost_cv_tune = RandomizedSearchCV(
    model_boost,
    param_grid,
    n_iter = 10,
    cv = ???,
    scoring = ????,
    n_jobs = -1,
    random_state = 42
)

model_boost_cv_tune.fit(X_train, y_train)

test_predictions = model_boost_cv_tune.predict(X_test)
accuracy_score(y_test, test_predictions)
```

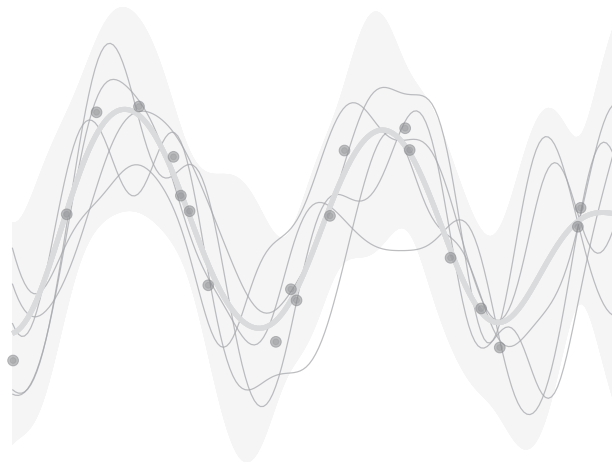
R

```
df_reviews = read_csv('https://tinyurl.com/moviereviewsdata')

df_reviews_sub = df_reviews %>%
    select(
        review_year,
        age,
        children_in_home,
        education,
        work_status,
        genre,
        release_year,
        word_count,
        rating_good
    ) |>
    mutate(
        across(where(is.character), \(x) as.factor(x))
```

```
    )

set.seed(42)

tsk_model_boost_cv_tune = as_task_classif(
    df_reviews_sub,
    target = "rating_good"
)

split = partition(tsk_model_boost_cv_tune, ratio = ??)

lrn_lgbm = lrn(
    "classif.lightgbm",
    num_iterations = to_tune(c(???, ???)),
    learning_rate = to_tune(1e-3, 1e-1, logscale = TRUE),
    max_depth = to_tune(c(???, ???)),
    min_data_in_leaf = to_tune(c(???, ???))
)

model_boost_cv_tune = auto_tuner(
    tuner = tnr("random_search"),
    learner = lrn_lgbm,
    resampling = rsmp("cv", folds = ???),
    measure = msr("classif.acc"),
    terminator = trm("evals", n_evals = ???)
)

model_boost_cv_tune$train(tsk_model_boost_cv_tune, row_ids = split$train)
model_boost_cv_tune$predict(
    tsk_model_boost_cv_tune,
    row_ids = split$test
)$score(msr("classif.acc"))
```

12

Extending Machine Learning



We've explored some fundamental aspects of machine learning (ML) for typical data settings and modeling objectives, but there are many other areas of ML that we haven't covered, and honestly, you just can't cover everything in a single book. The field is always evolving, progressing, branching out, and covers every data domain, which is what makes it so fun! Here we'll briefly discuss some of the other aspects of ML that you'll want to be aware of as you continue your journey.

12.1 Key Ideas

As we wrap up our focus on ML, here are some things to keep in mind:

- ML can be applied to virtually any modeling or data domain.
- Other widely used areas and applications of ML include unsupervised learning, reinforcement learning, computer vision, natural language processing, and more generally, artificial intelligence.
- While tabular data has traditionally been the primary format for modeling,

the landscape has changed dramatically, and you may need to incorporate other data to reach your modeling goals.

12.1.1 Why this matters

It's very important to know just how *unlimited* the modeling universe is, but also to recognize the common thread that connects all models. Even when we get into other data situations and complex models, we can always fall back on the core approaches we've already seen and know well at this point, and know that those ideas can potentially be applied in any modeling situation.

12.1.2 Helpful context

For the content in this chapter, a basic idea of modeling and machine learning would probably be enough. We're not going to get too technical in this section.

12.2 Unsupervised Learning

All the models considered thus far would fall under **supervised learning**. That is, we have a target variable that we are trying to predict with various features, and we use the data to train a model to predict it. However, there are settings in which we do not have a target variable, or we do not have a target variable for all of the data. In these cases, we can still use what's often referred to as **unsupervised learning** to learn about the data.

Unsupervised learning is a type of machine learning that involves training a model without an explicit target variable in the sense that we've seen. But to be clear, a model and target is still definitely there! Unsupervised learning attempts learn patterns in the data in a general sense and can be used in a wide range of applications, including cluster analysis, anomaly detection, and dimensionality reduction. Although these may initially seem as fundamentally different modeling approaches, just like much of what we've seen, it's probably best to think of these as different flavors of a more general approach.

Traditionally, one of the more common applications of unsupervised learning falls under the heading of **dimension reduction**, or **data compression**. Here we reduce our feature set to a smaller **latent**, or hidden, or unobserved, subset that accounts for most of the (co-)variance of the larger set. Alternatively, we may reduce the rows to a small number of hidden, or unobserved, clusters. For example, we start with 100 features and reduce them to 10 features that still account for most of what's important in the original set, or we classify each observation as belonging to 2-3 clusters. Either way, the primary goal is to reduce the dimensionality of the data, not predict an explicit target.

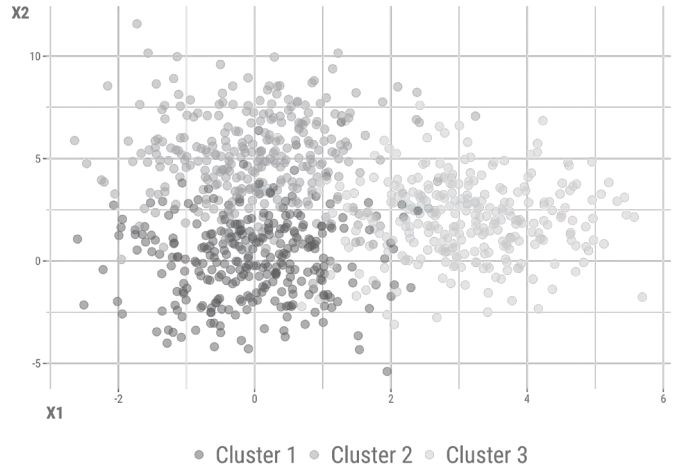


Figure 12.1: Two variables with three overlapping clusters.

Classical methods in this domain include **principal components analysis** (PCA), **singular value decomposition** (SVD), **factor analysis**, and **latent Dirichlet allocation**, which are geared toward reducing column dimensions. Also included are clustering methods such as **k-means** and **hierarchical clustering**, where we reduce observations into clusters or groups. Sometimes, these methods are often used as preprocessing steps for supervised learning problems, or as a part of exploratory data analysis, but often they are an end in themselves.

Most of us are familiar with **recommender systems**, e.g., with Netflix or Amazon recommendations, which suggest products or movies, and we're all now becoming extremely familiar with text analysis methods through chatbots and similar tools. While the underlying models are notably more complex these days, they actually just started off as SVD (recommender systems) or a form of factor analysis (text analysis via latent semantic analysis/latent Dirichlet allocation). Having a conceptual understanding of the simpler methods can aid in understanding the more complex ones.

i Dimension Reduction in Preprocessing

You probably should not use a dimension reduction technique as a preprocessing step for a supervised learning problem. Instead, use a modeling approach that can handle high-dimensional data, has a built-in way to reduce features (e.g., lasso, boosting, dropout), or use a dimension reduction technique that is specifically designed for supervised learning

(e.g., partial least squares). Creating a reduced set of features, but which are created without any connection to the target, will generally be suboptimal for a supervised learning problem.

12.2.1 Connections

Clusters are latent categorical features

In both clustering rows and reducing columns, we're essentially reducing the dimension of the features. For methods like PCA and factor analysis, we're explicitly reducing the number of data columns to a smaller set of numeric features. For example, we might take answers to responses to dozens of questions from a personality inventory, and reduce them to five key features that represent general aspects of personality. These new features are on their own scale, often standardized, but they still reflect at least some of the original items' variability[1].

Now, imagine if we reduced the features to a single categorical variable, say, with two or three groups. Now you have cluster analysis! You can discretize any continuous feature to a coarser set of categories, and this goes for latent variables as well as those we actually observe in our data. For example, if we do a factor analysis with one latent feature, we could either convert it to a probability of some class with an appropriate transformation, or just say that scores higher than some cutoff are in cluster A and the others are in cluster B. Indeed, there is a whole class of clustering models called **mixture models** that do just that – they estimate the latent probability of class membership. Many of these approaches are conceptually similar or even identical to the continuous method counterparts, and the primary difference is how we think about and interpret the results.

PCA as a neural network

Consider the following neural network, called an **autoencoder**. Its job is to shrink the features down to a smaller, simpler representation, and then rebuild the feature set from the compressed state, resulting in an output that matches the original as closely as possible. It's trained by minimizing the error between the original data and the reconstructed data. The autoencoder is a special case of a neural network used as a component of many larger architectures such as those seen with large language models, but it can be used for dimension

[1] Ideally we'd capture all the variability, but that's not the end result, and some techniques or results may only capture a relatively small percentage. In our personality example, this could be because the questions don't adequately capture the underlying personality constructs (i.e., an issue of the reliability of instrument), or because personality is just not that simple and we'd need more dimensions.

reduction in and of itself if we are specifically interested in the compression layer, sometimes called a **bottleneck**.

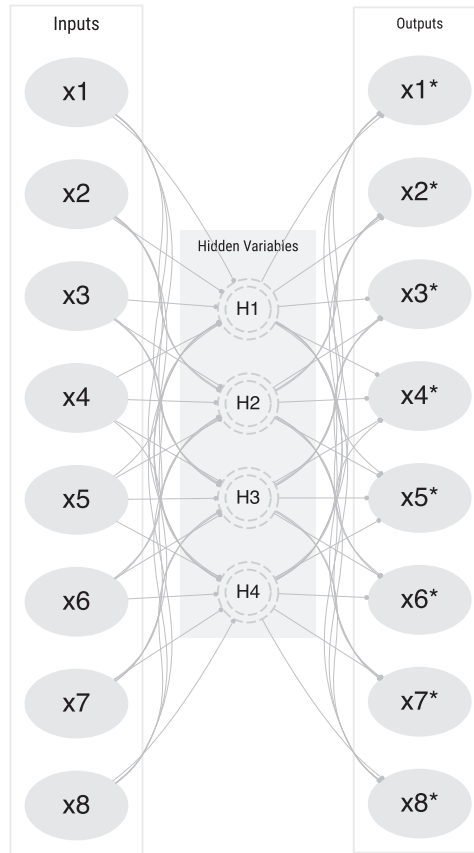


Figure 12.2: PCA or autoencoder.

Consider the following setup for such a situation:

- Single hidden layer
- Number of hidden nodes = number of inputs
- Linear activation function

An autoencoder in this case would be equivalent to principal components analysis. In the approach described, PCA perfectly reconstructs the original data when considering all components, and so the error would be zero. But that doesn't give us any dimension reduction, as we have as many nodes in the compression layer as we did inputs. So with PCA, we often only focus on a

small number of components that capture the data variance by some arbitrary amount. The discarded nodes are actually still estimated though.

Neural networks are not bound to linear activation functions, the size of the inputs, or even a single layer. As such, they provide a much more flexible approach that can compress the data at a certain layer, but still have very good reconstruction error. Typical autoencoders would have multiple layers with notably more nodes than inputs, at least for some layers. They may ultimately compress to a bottleneck layer consisting of a fewer set of nodes, before expanding out again. An autoencoder is not as easily interpretable as typical factor analytic techniques, and we still have to sort out the architecture. However, it's a good example of how the same underlying approach can be used for different purposes.

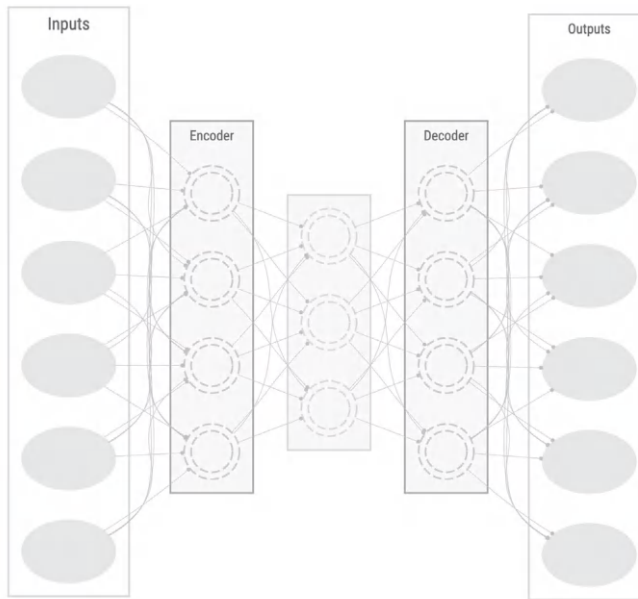


Figure 12.3: Conceptual diagram of an autoencoder.

Autoencoders and LLMs

Encoder-decoder models, which are the basis for large language models (LLMs), can be seen as a type of autoencoder, and are used in many applications, including machine translation, image captioning, and more. Autoencoders suggest the same inputs and outputs, but a similar type of architecture, or even part of it, as in 'decoder-only' approaches of many

of the current popular LLMs, might be used to classify or generate text, as with large language models.

Latent linear models

Some dimension reduction techniques can be thought of as *latent linear models*. The following depicts factor analysis as a latent linear model. The 'targets' are the observed features, and we predict each one by some linear combination of latent variables.

$$
\begin{aligned}
x_1 &= \beta_{11}h_1 + \beta_{12}h_2 + \beta_{13}h_3 + \beta_{14}h_4 + \epsilon_1 \\
x_2 &= \beta_{21}h_1 + \beta_{22}h_2 + \beta_{23}h_3 + \beta_{24}h_4 + \epsilon_2 \\
x_3 &= \beta_{31}h_1 + \beta_{32}h_2 + \beta_{33}h_3 + \beta_{34}h_4 + \epsilon_3
\end{aligned}
$$

In this scenario, the h are estimated latent variables, and β are the coefficients, which in some contexts are called **loadings**. The ϵ are the residuals, which are assumed to be independent and normally distributed as with a standard linear model. The β are usually estimated by maximum likelihood. The latent variables are not observed, but are to be estimated as part of the modeling process, and typically standardized with mean 0 and standard deviation of 1[2]. The number of latent variables we use is a hyperparameter in the ML sense and so they can be determined by the usual means[3]. To tie some more common models together:

- Factor analysis is the more general approach with varying residual variance. In a multivariate sense, we can write the model with $\mathbf{X}$ is the data matrix, $\mathbf{W}$ is the loading matrix (weights), $\mathbf{Z}$ is the latent variable matrix, and Ψ is the covariance.

$$\mathbf{X} = \mathrm{N}(\mathbf{ZW}, \Psi)$$

- Probabilistic PCA is a factor analysis with $\Psi = \sigma^2\mathbf{I}$, where σ^2 is the (constant across X) residual variance.
- PCA is a factor analysis with no (residual) variance, and the latent variables are orthogonal (independent).
- Independent component analysis is a factor analysis that does not assume an underlying Gaussian data generating process.
- Non-negative matrix factorization and latent Dirichlet allocation are factor analyses applied to counts (think Poisson and multinomial regression).

[2]They can also be derived in post-processing depending on the estimation approach.

[3]Actually, in application as typically seen in social sciences, cross-validation is very rarely employed, and the number of latent variables is determined by some combination of theory, model comparison for training data only, or trial and error. Not that we're advocating for that, but it's a common practice.

In other words, many traditional dimension reduction techniques can be formulated in the context of a linear model.

12.2.2 Other unsupervised learning techniques

There are several techniques that are used to visualize high-dimensional data in simpler ways, such as **multidimensional scaling**, **t-SNE**, and **(H)DBSCAN**. These are often used as a part of exploratory data analysis to identify groups.

Cluster analysis is a method with a long history and many different approaches, including hierarchical clustering algorithms (agglomerative, divisive), k-means, and more. Distance matrices are often the first step for these clustering approaches, and there are many ways to calculate distances between observations. With the distances we can group observations with small distances and separate those with large distances. Conversely, some methods use adjacency matrices, which focus on similarity of observations rather than differences (like correlations), and can be used for graph-based approaches to find hidden clusters (see network analysis).

Anomaly/outlier detection is an approach for finding 'unusual' data points, or otherwise small, atypical clusters. This is often done by looking for data points that are far from the rest of the data, or that are not well explained by the model. This approach is often used for situations like fraud detection or network intrusion detection. For example, standard clustering (small anomalous groups) or modeling techniques (observations with high residuals) might be used to identify outliers.

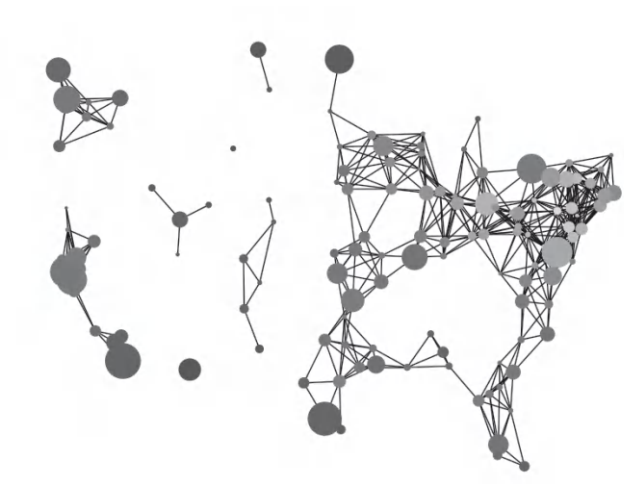


Figure 12.4: Network graph regarding some US cities.

Network analysis is a type of unsupervised learning that involves analyzing the relationships between entities. It is a graph-based approach that involves identifying nodes (e.g., people) and edges (e.g., do they know each other?) in a network. It is used in a wide range of applications, like identifying communities within a network or seeing how they evolve over time. It is also used to identify relationships between entities, such as people, products, or documents. One might be interested in such things as which nodes that have the most connections, or the general 'connectedness' of a network. Network analysis or similar graphical models typically have their own clustering techniques that are based on the edge (connection) weights between individuals, such as modularity, or the number of edges between individuals, such as k-clique.

In summary, there are many methods that fall under the umbrella of unsupervised learning, but even when you don't think you have an explicit target variable, you can still understand or frame these as models in familiar ways. It's important to not get hung up on trying to distinguish modeling approaches with somewhat arbitrary labels, and focus more on what their modeling goal is and how best to achieve it!

Generative vs. Discriminative Models

Many unsupervised learning and many deep learning techniques involved in computer vision and natural language processing are often thought of as **generative** models. These attempt to model the underlying data generating process, i.e., the features, but possibly a target variable also. In contrast, most supervised learning models are often thought of as **discriminative** models that try to model the conditional distribution of the target given the features only.

These labels are a bit problematic though. Any probabilistic model can be used to generate data, even if it is only for the target, so simply calling a model 'generative' isn't all that clarifying. And models that might be thought of as discriminative in a machine learning context might not be in others (e.g., Bayesian).

12.3 Reinforcement Learning

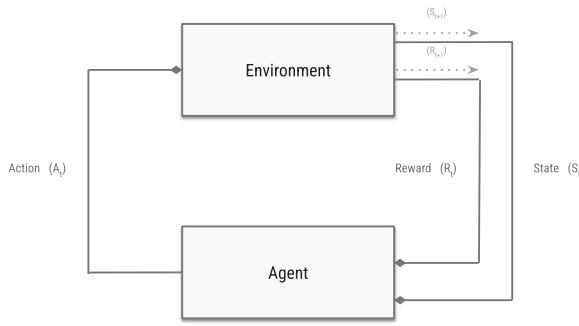


Figure 12.5: Reinforcement learning.

Reinforcement learning (RL) is a type of modeling approach that involves training an **agent** to make decisions in an **environment**. The agent receives feedback in the form of rewards or punishments for its actions, and the goal is to maximize its rewards over time by learning which actions lead to positive or negative outcomes. Typical data involves a sequence of states, actions, and rewards, and the agent learns a policy that maps states to actions. The agent learns by interacting with the environment, and the environment changes based on the agent's actions.

The agent's goal is to learn a **policy**, which is a set of rules that dictate which actions to take in different situations. The agent learns by trial and error, adjusting its policy based on the feedback it receives from the environment. The classic example is a game like chess or a simple video game. In these scenarios, the agent learns which moves (actions) lead to winning the game (positive reward) and which moves lead to losing the game (negative reward). Over time, the agent improves its policy to make better moves that increase its chances of winning.

One of the key challenges in reinforcement learning is balancing **exploration and exploitation**. Exploration is about trying new actions that could lead to higher rewards, while exploitation is about sticking to the actions that have already been found to give good rewards.

Reinforcement learning has many applications, including robotics, games, and autonomous driving, but there is little restriction on where it might be applied. It is also often a key part of some deep learning models, where reinforcement is supplied via human feedback or other means to an otherwise automatic modeling process.

12.4 Working with Specialized Data Types

While our focus in this book is on tabular data due to its ubiquity, there are many other types of data used for machine learning and modeling in general. This data often starts in a special format or must be considered uniquely. You'll often hear this labeled as 'unstructured', but that's probably not the best conceptual way to think about it, as the data is still structured in some way, sometimes in a strict format (e.g., images). Here we'll briefly discuss some of the other types of data you'll potentially come across.

12.4.1 Spatial

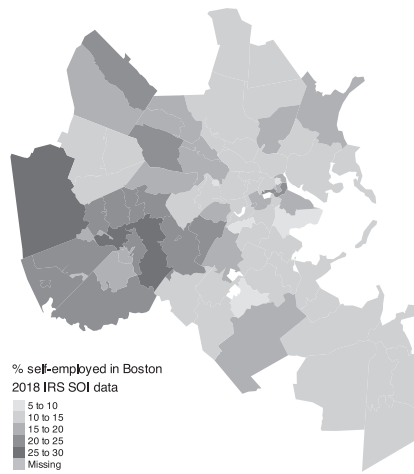


Figure 12.6: Spatial Demographic Data (code available from Kyle Walker)

Spatial data, which includes geographic and similar information, can be quite complex. It often comes in specific formats (e.g., shapefiles), and may require specialized tools to work with it. Spatial specific features may include continuous variables like latitude and longitude, or tracking data from a device

like a smartwatch. Other spatial features are more discrete, such as states or political regions within a country.

We could use these spatial features as we would others in the tabular setting, but we often want to take into account the uniqueness of a particular region, or the correlation of spatial regions. Historically, most spatial data can be incorporated into approaches like mixed models or generalized additive models, but in certain applications, such as satellite imagery, deep learning models are more the norm, and the models often transition into image processing techniques.

12.4.2 Audio

Figure 12.7: Sound wave.

Audio data is a type of time series data that is also the focus for many modeling applications. Think of the sound of someone speaking or music playing, as it changes over time. Such data is often represented as a waveform, which is a plot of the amplitude of the sound wave over time.

The goal of modeling audio data may include speech recognition, language translation, music generation, and more. Like spatial data, audio data is typically stored in specific formats and can be quite large by default. Also like spatial data, the specific type of data and research question may allow for a tabular format. In that case, the modeling approaches used are similar to those for other time series data.

Deep learning methods have proven very effective for analyzing audio data, and they can even create songs people actually like, even recently helping the Beatles to release one more song. Nowadays, you can generate an entire song in any genre you want, just by typing a text prompt!

12.4.3 Computer vision

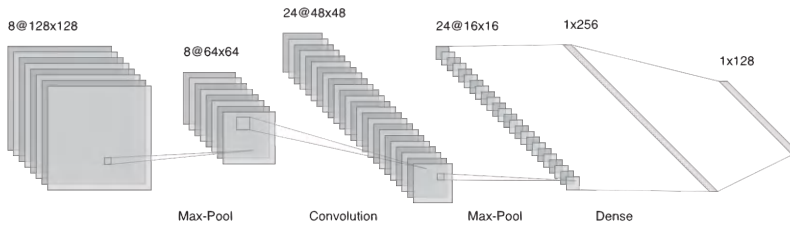


Figure 12.8: Convolutional neural network LeNet[4].

Computer vision involves a range of models and techniques for analyzing and interpreting image-based data. It includes tasks like image classification (labeling an image), object detection (finding the location of objects in an image), image segmentation (identifying the boundaries of objects in an image), and object tracking (following objects as they move over time).

Typically, your raw data is an image, which is represented as a matrix of pixel values. For example, each row of the matrix could be a grayscale value for a pixel, or it could be a three-dimensional array of Red, Green, and Blue (RGB) values for each pixel. The modeling goal is to extract features from the image data that can be used for the task at hand. For example, you might extract features that relate to color, texture, and shape. You can then use these features to train a model to classify images or whatever your task may be.

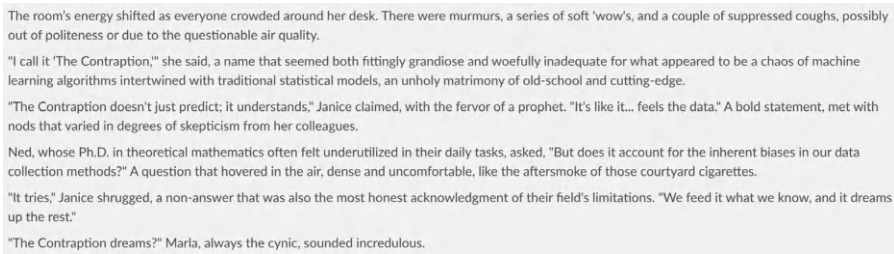
Image processing is a broad field with many applications. It is used in medical imaging, satellite imagery, self-driving cars, and more. And while it can be really fun to classify objects such as cats and dogs, or generate images from text and vice versa, it can be challenging due to the size of the data, issues specific to video/image quality, and the model complexity. Even if your base data is often the same or very similar across tasks, the model architecture and training process can vary widely depending on the task at hand.

These days we generally don't have to start from scratch though, as there are pretrained models that can be used for image processing tasks, which you can then fine-tune for your specific task. These models are often based on **convolutional neural networks** (CNNs), which are a type of deep learning model. CNNs are designed to take advantage of the spatial structure of images, and they use a series of convolutional layers to extract features from the image. These features are then passed through a series of fully connected layers to

[4]Image from Alex Lenail, LeNail (2024).

make a prediction. CNNs have been used to achieve state-of-the-art results on a wide range of image processing tasks and are the standard for many image processing applications. More recently, diffusion models, which seek to reconstruct images after successively adding noise to the initial input, have been shown to be quite effective for a wide range of tasks involving image generation.

12.4.4 Natural language processing



The room's energy shifted as everyone crowded around her desk. There were murmurs, a series of soft 'wow's, and a couple of suppressed coughs, possibly out of politeness or due to the questionable air quality.

"I call it 'The Contraption,'" she said, a name that seemed both fittingly grandiose and woefully inadequate for what appeared to be a chaos of machine learning algorithms intertwined with traditional statistical models, an unholy matrimony of old-school and cutting-edge.

"The Contraption doesn't just predict; it understands," Janice claimed, with the fervor of a prophet. "It's like it... feels the data." A bold statement, met with nods that varied in degrees of skepticism from her colleagues.

Ned, whose Ph.D. in theoretical mathematics often felt underutilized in their daily tasks, asked, "But does it account for the inherent biases in our data collection methods?" A question that hovered in the air, dense and uncomfortable, like the aftersmoke of those courtyard cigarettes.

"It tries," Janice shrugged, a non-answer that was also the most honest acknowledgment of their field's limitations. "We feed it what we know, and it dreams up the rest."

"The Contraption dreams?" Marla, always the cynic, sounded incredulous.

Figure 12.9: Partial GPT4 output from a prompt: Write a very brief short story about using models in data science. It should reflect the style of Donald Barthelme.

One of the hottest areas of modeling development in recent times regards **natural language processing**, as evidenced by the runaway success of models like ChatGPT. Natural language processing (NLP) is a field of study that focuses on understanding human language, and along with computer vision, is a very visible subfield of artificial intelligence. NLP is used in a wide range of applications, including language translation, speech recognition, text classification, and more. NLP is behind some of the most exciting modeling applications today, with tools that continue to amaze with their capabilities to generate summaries of articles, answer questions, write code, and even pass the bar exam with flying colors!

Early efforts in this field were based on statistical models, and then variations on things like PCA, but it took a lot of data pre-processing work to get much from those approaches, and results could still be unsatisfactory. More recently, deep learning models became the standard application, and there is no looking back in that regard due to their success. Current state-of-the-art models have been trained on massive amounts of data, even much of the internet, and require a tremendous amount of computing power. Thankfully, you don't have to train such a model yourself to take advantage of the results. Now you can simply use a pretrained model like GPT or Claude for your own tasks. In some cases, much of the trouble comes with just generating the best prompt to produce the desired results. However, the field and the models are evolving very rapidly, and, for those who don't have the resources of Google, Meta, or

OpenAI, things are getting easier to implement all the time. In the meantime, feel free to just play around with ChatGPT yourself.

12.5 Pretrained Models and Transfer Learning

Pretrained models are those that have been trained on a massive amount of data and can be used for a wide range of tasks, even on data they were not trained on! They are widely employed in image and natural language processing. The basic idea is that if you can use a model that was trained on the entire internet of text, why start from scratch? Computer vision models already understand things like edges and colors, so there is little need to reinvent the wheel when you know those features would be useful for your own task. These are viable in tasks where the inputs you are using are similar to the data the model was trained on, as is the case with images and text.

You can use a pretrained model as a starting point for your own model, and then **fine-tune** it for your specific task, and this is more generally called **transfer learning**. The gist is that you only need to train part of the model on your specific data, for example the last layer or two of a deep learning model. You 'freeze' the already learned weights for most of the model, while allowing the model to relearn the weights for the last layer(s) on your data. This can save a lot of time and resources, and it can be especially useful when you don't have a lot of data to train your model on.

12.5.1 Self-supervised learning

Self-supervised learning is a type of machine learning technique that involves training a model on a task that can be generated from the data itself. In this setting, there is no labeled data as such. For example, you might train a model to predict the next word in a sentence, and while you know what that word is, for purposes of modeling it is hidden. The idea is that the model learns to extract (represent) useful features from the data by trying to predict the missing information, which is imposed by a **mask** that hides parts of the data and may change from sample to sample[5]. We can then see how well our predictions match the targets that were masked.

This can be a useful approach when you don't have labeled data, or just when you don't have a lot of labeled data. Once trained, the model can be used as other pretrained models to predict other unlabeled data. Self-supervised

[5]If your 'mask' was a truly missing value rather than self-imposed, in the provided example self-supervised learning would essentially be the same as missing value imputation, but the latter doesn't sound as sexy and was already well established.

learning is often used in natural language processing but can be applied to other types of data as well.

12.6 Combining Models

It's important to note that the types of data used in ML and DL and their associated models are not mutually exclusive. For example, you might have a video that contains both audio and visual information pertinent to the task, or you might want to produce images from text inputs. When dealing with diverse data sources, you can combine different models to process them. This approach can range from simply adding extracted features to your dataset, to implementing complex **multimodal** deep learning architectures that handle multiple data types simultaneously.

Many computer vision, audio, natural language and other modeling approaches incorporate **transformers**. They are based on the idea of **attention**, which is a mechanism that allows the model to focus on certain parts of the input sequence and less on others. Transformers are used in many state-of-the-art models with different data types, such as those that combine text and images. The transformer architecture, although complex, underpins many of today's most sophisticated models, so it is worth being aware of even if it isn't your usual modeling domain.

As an example[6], we added a transformer-based approach to process the text reviews in the movie review dataset used in other chapters. We kept to the same basic data setup otherwise, and we ended up with notably better performance than the other models demonstrated, pushing toward 90% accuracy on test, even without fiddling too much with many hyperparameters. It's a good example of a case where we have standard tabular data, but we need to deal with additional data structure in a different way. By combining the approaches to obtain a final output for prediction, we obtained better results than we would with a single model. This won't always be the case, but keep it in mind when you are dealing with different data sources or types.

[6]We use a recently developed Python module torch_frame for this. Our approach is in a notebook available in the python chapter notebooks.

12.7 Artificial Intelligence

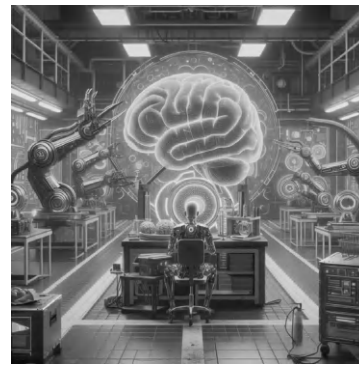


Figure 12.10: AI as envisioned by AI[7]

The prospect of combining models for computer vision, natural language processing, audio processing, and other domains can produce tools that mimic many aspects of what we call intelligence[8]. Current efforts in **artificial intelligence** (AI) produce models that can pass law and medical exams, create better explanations of images and text than average human effort, and produce conversation on par with humans. AI even helped to create this book! From code assistance to editing for clarity, and fleshing out ideas, AI has been a key part of the process of putting this book together.

In many discussions of ML and AI, many put ML as a subset of AI, but this is a bit off the mark from a modeling perspective in our opinion[9]. In terms of models, practically most of what we'd call modern AI almost exclusively employs deep learning models, particularly transformer architectures covered earlier, which enabled the leap from early, limited AI systems to today's more capable ones. Meanwhile, the ML approach to training and evaluating models

[7]Image created by MC using Dalle-2.

[8]It seems most discussions of AI in the public sphere haven't really defined intelligence very clearly in the first place, and the academic realm has struggled with the concept for centuries. This is why you can see people arguing about whether a model is 'intelligent', what is 'general intelligence', whether AI 'reasons', etc. The only place it makes sense to ask these questions is with an operational definition of these terms that works for (data) science, but that doesn't mean the definition would be satisfying to most people. Unfortunately, the leading researchers in AI keep changing the definitions as well, so at this point A*I is whatever we're currently deciding it is. Also, just like what happened with ML, many are now referring to potentially anything within the realm of data science as AI.

[9]In almost every instance of this we've seen, there isn't any actual detail or specific enough definitions provided to make the comparison meaningful to begin with, so don't take it too seriously.

can be used for any underlying model, from simple linear models to the most complex deep learning models, and whether the application falls under the domain of AI or not. Furthermore, statistical model applications have never seriously attempted what we might call AI.

If AI is some 'autonomous and general set of tools that attempt to engage the world in a human-like way or better', it's not clear why it'd be compared to ML in the first place. That's kind of like saying the brain is a subset of cognition. The brain does the work, much like ML does the modeling work with data, and gives rise to what we call cognition, but generally we would not compare the brain to cognition. We also wouldn't call ML a subset of climate science or medicine for similar reasons. They are domains in which it is used, much like the domain of artificial intelligence.

The main point is to not get too hung up on the labels, and focus on the modeling goal and how best to achieve it. Deep learning models, and machine learning in general, can be used for AI or non-AI settings, as we have seen for ourselves. And models used for AI still employ the *perspective* of the ML approach. The steps taken from data to model output are largely the same, as we are concerned with validation and generalization.

Many of the non-AI settings we use modeling for may well be things we can eventually rely on AI to do. At present though, the computational limits, and the amount of data that would be required for AI models to do well, or the ability of AI to deal with situations in which there is *only* small bits of data to train on, are still hindrances in many places we would like to use it. However, we feel it's likely these issues will eventually be overcome. Even then, a statistical approach may still have a place when the data is small.

In addition, as AI capabilities expand, ethical considerations become increasingly important. Issues of bias, privacy, transparency, job displacement, and security require careful thought alongside technical advancement. The models we build reflect our data, values, and assumptions - a reality that demands responsible development and deployment.

Artificial general intelligence (AGI) is the "holy grail" of AI, and like AI itself, it is not consistently defined. Generally, the idea behind AGI is the creation of some autonomous agent that can perform any task that a human can perform, many that humans cannot, and generalize abilities to new problems that have not even been seen yet. It seems we are getting closer to AGI all the time, especially with recent developments in **Agentic AI**, which are AI systems that can autonomously plan and execute sequences of actions to achieve goals, rather than just responding to prompts. But it's not yet clear when it will be achieved, or even what it will look like when it is, especially since no one has an agreed-upon definition of what intelligence is in the first place.

All that being said, to be perfectly honest, you may well be reading a history book. Given advancements just in the last couple years, it almost seems unlikely that the data science being performed 5 years from now will resemble much of how things are done today. We are already capable of making faster and further advancements in many domains due to AI, and it's likely that the next generation of data scientists will be able to do so even more easily. The future is here, and it is amazing. Buckle up!

12.8 Wrapping Up

We hope you've enjoyed the journey and have a better understanding of the core concepts. By now you also have a couple of modeling tools in hand, and you also have a good idea of where things can go. We encourage you to continue learning and experimenting with what you've seen, and to apply what you've learned to your own problems. The best way to learn is by doing, so don't be afraid to get your hands dirty and start building models!

12.8.1 The common thread

Even the most complex models can be thought of as a series of steps that go from input to output. In between, things can get very complicated, but often the underlying operations are the same ones you saw used with the simplest models. One of the key goals of any model is to generalize to new data, and this is the same no matter what type of data you're working with or what type of model you're using.

12.8.2 Choose your own adventure

The sky's the limit with machine learning modeling techniques, so go where your heart leads you, and have some fun! If you started here, feel free to go back to the linear model chapters for a more traditional and statistical modeling overview. Otherwise, continue on for an overview of a few more modeling topics, such as causal modeling (Chapter 13), data issues (Chapter 14), and things to avoid (Chapter 15).

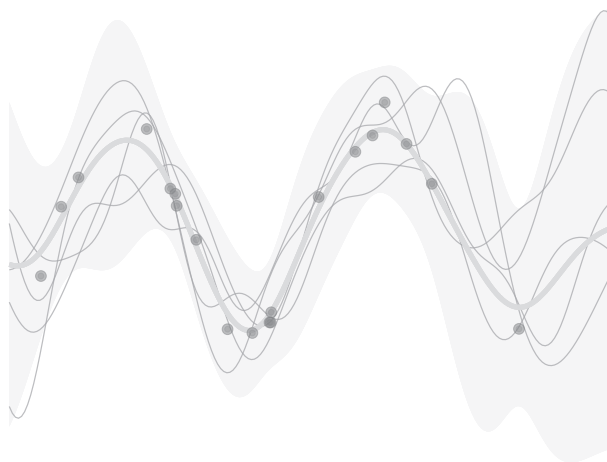
12.8.3 Additional resources

- Courses on ML and DL: FastAI (Howard (2024)), Coursera, edX, DeepLearning.AI, and many others are great places to get more formal training.
- Kaggle: Even if you don't compete, you can learn a lot from what others are doing.
- Unsupervised learning overview at Google

- Machine Learning and AI: Beyond the Basics (Raschka (2023b))
- A Visual Introduction to LLMs (3Blue1Brown (2024))
- Build a LLM from Scratch (Raschka (2023a))
- Visualizing Transformer Models (Vig (2019))
- Self-supervised learning: The dark matter of intelligence (LeCun and Misra (2021))

13

Causal Modeling



Causal inference is a very important topic in machine learning and statistical modeling approaches. It is also a very difficult one to understand well, or consistently, because *not everyone agrees on how to define a cause in the first place*. Our focus here is merely practical – we just want to discuss some of the modeling approaches commonly used when attempting to answer causal questions. But causal modeling in general is such a deep topic that we won't be able to go into as much detail as it deserves. However, we will try to give you a sense of the landscape and some of the key ideas.

13.1 Key Ideas

- No model can tell you whether a relationship is causal or not. Causality is inferred, not proven, based on the available evidence.
- The same models could be used for similar data settings to answer a causal question or a purely predictive question. A key difference is in the interpretation of the results.

- Experimental design, such as randomized control trials, are considered the gold standard for causal inference. But the gold standard is often not practical, and not without its limitations even when it is.
- Causal inference is often done with observational data, which is often the only option, and that's okay.
- Counterfactual thinking is at the heart of causal inference but can be useful for all modeling contexts.
- Several models exist which are typically employed to answer a more causal-oriented question. These include graphical models, uplift modeling, and more.
- Interactions are the norm for most modeling scenarios, while causal inference generally regards a single effect. If an effect varies depending on other features, you should be cautious trying to aggregate your results to a single effect, since that effect would be potentially misleading.

13.1.1 Why it matters

Often we need a precise statement about the feature-target relationship, not just a declaration that there is 'some' relationship. For example, we might want to know how well a drug works and for whom, or show that an advertisement results in a certain amount of new sales. We generally need to know whether the effect is real, and the size of the effect, and often, the uncertainty in that estimate.

Causal modeling is, like machine learning, more of an approach than a specific model, and that approach may involve the design or implementation of models we've already seen, but conducted in a different way to answer the key question. Without more precision in our understanding, we could miss the effect, or overstate it, and make bad decisions as a result.

13.1.2 Helpful context

This section is pretty high level, and we are not going to go into much detail here, so even just some understanding of correlation and modeling would likely be enough.

13.2 Prediction and Explanation Revisited

We introduced the idea of prediction and explanation in the context of linear models in Section 3.4.3, and it's worth revisiting here. One attribute of a causal model is an intense focus on the explanatory power of the model. We want to demonstrate that there is a relationship between (usually) a single feature and

the target, and we want to know the precise manner of this relationship as much as possible. Even if we use complex models, the endeavor is to explain the specifics.

Let's say that we used some particular causal modeling approach to explain a feature-target relationship in a classification setting. We have 10,000 observations, and the baseline rate of the target is about ~50%. We have a model that predicts the target `y` based on the feature of interest `x`, and we may have used some causal technique like propensity score weighting or some other approach to help control for confounding (we'll discuss these later).

The coefficient, though small with an odds ratio of 1.05, is statistically significant (take our word for it), so we have a slight positive relationship. Under certain settings such as this, where we are interested in causal effects and where we have controlled for various other factors to obtain this result, we might be satisfied with interpreting this relationship as is.

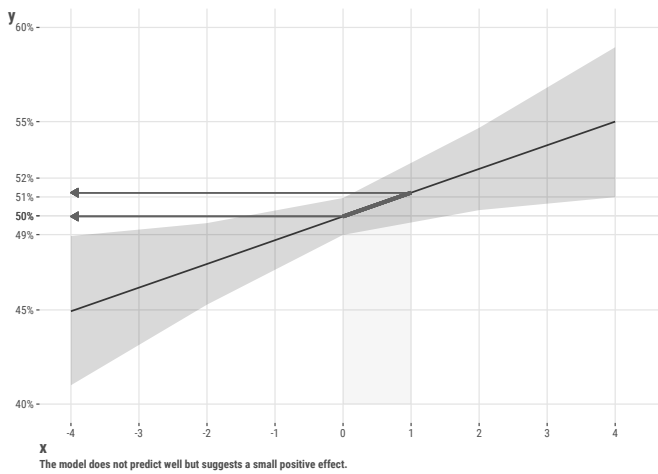


Figure 13.1: Results from a hypothetical causal model.

But if we are interested in predictive performance, we would be disappointed with this model. It predicts the target at about the same rate as guessing, even with the data it's fit on, and does even worse with new data. Even the effect as shown is quite small by typical standards, as it would take a standard deviation change in the feature to get a ~1% change in the probability of the target (x is standardized).

If we are concerned solely with explanation, we now would want to ask ourselves first if we can trust our result based on the data, model, and various issues that went into producing it. If so, we can then see if the effect is large enough

to be of interest, and if the result is useful in making decisions[1]. It may very well be, maybe the target concerns the rate of survival, where any increase is worthwhile. Or perhaps the data circumstances demand such interpretation, because it is costly to obtain more. For more exploratory efforts however, this sort of result would likely not be enough to come to any strong conclusion, even if explanation is the only goal.

As another example, consider the world happiness data we've used in previous demonstrations. We want to explain the association of country level characteristics and the population's happiness. We likely aren't going to be as interested in predicting next year's happiness score, but rather what attributes are correlated with a happy populace in general. For another example, in the U.S., we might be interested in specific factors related to presidential elections, of which there are relatively very few data points. In these cases, explanation is the focus, and we may not even need a model at all to come to our conclusions.

So we can see that in some settings we may be more interested in understanding the underlying mechanisms of the data, and in others we may be more interested in predictive performance. However, the distinction between prediction and explanation in the end is a bit problematic, not the least of which is that we often want to do both.

Although it's often implied as such, *prediction is not just what we do with new data*. It is the very means by which we get any explanation of effects via coefficients, marginal effects, visualizations, and other model results. Additionally, when the focus is on predictive performance, if we can't explain the results we get, we will typically feel dissatisfied and may still question how well the model is actually doing.

Here are some ways we might think about different modeling contexts:

- **Descriptive Analysis**: Here we have an exploration of data with no modeling focus. We'll use descriptive statistics and visualizations to help us understand what's going on. An end product may be an infographic or a highly visual report. Even here, we might use models to aid visualizations, or otherwise to help us understand the data better, but their specific implementation or result is not of much interest.
- **Exploratory Modeling**: When using models for exploration, focus should probably be on both prediction and explanation. The former can help inform

[1]This is a contrived example, but it is definitely something that you might see in the wild. The relationship is weak, and though statistically significant, the model can't predict the target well at all. The **statistical power** is actually decent in this case, roughly 70%, but this is mainly because the sample size is so large, and it is a very simple model setting. The same coefficient with a base rate of around 5% would have a power of around 20%. This is a common issue, and it's why we always need to be careful about how we interpret our models. In practice, we would generally need to consider other factors, such as the cost of a false positive or false negative, or the cost of the data and running the model itself, to determine if the model is worth using.

the strength of the results for future exploration, while the latter will often provide useful insights.

- **Causal Modeling**: Here the focus is on understanding causal effects. We focus on explanation, and prediction on the current data. We may very well be interested in predictive performance also, and we often are in industry.
- **Generalization**: When our goal is generalizing to unseen data as we have discussed elsewhere, the focus is mostly on predictive performance[2], as we need something to help us predict things in the future. This does not mean we can't use the model to understand the data though, and explanation could still possibly be as important depending on the context.

Depending on the context, we may be more interested in explanation or predictive performance, but in practice we often want both. It is crucial to remind yourself why you are interested in the problem, what a model is capable of telling you about it, and to be clear about what you want to get out of the result.

13.3 Classic Experimental Design

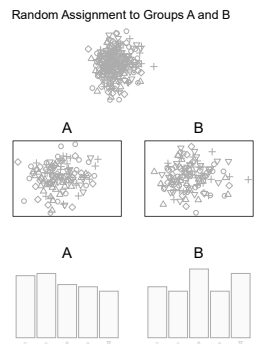


Figure 13.2: Random assignment.

Many are familiar with the basic idea of an experiment, where we have a **treatment** group and a **control** group, and we want to measure the difference between the two groups. The 'treatment' could regard a new drug, a marketing campaign, or a new app's feature. If we randomly assign our observational

[2]In causal modeling, there is the notion of **transportability**, which is the idea that a model can be used in, or generalize to, a different setting than it was trained on. For example, you may see an effect for one demographic group and want to know whether it holds for another. It is closely related to the notion of external validity and is also related to the concepts we've hit on in our discussion of interactions (Section 9.2).

units to the two groups, say, one that gets the new app feature and the other doesn't, we can be more confident that the two groups are essentially the same aside from the treatment. Furthermore, any difference we see in the outcome, for example, customer satisfaction with the app, is probably due to the treatment.

This is the basic idea behind a **randomized control trial** (RCT). We can randomly assign the groups in a variety of ways, but you can think of it as flipping a coin, and assigning each sample to the treatment when the coin comes up on one side, and to the control when it comes up on the other. The idea is that the only difference between the two groups is the treatment, and so any difference in the outcome can be attributed to the treatment. This is visualized in Figure 13.2, where the color/shapes represent different groups that are the same. Their distribution is roughly similar after assignment to the treatment groups and would become more so with more data.

13.3.1 Analysis of experiments

Many who have taken a statistics course have been exposed to the simple **t-test** to determine whether two groups are different. For many, this is their first introduction to statistical modeling. The t-test tells us whether the difference in means between the two groups is *statistically* significant. However, it definitely *does not* tell us whether the treatment itself caused the difference, whether the effect is large, nor whether the effect is real, or even if the treatment is a good idea to do in the first place. It just tells us whether the two groups are statistically different.

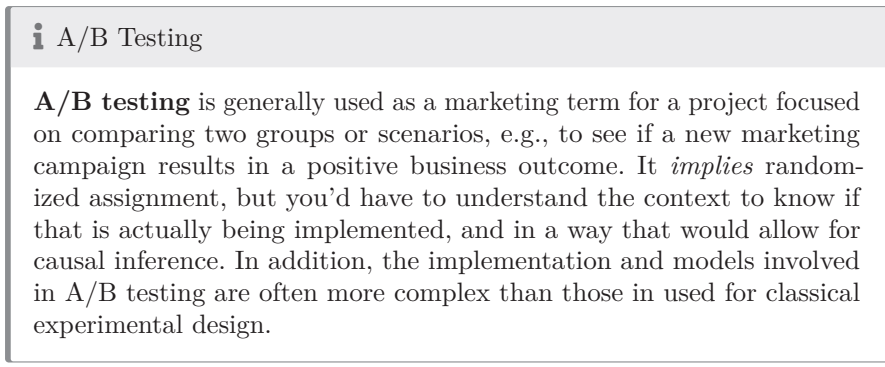
It turns out that a t-test is just a linear regression model. It's a special case of linear regression where there is only one independent variable, and it is a categorical variable with two levels. The coefficient from the linear regression would tell you the mean difference of the outcome between the two groups. Under the same conditions, the t-statistic from the linear regression and the t-test from a separate function would have identical statistical results.

Analysis of variance (ANOVA), allows the t-test to be extended to more than two groups, and multiple features, and is also commonly employed to analyze the results of experimental design settings. But ANOVA is still just a linear regression. Even when we get into more complicated design settings such as repeated measures and mixed design, it's still just a linear model, we'd just be using mixed models (Section 9.3). In general, we're going to use similar tools to analyze the results of our experiments as we would for other modeling settings.

If linear regression didn't suggest any notion of causality to you before, it shouldn't now either. The model is *identical* whether there was an experimental design with random assignment or not. The only difference is that the data was collected in a different way, and the theoretical assumptions and motivations

are different. Even the statistical assumptions are the same whether you use random assignment, or whether there are more than two groups, or whether the treatment is continuous or categorical.

Experimental design[3] can give us more confidence in the causal explanation of model results, whatever model is used, and this is why we like to use it when we can. It helps us control for the unobserved factors that might otherwise be influencing the results. If we can be fairly certain the observations are essentially the same *except* for the treatment, then we can be more confident that the treatment is the cause of any differences we see, and be more confident in a causal interpretation of the results. But it doesn't change the model itself, and the results of a model don't prove a causal relationship on their own. Your experimental study will also be limited by the quality of the data, and the population it generalizes to. Even with strong design and modeling, if care isn't taken in the modeling process to even assess the generalization of the results (Section 10.4), you may find they don't hold up[4].



[3]Note that experimental design is not just any setting that uses random assignment, but more generally how we introduce *control* in the sample settings.

[4]Many experimental design settings involve sometimes very small samples due to the cost of the treatment implementation and other reasons. This often limits exploration of more complex relationships (e.g., interactions), and it is relatively rare to see any assessment of performance generalization. It would probably worry many to know how many important experimental results are based on p-values with small data, and this is the part of the problem seen with the replication crisis in science.

13.4 Natural Experiments

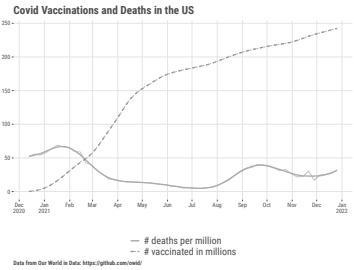


Figure 13.3: Covid vaccinations and deaths in the US.

As we noted, random assignment or a formal experiment is not always possible or practical to implement. But sometimes we get to do it anyway, or at least we can get pretty close! Occasionally, the world gives us a **natural experiment**, where the assignment to the groups is essentially random, or where there is clear break before and after some event occurs, such that we examine the change as we would in pre-post design.

The COVID-19 pandemic provides an example of a natural experiment. The pandemic introduced sudden and widespread changes that were not influenced by individuals' prior characteristics or behaviors, such as lockdowns, remote work, and vaccination campaigns. The randomness in the timing and implementation of these changes allows researchers to compare outcomes before and after the policy implementation or pandemic, or between different regions with varying policies, to infer causal effects.

For instance, we could compare states or counties that had mask mandates to those that didn't at the same time or with similar characteristics. Or we might compare areas that had high vaccination rates to those nearby that didn't. But these still aren't true experiments. So we'd need to control for as many additional factors that might influence the results, like population density, age, wealth and so on, and eventually we might still get a pretty good idea of the causal impact of these interventions.

13.5 Causal Inference

While we all have a natural intuition about causality, it can actually be a fairly elusive notion to grasp. Causality is a very old topic, philosophically dating back millennia, and more formally hundreds of years. Random assignment is a relatively new idea, say 150 years old, and was posited even before Wright, Fisher, and Neyman, and the 20th century rise of statistics. But with stats and random assignment, we had a way to start using models to help us reason about causal relationships. Pearl and others came along to provide an algorithmic perspective from computer science, and economists like Heckman also got into the game too. We were even using programming approaches to do causal inference back in the 1970s! Eventually most scientific academic disciplines were well acquainted with causal inference in some fashion, and things have been progressing along for some time.

Because of its long history, causal inference is a broad field, and there are many ways to approach it. We've already discussed some of the basics, but there are many other ways to reason about causality. And of course, we can use models to help us understand the causal effects we are interested in.

13.5.1 Key assumptions of causal inference

Causal inference at its core is the process of identifying and estimating causal effects. But like other scientific and modeling endeavors, it relies on several key assumptions to identify and estimate those effects. The main assumptions include:

- **Consistency**: The *potential* outcome under the observed treatment is the same as the *observed* outcome. This suggests there is *no interference* between units, and that there are *no hidden variations of the treatment.*
- **Exchangeability**: The treatment assignment is independent of the potential outcomes, given the observed covariates. In other words, the treatment assignment is as good as random after conditioning on the covariates. This is often referred to as *no unmeasured* ***confounding***.
- **Positivity**: Every individual has a positive probability of receiving each treatment level.

It can be difficult to meet these assumptions, and there is not always a clear path to a solution. As an example, say we want to assess a new curriculum's effect on student performance. We can randomly assign students, but they can interact with one another both in and outside of the classroom. Those who receive the treatment may be more likely to talk to one another, and this could affect the outcome, enhancing its effects if it is beneficial. This would

violate our assumption of no interference between units, and we'd need to maybe choose an alternative design or outcome to account for this.

The following demonstrates a common assumption that is regularly guarded against in causal modeling – confounding. The confounder, `U`, is a variable that affects both treatment `X` and target `Y`. We'll generate some synthetic data with a confounder, and fit two models, one with the confounder and one without. We'll compare the coefficients of the feature of interest in both models.

Python

```
from numpy.random import normal as rnorm
import pandas as pd
import statsmodels.api as sm

def get_coefs(n = 100, true = 1):
    U = rnorm(size=n)                 # Unmeasured confounder
    X = 0.5 * U + rnorm(size=n)       # Treatment influenced by U
    Y = true * X + U + rnorm(size=n)  # Outcome influenced by X and U

    data = pd.DataFrame({'X': X, 'U': U, 'Y': Y})

    # Fit a linear regression model with and
    # without adjusting for the unmeasured confounder
    model = sm.OLS(data['Y'], sm.add_constant(data['X'])).fit()
    model2 = sm.OLS(data['Y'], sm.add_constant(data[['X', 'U']])).fit()
    return model.params['X'], model2.params['X']

def simulate_confounding(nreps = 100, n = 100, true=1):
    results = []
    for _ in range(nreps):
        results.append(get_coefs(n, true))

    results = np.mean(results, axis=0)

    return pd.DataFrame({
        'true': true,
        'estimate_1': results[0],
        'estimate_2': results[1],
    }, index=['X']).round(3)


simulate_confounding(n=1000, nreps=500)
```

R

```
get_coefficients = function(n = 100, true = 1) {
    U = rnorm(n)                 # Unmeasured confounder
    X = 0.5 * U + rnorm(n)       # Treatment influenced by U
    Y = true * X + U + rnorm(n)  # Outcome influenced by X and U

    data = data.frame(X = X, Y = Y)

    # Fit a linear regression model with and
    # without adjusting for the unmeasured confounder
    model = lm(Y ~ X, data = data)
    model2 = lm(Y ~ X + U, data = data)
    c(coef(model)['X'], coef(model2)['X'])
}

simulate_confounding = function(nreps, n, true) {
    results = replicate(nreps,  get_coefficients(n, true))

    results = rowMeans(results)

    data.frame(
        true = true,
        estimate_1 = results[1],
        estimate_2 = results[2]
    )
}

simulate_confounding(nreps = 500, n = 1000, true = 1)
```

Results suggest that the coefficient for `X` is different in the two models. If we don't include the confounder, the feature's relationship with the target is biased upward. The nature of the bias ultimately depends on the relationship between the confounder and the treatment and target, but in this case it's pretty clear!

Table 13.1: Coefficients with and without the Confounder

true	with conf.	no conf.
1.00	1.00	1.40

Though this is a simple demonstration, it shows why we need to be careful in our modeling and analysis, and if we are interested in causal relationships, we need to be aware of our assumptions and help make them plausible. If we

suspect something is a confounder, we can include it in our model to get a more accurate estimate of the effect of the treatment.

More generally, with causal approaches to modeling, we are expressly interested in interpreting the effect of one feature on another, and we are interested in the mechanisms that bring about that effect. We are not just interested in the mere correlation between variables, or just predictive capabilities of the model. As we'll see though, we can use the same models we've seen already, but we'll need these additional considerations to draw causal conclusions.

13.6 Models for Causal Inference

We can use many modeling approaches to help us reason about causal relationships, and this can be both a blessing and a curse. Our models can be more complex, and we can use more data, which can potentially give us more confidence in our conclusions. But we can still be easily fooled by our models, as well as by ourselves. We'll need to be careful in how we go about things, but let's see what some of our options are!

Any model can potentially be used to answer a causal question, and which one you use will depend on the data setting and the question you are asking. The following covers a few models that might be seen in various academic and professional settings.

13.6.1 Linear regression

Yep, linear regression. The old standby is possibly the mostly widely used model for causal inference, historically speaking and even today. We've seen linear regression as a kind of graphical model in Figure 3.2, and in that sense, it can serve as the starting point for those that many consider to be true causal models. It can also be used as a baseline model for other more complex causal model approaches.

Linear regression can potentially tell us for any particular feature what that feature's relationship with the target is, holding the other features constant. This **ceteris paribus** interpretation – 'all else being equal' – already gets us into a causal mindset. If we had randomization and no confounding, and the feature-target relationship was linear, we could interpret the coefficient of the feature as the causal effect.

However, your standard linear model doesn't care where the data came from or what the underlying structure *should* be. It only does what you ask of it, and will tell you about group differences whether they come from a randomized experiment or not. For example, as we saw earlier, if potential confounders

aren't included, the estimated effect could be biased[5]. It also cannot tell you whether X effects Y or vice versa. So linear regression by itself cannot save us from the difficulties of causal inference, nor really can be considered a causal model. But it can be useful as a starting point in conjunction with other approaches.

> **i** Weighting and Sampling Methods
>
> Common techniques for traditional statistical models used for causal inference include a variety of **weighting** or **sampling** methods. These methods are used to adjust the data so that the treatment groups are more similar, and a causal effect can be more accurately estimated. Sampling methods include techniques such as **stratification** and **matching**, which focus on the selection of the sample as a means to balance treatment and control groups. Weighting methods include **inverse probability weighting** and **propensity score weighting**, which focus on adjusting the weights of the observations to make the groups more similar. They have extensions to continuous treatments as well.
>
> Sampling and weighting methods are not models themselves, and potentially can be used with just about any model that attempts to estimate the effect of a treatment, or balance the data in some fashion. An nice overview of using such methods vs. standard regression/ML can be found on Cross Validated.

13.6.2 Graphical and structural equation models

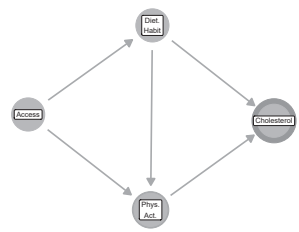


Figure 13.4: Causal DAG.

Graphical and Structural Equation Models (SEM) are flexible approaches to regression and classification, and they have one of the longest

[5]A reminder that a conclusion of 'no effect' is also a causal statement, and it too can be a biased one. Also, you can come to the same *practical* conclusion with a biased estimate as with an unbiased one.

histories of formal statistical modeling, dating back over a century[6]. As an initial example, Figure 13.4 shows a *directed acyclic graph* (DAG) that represents a causal model. The arrows indicate the direction of the causal relationship, and each node is a feature or target, and some features are influenced by others.

In that graph, our focal treatment, or 'exposure', is physical activity, and we want to see its effect on a health outcome like cholesterol levels. However, dietary habits would affect both the outcome and affect how much physical activity one does. Both dietary habits and physical activity may in part reflect access to healthy food. The target in question does not affect any other nodes, and in fact the causal flow is in one direction, so there is no cycle in the graph (i.e., it is 'acyclic').

One thing to note relative to the other graphical model depictions we've seen is that the arrows directly flow to a target or set of targets, as opposed to just producing an 'output' that we then compare with the target. In graphical causal models, we're making clear the direction and focus of the causal relationships, i.e., the causal structure, as opposed to the model structure. Also, in graphical causal models, the effects for any given feature are adjusted for the other features in the model in a particular way, so that we can think about them in isolation, rather than as a collective set of features that are all influencing the target[7].

Structural equation models are widely employed in the social sciences and education, and they are often used to model both observed and *latent* variables (Section 14.9), with either serving as features or targets[8]. They are also used to model causal relationships, to the point that historically they were even called 'causal graphical models' or 'causal structural models'. SEMs are actually a special case of the graphical models just described, which are more common in non-social science disciplines. Compared to other graphical modeling techniques

[6]Sewall Wright is credited with what would be called **path analysis** back in the 1920s, which is a precursor to and part of SEM and a form of graphical model.

[7]If we were to model this in an overly simple fashion with linear regressions for any variable with an arrow to it, you could say physical activity and dietary habits would basically be the output of their respective models. It isn't that simple in practice though, such that we can just run separate regressions and feed in the results to the next one, though that's how they used to do it back in the day. We have to take more care in how we adjust for all features in the model, as well as correctly account for the uncertainty if we do take a multistage approach.

[8]Your authors have to admit some bias here, but we hope the presentation for SEM is balanced. We've spent a lot of our past dealing with SEMs, and almost every application we saw had too little data and was grossly overfit. Many SEM programming approaches even added multiple ways to overfit the data even further, and it is difficult to trust the results reported in many papers that used them. But that's not the fault of SEM in general. Like any model it can be a useful tool when used correctly, and it can help answer causal questions. But it can easily be misused by those not familiar with its assumptions and limitations.

like DAGs, SEMs will typically have more assumptions, and these are often difficult to meet[9].

The following shows a relatively simple SEM, a latent variable **mediation model** involving social support and self-esteem, and with depression as the outcome of interest (Figure 13.5). Each latent variable has three observed measures, e.g., item scores collected from a psychological inventory or personal survey. The observed variables are *caused* by the latent, i.e., *unseen* or *hidden*, variables. In other words, the observed item score is a less than perfect reflection of the true underlying latent variable, which is what we're really interested in. The effects of the latent constructs of social support and self-esteem on depression may be of equal interest in this setting. For social support, we'd be interested in the direct effect on depression, as well as the indirect effect through self-esteem.

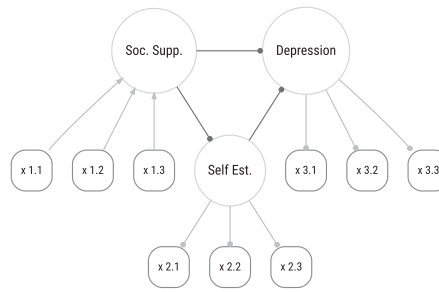


Figure 13.5: SEM with latent and observed variables.

Formal graphical models provide a much richer set of tools for controlling various confounding, interaction, and indirect effects than simpler linear models. For this reason, they can be very useful for causal inference. It may be cautionary to note that models like linear regression can be seen as a special case, and we know that linear regression by itself is not a causal model. So in order for these tools to provide valid causal estimates, they need to be used in a way that is consistent with the assumptions of both the underlying causal model as well as the model estimation approach.

Causal Language

It's often been suggested that we keep certain phrasing, for example, feature X has an *effect* on target Y, only for the causal model setting. But the model we use can only tell us that the data is consistent with

[9]VanderWeele (2012) provides a nice overview of the increased assumptions of SEM relative to other methods.

> the effect we're trying to understand, not that it actually exists. In everyday language, we often use causal language whenever we think the relationship is or should be causal, and that's fine, and we think that's okay in a modeling context too, as long as you are clear about the limits of your generalizability.

13.6.3 Counterfactual thinking

When we think about causality, we really ought to think about **counterfactuals**. What would have happened if I had done something different? What would have happened if I had done something sooner rather than later? What would have happened if I had done nothing at all? It's natural to question our own actions in this way, but we can think like this in a modeling context too. In terms of our treatment effect example, we can summarize counterfactual thinking as:

> The question is not whether there is a difference between A and B but whether there would still be a difference if A *was* B and B *was* A.

This is the essence of counterfactual thinking. It's not about whether there is a difference between two groups, but whether there would still be a difference if those in one group had actually been treated differently. In this sense, we are concerned with the **potential outcomes** of the treatment, however defined.

Here is a more concrete example:

- Roy is shown ad A and buys the product.
- Pris is shown ad B and does not buy the product.

What are we to make of this? Which ad is better? **A** seems to be, but maybe Pris wouldn't have bought the product if shown that ad either, and maybe Roy would have bought the product if shown ad **B** too! With counterfactual thinking, we are concerned with the potential outcomes of the treatment, which in this case is whether or not to show the ad.

Let's say ad A is the new one, i.e., our treatment group, and B is the status quo ad, our control group. Without randomization, our real question can't be answered by a simple test of whether means or predictions are different among the two groups, as this estimate would be biased if the groups are already different in some way to start with. The real effect is, for those who saw ad A, what the difference in the outcome would be if they hadn't seen it.

From a prediction standpoint, we can get an initial estimate straightforwardly. We demonstrated counterfactual predictions before in Section 5.6, but we can revisit it briefly here. For those in the treatment, we can just plug in their feature values with treatment set to ad A. Then we just make a prediction with treatment set to ad B. This approach is basically the **S-Learner** approach

to meta-learning, which we'll discuss in a bit, as well as a simple form of **G-computation**, widely used in causal inference.

Python

```
model.predict(X.assign(treatment = 'A')) -
    model.predict(X.assign(treatment = 'B'))
```

R

```
predict(model, X |> mutate(treatment = 'A')) -
    predict(model, X |> mutate(treatment = 'B'))
```

With counterfactual thinking explicitly in mind, we can see that the difference in predictions is the difference in the potential outcomes of the treatment. This is a very simple demo to illustrate how easy it is to start getting some counterfactual results from our models. But it's typically not quite that simple in practice, and there are many ways to get this estimate wrong as well. As in other circumstances, the data and our assumptions about the problem can potentially lead us astray. But, assuming those aspects of our modeling endeavor are in order, this is one way to get an estimate of a causal effect.

13.6.4 Uplift modeling

The counterfactual prediction we just did provides a result that can be called the **uplift** or **gain** from the treatment, particularly when compared to a baseline metric. **Uplift modeling** is a general term applied to models where counterfactual thinking is at the forefront, especially in a marketing context. Uplift modeling is not a specific model per se, but any model that is used to answer a question about the potential outcomes of a treatment. The key question is what is the gain, or uplift, in applying a treatment vs. the baseline? Typically any statistical model can be used to answer this question, and often the model is a classification model, for example, whether Roy from the previous section bought the product or not.

It is common in uplift modeling to distinguish certain types of individuals or instances, and we think it's useful to extend this to other modeling contexts as well. In the context of our previous example, they are:

- **Sure things**: those who would buy the product whether or not shown the ad.
- **Lost causes**: those who would not buy the product whether or not shown the ad.
- **Sleeping dogs**: those who would buy the product if not shown the ad, but not if they are shown the ad. Also referred to as the 'Do not disturb' group!

- **Persuadables**: those who would buy the product if shown the ad, but not if not shown the ad.

We can generalize these conceptual groups beyond the marketing context to any treatment effect we might be interested in. So it's worthwhile to think about which aspects of your data could correspond to these groups. One of the additional goals in uplift modeling is to identify persuadables for additional treatment efforts, and to avoid wasting money on the lost causes. But to reach such goals, we have to think causally first!

> **i** Uplift Modeling in R and Python
>
> There are more widely used tools for uplift modeling and meta-learners in Python than in R, but there are some options in R as well. In Python you can check out causalml and sci-kit uplift for some nice tutorials and documentation.

13.6.5 Meta-Learners

Meta-learners are used in machine learning contexts to assess potentially causal relationships between some treatment and outcome. The core model can actually be any kind you might want to use, but in which extra steps are taken to assess the causal relationship. The most common types of meta-learners are:

- **S-learner**: **s**ingle model for both groups; predict the (counterfactual) difference as when all observations are treated vs. when all are not, similar to our previous demonstrations of counterfactual predictions.
- **T-learner**: **t**wo models, one for each of the control and treatment groups respectively; get predictions as if all observations are 'treated' (i.e., using the treatment model) vs. when all are 'control' (using the control model), and take the difference.
- **X-learner**: a more complicated modification to the T-learner using a multi-step approach.
- **R-learner**: also called (Double) Debiased ML. An approach that uses a **r**esidual-based model to adjust for the treatment effect[10].

Some variants of these models exist also. As elsewhere, the key idea is to use the model to predict the potential outcomes of the treatment levels to estimate

[10] As a simple overview, think of it this way with Y outcome, T treatment and X confounders/other features. Y and T are each regressed on X via some ML model, and the residuals from both are used in a subsequent model, $Y_{res} \sim T_{res}$, to estimate the treatment effect. Or, if you know how path analysis works, or even standard linear regression, it's pretty much just that with ML. As an exercise, start with a linear regression for the target on all features, then just do a linear regression for a chosen focal feature predicted by the nonfocal features. Next, regress the target on the nonfocal features. Finally, just do a linear regression with the residuals from both models. The resulting coefficient will be what you started with for the focal feature in the first regression.

the causal effect. Most models traditionally used in a machine learning context, e.g., random forests, boosted trees, or neural networks, are not designed to accurately estimate causal effects, nor correctly estimate the uncertainty in those effects. Meta-learners attempt to address the issue with regard to the effect, but you'll typically still have your work cut out for you to understand the uncertainty in that effect.

> **i** Meta-Learners vs. Meta-Analysis
>
> Meta-learners are not to be confused with **meta-analysis**, which is also related to understanding causal effects. Meta-analysis attempts to combine the results of multiple *studies* to get a better estimate of the true effect. The studies are typically conducted by different researchers and in different settings. The term **meta-learning** has also been used to refer to what is more commonly called **ensemble learning**, the approach used in random forests and boosting. It is also probably used by other people who don't bother to look things up before naming their technical terms.

13.6.6 Other models used for causal inference

Note that there are many models that would fall under the umbrella of causal inference. But typically these models are only a special application of some of the ones we've already become well acquainted with, so you should feel good about trying them out. Here are a few you might come across specific to the causal modeling domain:

- G-computation, doubly robust estimation, targeted maximum likelihood estimation[11]
- Marginal structural models[12]
- Instrumental variables and two-stage least squares[13]
- Propensity score matching/weighting
- Regression discontinuity design[14]
- Difference-in-differences[15]

[11] The G-computation approach and S-learners are essentially the same approach, but came about from different domain contexts.

[12] Very common in epidemiology, and a nice introduction can be found in Robins, Hernán, and Brumback (2000).

[13] Instrumental variables are used in econometrics and are a way to get around the problem of unmeasured confounding.

[14] Regression discontinuity design is a quasi-experimental design that is used when comparing an outcome on either side of a threshold, such as a cutoff for a program or policy. The idea is that those just above the threshold are similar to those just below, and the difference in the outcome can be attributed to the program or policy. This is at its core just a pre-post type of analysis.

[15] Difference-in-differences just involves an interaction of a treatment with something else, typically time.

- Mediation/moderation analysis[16]
- Meta-analysis
- Bayesian networks

In general, any modeling technique can be employed as part of a causal modeling endeavor. To actually make causal statements, you'll generally need to ensure that the assumptions for those claims are tenable.

13.7 Wrapping Up

We've been pretty loose in our presentation here, and we intentionally glossed over many details with causal modeling. Our main goal is to give you some idea of the domain, but more so the models used and things to think about when you want to answer a causal question with your data.

Models used in statistical analysis and machine learning are not causal models, but when we take a causal model from the realm of ideas and apply it to the real world, a causal model becomes a statistical/ML model with more assumptions, and with additional steps taken to address those assumptions[17]. These assumptions are required in order to make stronger causal statements, but neither the assumptions, data, nor model can prove that the underlying theory is causally correct. Things like random assignment, sampling, a complex model and good data can possibly help the situation, but they can't save you from a fundamental misunderstanding of the problem, or data that may still be consistent with that misunderstanding. Nothing about employing a causal model inherently makes better predictions either.

Causal modeling is hard, and most of the difficulty lies outside of the realm of models and data. The model implemented reflects the causal theory, which can be a correct or incorrect idea about how the world works. In the end, the main thing is that when we want to make causal statements, we'll make do with what data we have, and be careful that we rule out some of the other obvious explanations and issues. The better we can control the setting, or the better we can do things from a modeling standpoint, the more confident we can be in making causal claims. Causal modeling is really an exercise in *reasoning*, which makes it such an interesting endeavor!

[16] Mediation/moderation are special applications of structural equation modeling.

[17] Gentle reminder that making an assumption does not mean the assumption is correct, or even provable.

13.7.1 The common thread

Engaging in causal modeling may not even require you to learn any new models, but you will typically have to do more to be able to make causal statements. The key is to think about the problem in a different way, and to be more clear and careful about the assumptions you are making. You may need to do more work to ensure that your data and chosen model are consistent with the assumptions you are making.

13.7.2 Choose your own adventure

From here you might revisit some of the previous models and think about how you might use them to answer a causal question. You might also look into some of the other models we've mentioned here and see how they are used in practice via the additional resources.

13.7.3 Additional resources

We have only scratched the surface here, and there is a lot more to learn. Here are some resources to get you started:

- Causal Inference in R, Barrett, McGowan, and Gerke (2024)[18]
- Causal Inference The Mixtape, Cunningham (2023)
- Causal Inference for the Brave and True, Facure Alves (2022)
- Applied Causal Inference Powered by ML and AI,Chernozhukov et al. (2024)
- Metalearners for estimating heterogeneous treatment effects using machine learning, Künzel et al. (2019)
- The C-Word, Hernán (2018)

13.8 Guided Exploration

If you look into causal modeling, you'll find mention of problematic covariates such as colliders or confounders. We've talked about confounders already. A **collider** is a variable that is caused by two other variables, and when you condition on it, it can induce a spurious relationship between those two variables.

In this exercise, we'll look at a simple example of a collider in the manner we did the confounder. First, run the available code to see what you get. Then,

[18]Malcolm Barrett was kind enough to give us a review of the content in this chapter, and their text was a great resource for much of it. As a result, it's much better than it would have been, and we definitely recommend it for a more in-depth look at causal inference.

attempt to incorporate the simulation approach we used for the confounder example (Section 13.5.1), and change some of the relevant coefficients around.

Python

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Set seed for reproducibility
np.random.seed(42)

# Generate synthetic data
n = 2500
x = np.random.normal(size=n)          # the feature
y = np.random.normal(size=n)          # the target (no relation to x)
z = x + y + np.random.normal(size=n)  # the collider

data = pd.DataFrame({'x': x, 'y': y, 'z': z})

# Fit linear models
model_without_z = LinearRegression().fit(data[['x']], data['y'])
model_with_z = LinearRegression().fit(data[['x', 'z']], data['y'])

# Compare x coefficients
pd.DataFrame({
    'estimate_1': model_without_z.coef_[0],
    'estimate_2': model_with_z.coef_[0]
}, index=['x']).round(3)
```

R

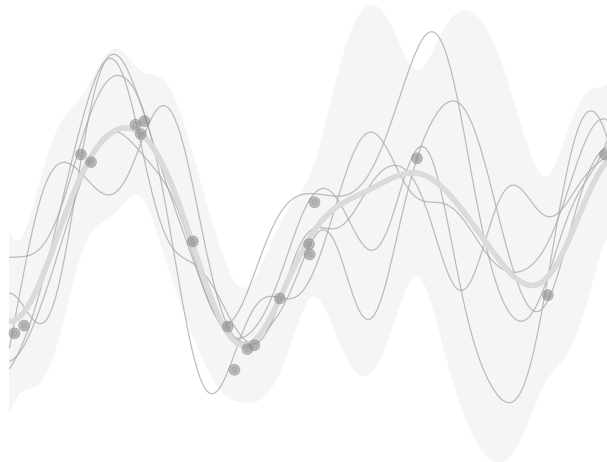
```
# Set seed for reproducibility
library(tidyverse)

set.seed(42)

# Generate synthetic data
n = 2500
x = rnorm(n)          # the feature
y = rnorm(n)          # the target (no relation to x)
z = x + y + rnorm(n)  # the collider

data = tibble(x = x, y = y, z = z)
```

```
# Fit linear models
model_without_z = lm(y ~ x, data = data)
model_with_z = lm(y ~ x + z, data = data)

# Compare x coefficients
tibble(
    estimate_1 = coef(model_without_z)['x'],
    estimate_2 = coef(model_with_z)['x']
)
```

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

14

Dealing with Data



It's an inescapable fact that models need data to work. One of the dirty secrets in data science is that getting the data right will do more for your model than any fancy algorithm or hyperparameter tuning. Data is messy, and there are a lot of ways it can be messy. In addition, dealing with a target variable on its terms can lead to more interesting results. In this chapter we'll discuss some of the most common data issues and things to consider. There's a lot to know about data before you ever get into modeling it, so we'll give you some things to think about in this chapter.

14.1 Key Ideas

- Data transformations can provide many modeling benefits.
- Label and text-based data still needs a numeric representation, and this can be accomplished in a variety of ways.
- The data type for the target may suggest a particular model but does not necessitate one.

- The data *structure*, for example, temporal, spatial, censored, etc., may suggest a particular modeling domain to use.
- Missing data can be handled in a variety of ways, and the simpler approaches are typically not great.
- Class imbalance is a very common issue in classification problems, and there are a number of ways to deal with it.
- Latent variables are everywhere!

14.1.1 Why this matters

Knowing your data is one of the most important aspects of any application of *data* science. It's not just about knowing what you have, but also what you can do with it. The data you have will influence the models you can potentially use, the features you can create and manipulate, and have a say on the results you can expect.

14.1.2 Helpful context

We're talking very generally about data here, so not much background is needed. The models mentioned here are covered in other chapters, or build upon those, but we're not doing any actual modeling here.

14.2 Feature and Target Transformations

Transforming variables from one form to another provides several benefits in modeling, whether applied to the target, features, or both. *Transformation should be used in most model situations.* Just some of these benefits include:

- More comparable feature effects and related parameters
- Faster estimation
- Easier convergence
- Helping with heteroscedasticity

For example, just **centering** features, i.e., subtracting their respective means, provides a more interpretable intercept that will fall within the actual range of the target variable in a standard linear regression. After centering, the intercept tells us what the value of the target variable is when the features are at their means (or reference value if categorical). Centering also puts the intercept within the expected range of the target, which often makes for easier parameter estimation. So even if easier interpretation isn't a major concern, variable transformations can help with convergence and speed up estimation, so can always be of benefit.

14.2.1 Numeric variables

The following table shows the interpretation of some very common transformations applied to numeric variables: logging, and standardizing to mean zero, standard deviation of one[1]. Note that for logging, these are approximate interpretations.

Table 14.1: Common Numeric Transformations

Target	Feature	Change in X	Change in Y
y	x	1 unit	B unit
log(y)	x	1 unit	100 * (exp(B) -1)%
log(y)	log(x)	1% change	B%
y	scale(x)	1 standard deviation	B unit
scale(y)	scale(x)	1 standard deviation	B standard deviation

For example, it is very common to use **standardized** or **scaled** variables. Some also call this **normalizing**, as with **batch or layer normalization** in deep learning, but this term can mean a lot of things, so one should be clear in their communication. If y and x are both standardized, a one-unit (i.e., one standard deviation) change in x leads to a β standard deviation change in y. So, if β was .5, a standard deviation change in x leads to a half standard deviation change in y. In general, there is nothing to lose by standardizing, so you should employ it often.

Another common transformation, particularly in machine learning, is **min-max scaling**. This involves changing variables to range from a chosen minimum value to a chosen maximum value, and usually this means zero and one respectively. This transformation can make numeric and categorical indicators more comparable, or at least put them on the same scale for estimation purposes, and so can help with convergence and speed up estimation. The following demonstrates how we can employ such approaches.

Python

When using sklearn, it's a bit of a verbose process to do such a simple transformation. However, it is beneficial when you want to do more complicated things, especially when using data pipelines.

[1]Scaling to a mean of zero and standard deviation of one is not the only way to scale variables. You can technically scale to any mean and standard deviation you want, but in tabular data settings you will have different and possibly less interpretability, and you may lose something in model estimation performance (convergence). For deep learning, the actual normalization may be adaptive applied across iterations.

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import numpy as np

# Create a random sample of integers
data = np.random.randint(low=0, high=100, size=(5, 3))

# Apply StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)

# Apply MinMaxScaler
minmax_scaler = MinMaxScaler()
minmax_scaled_data = minmax_scaler.fit_transform(data)
```

R

R makes it easy to do simple transformations like standardization and logs without external packages, but you can also use tools like recipes and mlr3 pipeline operations when needed to make sure your preprocessing is applied appropriately.

```
# Create a sample dataset
data = matrix(sample(1:100, 15), nrow = 5)

# Standardization
scaled_data = scale(data)

# Min-Max Scaling
minmax_scaled_data = apply(data, 2, function(x) {
    (x - min(x)) / (max(x) - min(x))
})
```

Using a **log** transformation for numeric targets and features is straightforward and comes with several benefits. For example, it can help with **heteroscedasticity**, which is when the variance of the target is not constant across the range of the predictions[2]. It can also help to keep predictions positive after transformation, allows for interpretability gains, and more.

[2]For the bazillionth time, logging does not make data 'normal' so that you can meet your normality assumption in linear regression.

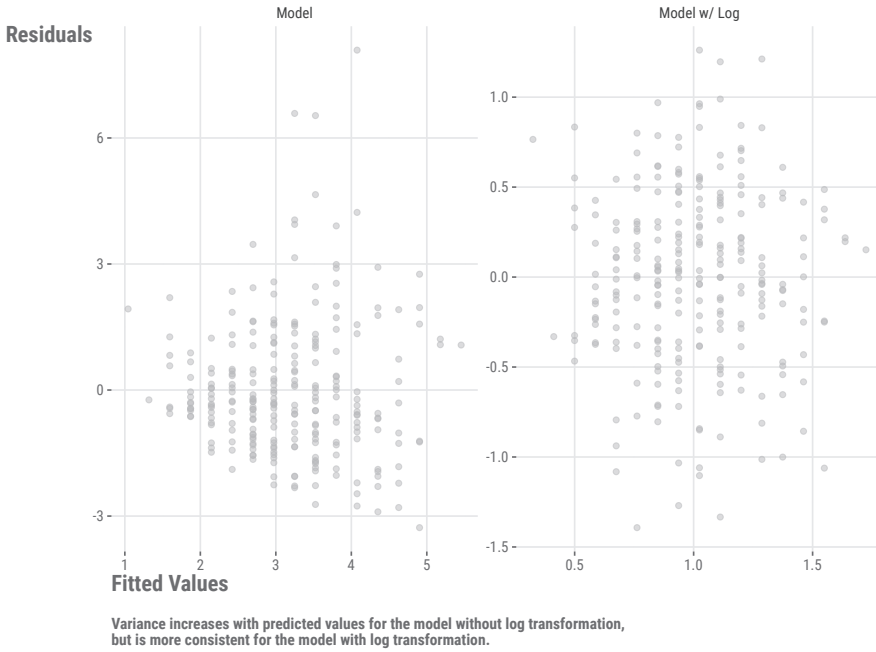


Figure 14.1: Log transformation and heteroscedasticity.

One issue with logging is that it is not a linear transformation. While this can help capture nonlinear feature-target relationships, it can also make some post-modeling transformations less straightforward. Also if you have a lot of zeros, 'log plus one' transformations are not going to be enough to help you overcome that hurdle[3]. Logging also won't help much when the variables in question have few distinct values, like ordinal variables, which we'll discuss later in Section 14.2.3.

Categorizing Continuous Variables

It is rarely a good idea or necessary to transform a numeric feature or target to a categorical one. Doing so potentially throws away useful information by making the feature a less reliable measure of the underlying construct. For example, discretizing age to 'young' and 'old' does not help your model, and you can always get predictions for what you would consider 'young' and 'old' after the fact. It is extremely common to see, particularly in machine learning contexts when applied to target variables. The main reason appears to be just so that a classification

[3]That doesn't mean you won't see many people try (and fail).

approach can be used. But it's just not a statistically or practically sound thing to do, and can ultimately hinder interpretation.

One caveat to discretizing continuous variables is in very large data situations where it is applied to features or parameters, and is often done to speed up estimation or provide computational benefits. In the long run, the desired results will be similar to using the original continuous values. For example, this principle is at the heart of **quantization**, which is a common approach in deep learning that reduces the precision of values (like converting 32-bit floating points to 8-bit integers) to improve computational efficiency while maintaining acceptable model performance.

14.2.2 Categorical variables

Despite their ubiquity in data, we can't analyze raw text information as it is. Character strings, and labeled features like factors, must be converted to a numeric representation before we can analyze them. For categorical features, we can use something called **effects coding** to test for specific types of group differences. Far and away the most common type is called **dummy coding** or **one-hot encoding**[4], which we visited previously in Section 3.5.2. In these situations we create columns for each category, and the value of the column is 1 if the observation is in that category, and 0 otherwise. Here is a one-hot encoded version of the `season` feature that was demonstrated previously.

Table 14.2: One-Hot Encoding

seasonFall	seasonSpring	seasonSummer	seasonWinter	season
1	0	0	0	Fall
1	0	0	0	Fall
1	0	0	0	Fall
1	0	0	0	Fall
0	0	1	0	Summer
0	0	1	0	Summer
1	0	0	0	Fall
0	0	1	0	Summer
0	0	0	1	Winter

[4]Note that one-hot encoding can refer to just the 1/0 coding for all categories, or to the specific case of dummy coding where one category is dropped. Make sure the context is clear.

> **i** Dummy Coding Explained
>
> For statistical models, when doing one-hot encoding all relevant information is incorporated in k-1 groups, where k is the number of groups, so one category will be dropped from the model matrix. This dropped category is called the **reference**. In a standard linear model, the intercept represents the mean of the target for the reference category. The coefficients for the other categories are the difference between the mean for the reference category and the group mean of the category being considered.
> As an example, in the case of the `season` feature, if the dropped category is `winter`, the intercept tells us the mean rating for `winter`, and the coefficients for the other categories are the difference between the value for `winter` and the mean of the target for `fall`, `summer` and `spring`.
> In other models, we include all categories in the model. The model learns how to use them best and might only consider some or one of them at a time.

When we encode categories for statistical analysis, we can summarize their impact on the target variable in a single result for all categories. For a model with only categorical features, we can use an **ANOVA** (Section 3.5.2) for this. But a similar approach can also be used for mixed models, splines, and other models to summarize categorical, spline, and other effects. Techniques like SHAP also provide a way to summarize the total effect of a categorical feature (Section 5.7).

Text embeddings

When it comes to other string representations like sentences and paragraphs, we can use other methods to represent them numerically. One important way to encode text is through an **embedding**. This is a way of representing the text as a vector of numbers, at which point the numeric embedding feature is used in the model like any other. The way to do this usually involves a model or a specific part of the model's architecture, one that learns the best way to represent the text or categories numerically. This is commonly used in deep learning, and natural language processing in particular. However, embeddings can also be used as a preprocessing step in *any* modeling situation.

To understand how embeddings work, consider a one-hot encoded matrix for a categorical variable. This matrix then connects to a hidden layer of a neural network. The weights learned for that layer are the embeddings for the categorical variable. While this isn't the exact method used (there are more efficient methods that don't require the actual matrix), the concept is the same. In addition, we normally don't even use whole words. Instead, we break the text into smaller units called **tokens**, like characters or subwords, and then use embeddings for those units. Tokenization is used in many of the

most successful models for natural language processing, including those such as ChatGPT.

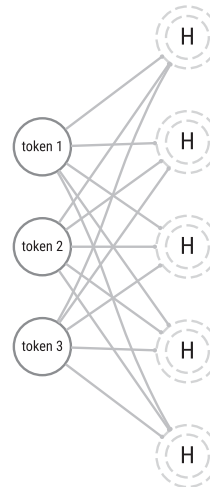


Figure 14.2: Conceptual example of an embedding.

Multiclass targets

We've talked about and demonstrated models with binary targets, but what about when there are more than two classes? In statistical settings, we can use a **multinomial regression**, which is a generalization of (binomial) logistic regression to more than two classes via the multinomial distribution. Depending on the tool, you may have to use a *multivariate* target of the counts, though most commonly they would be zeros and ones for a classification model, which then is just a one-hot encoded target. The following table demonstrates how this might look.

Table 14.3: Multinomial Data Example

x1	x2	target	Class A	Class B	Class C
−0.61	2	A	1	0	0
−0.20	1	C	0	0	1
−0.27	7	C	0	0	1
−0.47	5	B	0	1	0
0.70	7	A	1	0	0

With Bayesian tools, it's common to use the **categorical distribution**, which is a different generalization of the Bernoulli distribution to more than two

classes. Unlike the bi/multinomial distribution, it is not a count distribution, but an actual distribution over discrete values.

In the machine learning context, we can use a variety of models we'd use for binary classification. How the model is actually implemented will depend on the tool, but one of the more popular methods is to use **one-vs.-all** or **one-vs.-one** strategies, where you treat each class as the target in a binary classification problem. In the first case of one vs. all, you would have a model for each class that predicts whether an observation is in that class versus the other classes. In the second case, you would have a model for each pair of classes. You should generally be careful with either approach if interpretation is important, as it can make the feature effects very difficult to understand. As an example, we can't expect feature X to have the same effect on the target in a model for class A vs. B, as it does in a model for class A vs. (B & C) or A & C. As such, it can be misleading when the models are conducted as if the categories are independent.

Regardless of the context, interpretation is now spread across multiple target outputs, and so it can be difficult to understand the overall effect of a feature on the target. Even in the statistical model setting (e.g., a multinomial regression), you now have coefficients that regard *relative* effects for one class versus a reference group, and so they cannot tell you a *general* effect of a feature on the target. This is where tools like marginal effects and SHAP can be useful (Chapter 5).

Multilabel targets

Multilabel targets are a bit more complicated and are not as common as multiclass targets. In this case, each observation can have multiple labels. For example, if we wanted to predict genre based on the movie review data, we could choose to allow a movie to be both a comedy and action film, a sci-fi horror, or a romantic comedy. In this setting, labels are not mutually exclusive. If there are not too many unique label settings, we can treat the target as we would other multiclass targets. But if there are many, we might need to use a different model to go about things more efficiently.

> **i** Categorical Objective Functions
>
> In many situations where you have a categorical target, you will use a form of **cross-entropy loss** for the objective function. You may see other names such as *log loss* or *logistic loss* or *negative log-likelihood* depending on the context, but usually it's just a different name for the same underlying objective.

14.2.3 Ordinal variables

So far in our discussion of categorical data, the categories are assumed to have no order. But it's quite common to have labels like "low", "medium", and "high", or "very bad", "bad", "neutral", "good", "very good", or are a few numbers, like ratings from 1 to 5. **Ordinal data** is categorical data that has a known ordering, but which still has arbitrary labels. Let us repeat that, *ordinal data is categorical data.*

Ordinal features

The simplest way to treat ordinal features is as if they were numeric. If you do this, then you're just pretending that it's not categorical. In practice this is usually fine for features. Most of the transformations we mentioned previously aren't going to be as useful, but you can still use them if you want. For example, logging ratings 1-5 isn't going to do anything for you model-wise, but it technically doesn't hurt anything. You should know that typical statistics like means and standard deviations don't really make sense for ordinal data, so the main reason for treating them as numeric is for modeling convenience.

If you choose to treat an ordinal feature as categorical, you can ignore the ordering and do the same as you would with categorical data. This would allow for some nonlinearity since the category means will be whatever they need to be. There are some specific techniques to coding ordinal data for use in linear models, but they are not commonly used, and they generally aren't going to help the model performance or interpreting the feature, so we do not recommend them. You could, however, use old-school **effects coding** that you would incorporate traditional ANOVA models, but again, you'd need a good reason to do so[5].

The take-home message for ordinal features is generally simple. Treat them as you would numeric features or non-ordered categorical features. Either is fine.

Ordinal targets

Ordinal targets, on the other hand, can be trickier to deal with. If you treat them as numeric, you're assuming that the difference between 1 and 2 is the same as the difference between 2 and 3, and so on. This is probably not true. You are also ignoring how predictions are bounded by the observed values. If you treat them as categorical and use standard models for that setting, you're assuming that there is no connection between categories. So what should you do?

There are a number of ways to model ordinal targets, but probably the most common is the **proportional odds model**. This model can be seen as a generalization of the logistic regression model, and is very similar to it, and

[5]Try Helmert coding for instance! No, don't do that.

actually identical if you only had two categories. It basically is a model of category (2 or higher) vs. category 1, (3 or higher) vs. (2 or 1), etc. Basically you can think of it as subsequent binary model settings, but which you assume the feature effects are constant across settings. But other models beyond proportional odds that relax the assumptions are also possible. As an example, one approach would concern subsequent categories, the 1-2 category change, the 2-3 category change, and so on.

As an example, here are predictions from an ordinal model. In this case, we categorize[6] rounded movie ratings as 2 or less (Low), 3 (Average), or 4 or more (High), and predict the probability of each category based on the release year of the movie. So we get three sets of predicted probabilities, one for each category. In this example, we see that the probability of a movie being rated 4 or more has increased over time, while the probability of a movie being rated 2 or less has decreased. The probability of a movie being rated 3 has remained relatively constant, and is most likely.

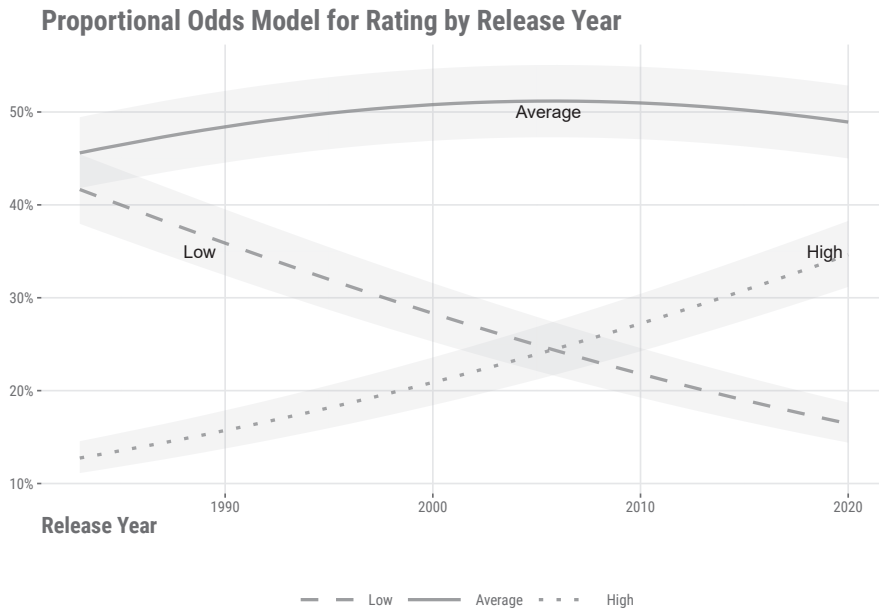


Figure 14.3: Proportional odds model for rating by release year.

Ordinality of a categorical outcome is largely ignored in machine learning applications. The outcome is either treated as numeric or multiclass classification.

[6]This is just for demonstration purposes. You should not categorize a continuous variable unless you have a very good reason to do so.

This is not necessarily a bad thing, especially if prediction is the primary goal. But if you need a categorical prediction, treating the target as numeric means you have to make an arbitrary choice to classify the predictions. And if you treat it as multiclass, you're ignoring the ordinality of the target, which may not work as well in terms of performance.

Rank data

Though ranks are ordered, with rank data we are referring to cases where the observations are uniquely ordered. An ordinal vector of 1-6 with numeric labels could be something like [2, 1, 1, 3, 4, 2], where no specific value is required and any value could be repeated. In contrast, rank data would be [2, 1, 3, 4, 5, 6], each being unique (unless you allow for ties). For example, in sports, a ranking problem would regard predicting the actual finish of the runners. Assuming you have a modeling tool that actually handles this situation, the objective will be different from other scenarios. Statistical modeling methods include using the Plackett-Luce distribution (or the simpler variant Bradley-Terry model). In machine learning, you might use so-called learning to rank methods, like the RankNet and LambdaRank algorithms, and other variants for deep learning models.

14.3 Missing Data

Table 14.4: Data with Missing Values

x1	x2	x3
4	0	?
7	3	B
?	5	A
8	?	B
?	3	C

Missing data is a common challenge in data science, and there are a number of ways to deal with it, usually by substituting, or **imputing**, the substituted value for the missing one. Here we'll provide an overview of common techniques to deal with missing data.

14.3.1 Complete case analysis

The first way to deal with missing data is the simplest – **complete case analysis**. Here we only use observations that have no missing data and drop the rest. Unfortunately, this can lead to a lot of lost data, and it can lead to

biased statistical results if the data is not **missing completely at random** (missingness is not related to the observed or unobserved data). There are special cases of some models that by their nature can ignore the missingness under an assumption of **missing at random** (missingness is not related to the unobserved data), but even those models would likely benefit from some sort of imputation. If you don't have much missing data though, dropping the missing data is fine for practical purposes[7]. How much is too much? Unfortunately that depends on the context, but if you have more than 10% missing, you should probably be looking at alternatives.

14.3.2 Single value imputation

Single value imputation involves replacing missing values with a single value, such as the mean, median, mode or some other typical value of the feature. As common an approach as this is, it will rarely help your model for a variety of reasons. Consider a numeric feature that is 50% missing, and for which you replace the missing with the mean. How good do you think that feature will be when at least half the values are identical? Whatever variance it normally would have and share with the target is probably reduced, and possibly dramatically. Furthermore, you've also attenuated correlations it has with the other features, which may mute other modeling issues that you would otherwise deal with in some way (e.g., collinearity), or cause you to miss out on interactions.

Single value imputation makes perfect sense if you *know* that the missingness should be a specific value, like a count feature where missing means a count of zero. If you don't have much missing data, it's unlikely this would have any real benefit over complete case analysis. One exception is the case where imputing the feature then allows you to use all the other complete feature samples that would otherwise be dropped. But then, you could just drop this less informative feature while keeping the others, as it will often not be very useful in the model.

14.3.3 Model-based imputation

Model-based imputation is more complicated but can be very effective. In essence, you run a model for complete cases in which the feature with missing values is now the target, and all the other features and primary target are used to predict it. You then use that model to predict the missing values, using the predictions as the imputed values. After these predictions are made, you

[7]While many statisticians will possibly huff and puff at the idea of dropping data, there are two things to consider. With minimal missingness you'll likely never come to a different conclusion unless you have very little data to come to a conclusion about, which is already the bigger problem. Secondly, it's impossible to prove one way or another if the data is missing at random, because doing so would require knowing the missing values.

move on to the next feature and do the same. There are no restrictions on which model you use for which feature. If the other features in the imputation model also have missing data, you can use something like mean imputation to get more complete data if necessary as a first step, and then when their turn comes, impute those values.

Although the implication is that you would have one model per feature and then be done, you can do this iteratively for several rounds, such that the initial imputed values are then used in subsequent model rounds to reimpute other features' missing values. You can do this as many times as you want, but the returns will diminish. In this setting, we are assuming you'll ultimately end with a single imputed value for each missing one, which reflects the last round of imputation.

14.3.4 Multiple imputation

Multiple imputation (MI) is a more complicated technique, but it can be very useful in some situations, depending on what you're willing to sacrifice for having better uncertainty estimates. The idea is that you create multiple imputed datasets, each of which is based on the **predictive distribution** of the model used in model-based imputation (see Section 4.4). Say we use a linear regression assuming a normal distribution to impute feature A. We would then draw repeatedly from the predictive distribution of that model to create multiple datasets with (randomly) imputed values for feature A.

Let's say we do this 10 times, and we now have 10 imputed datasets, each with a now complete feature A, but each with somewhat different imputed values. We now run our desired model on each of these datasets. Final model results are averaged in some way to get final parameter estimates. Doing so acknowledges that your single imputation methods have uncertainty in those imputed values, and that uncertainty is incorporated into the final model estimates, including the uncertainty in those estimates.

MI can in theory handle any source of missingness and can be a very powerful technique. But it has some drawbacks that are often not mentioned, but which everyone that's used it has experienced. One is that you need a specified target distribution for all imputation models used, in order to generate random draws with appropriate uncertainty. Your final model presumably is also a probabilistic model with coefficients and variances you are trying to estimate and understand. MI probably isn't going to help boosting or deep learning models that have native methods for dealing with missing values, or at least, offers little over single value imputation for those approaches. In addition, if you have very large data and a complicated model, you could be spending a long time both waiting for the models and debugging them, because you still would need to assess the imputation models much like any other for the most part. Finally, few data or post-model processing tools that you commonly use

will work with MI results, especially those regarding visualization. So you will have to hope that whatever package you use for MI will do what you need. As an example, you'd have to figure out how you're going to impute interaction or spline terms if you have them.

Practically speaking, MI takes a lot of effort to often come to the same conclusions you would have with a single imputation method, or possibly fewer conclusions for anything beyond GLM coefficients and their standard errors. But if you want the best uncertainty estimates for those models, MI can be the way to go.

14.3.5 Bayesian imputation

One final option is to run a Bayesian model where the missing values are treated as parameters to be estimated, and they would have priors just like other parameters as well. MI is basically a variant of Bayesian imputation that can be applied to the non-Bayesian model setting, so why not just use the actual Bayesian approach? Some modeling packages can allow you to try this very easily, and it can be very effective. But it is also very computationally intensive and can be very slow as you may be increasing the number of parameters to estimate dramatically. At least it would be more fun than standard MI, so we recommend exploring it if you were going to do MI anyway.

14.4 Class Imbalance

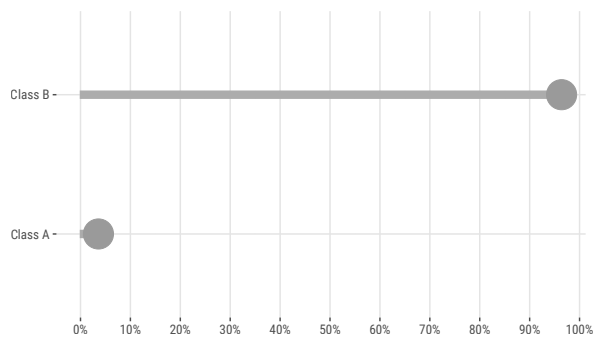


Figure 14.4: Class imbalance.

Class imbalance refers to the situation where the target variable has a large difference in the number of observations in each class. For example, if you have a binary target, and 90% of the observations are in one class, and 10% in the other, you would have class imbalance. You'll almost never see a 50/50 split in the real world, but the issue is that as we move further away from that point, we can start to see problems in model estimation, prediction, and interpretation. In this example, if we just predict the majority class in a binary classification problem, our accuracy would be 90%! Under other circumstances that might be a great result for accuracy, but in this case it's not. So right off the bat one of our favorite metrics to use for classification models isn't going to help us much.

For classification problems, *class imbalance is the rule, not the exception.* This is because nature just doesn't sort itself into nice and even bins. The majority of people in a random sample do not have cancer, the vast majority of people have not had a heart attack in the past year, most people do not default on their loans, and so on.

There are a number of ways to help deal with class imbalance, and the method that works best will depend on the situation. Some of the most common are:

- **Use different metrics**: Use metrics that are less affected by class imbalance, such as area under a receiver operating characteristic curve (AUC), or those that balance the tradeoff between precision and recall, like the F1 score, or something like the balanced accuracy score, which balances recall and true negative rate.
- **Oversampling/Undersampling**: Randomly sample from the minority (majority) class to increase (decrease) the number of observations in that class. For example, we can randomly sample with replacement from the minority class to increase the number of observations in that class. Or we can take a sample from the majority class to balance the resulting dataset.
- **Weighted objectives**: Weight the loss function to give more weight to the minority class. Although commonly employed, and simple to implement with tools like lightgbm and xgboost, it often fails to help and can cause other issues.
- **Thresholding**: Change the threshold for classification to be more sensitive to the minority class. Nothing says you have to use 0.5 as the threshold for classification, and you can change it to be more sensitive to the minority class. This is a very simple approach and may be all you need.

These are not necessarily mutually exclusive. For example, it's probably a good idea to switch your focus to a metric besides accuracy even as you employ other techniques to handle imbalance. See (Clark 2025).

14.4.1 Calibration issues in classification

Probability **calibration** is often a concern in classification problems. It is a bit more complex of an issue than just having class imbalance but is often discussed in the same setting. Having calibrated probabilities refers to the situation where the predicted probabilities of the target match up well to the actual proportion of observed classes. For example, if a model predicts an average 0.5 probability of loan default for a certain segment of the samples, the actual proportion of defaults should be around 0.5.

One way to assess calibration is to use a **calibration curve**, which is a plot of the predicted probabilities vs. the observed proportions. We bin our predicted probabilities, say, into 5 or 10 equal bins. We then calculate the average predicted probability and the average observed proportion of the target in each bin. If the model is well calibrated, the points should fall along the 45-degree line. If not, the points will fall above or below the line.

In Figure 14.5, one model seems to align well with the observed proportions based on the chosen bins. The other model (dashed line) is not so well calibrated and is overshooting with its predictions. For example, that model's average prediction for the third bin predicts a ~0.5 probability of the outcome, while the actual proportion is around 0.2.

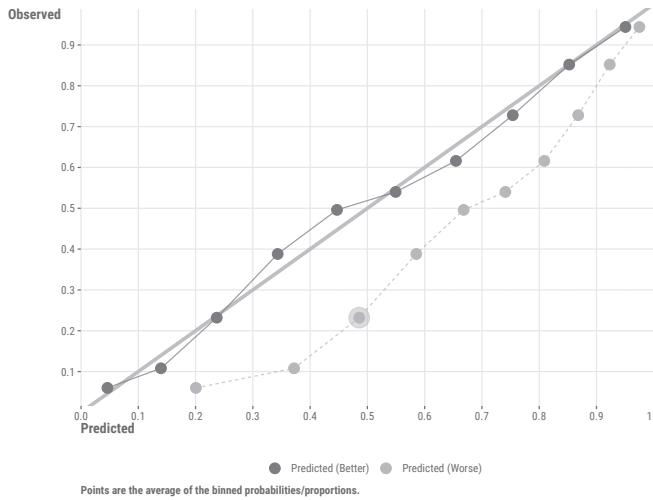


Figure 14.5: Calibration plot.

While the issue is an important one, it's good to keep the issue of calibration and imbalance separate. As miscalibration implies bias, bias can happen irrespective of the class proportions, and it can be due to a variety of factors

related to the model, target, or features. Furthermore, miscalibration is not inherent to any particular model.

The assessment of calibration in this manner also has a few issues that we haven't seen reported in the documentation for it. For one, the observed 'probabilities' are proportions based on arbitrarily chosen bins, and there are multiple ways to choose the bins. The observed values also have some **measurement error** and have a natural variability that will partly reflect sample size[8]. In addition, these plots are often presented such that observed proportions are labeled as the 'true' probabilities. However, you do not have the *true* probabilities, just the *observed* class labels, so whether your model's predicted probabilities match observed proportions is actually a bit of a different question. The predictions have uncertainty as well, and this will depend on the model, sample size, and other factors. And finally, the number of bins chosen can also affect the appearance of the plot in a notable way if the sample size is small, which is a perceptual issue that can be misleading regardless of the models in question.

All this is to say that each point in a calibration plot, 'true' or predicted, has some uncertainty with it, and the difference in those values is not formally tested in any way by a calibration curve plot. Their uncertainty, if it was actually measured, could even overlap while still being statistically different! So, if we're interested in a more rigorous statistical assessment, the differences between models and the 'best case scenario' would need additional steps to suss out.

Some methods are available to calibrate probabilities if they are deemed miscalibrated, but they are not commonly implemented in practice and often involve another model-based technique, with all of its own assumptions and limitations. It's also not exactly clear that forcing your probabilities to be on the line is helping solve the actual modeling goal in any way[9]. But if you are interested, you can read more at the sklearn documentation on calibration.

[8]Note that each bin will reflect the portion of the test set size in this situation. If you have a small test set, the observed proportions will be more variable, and the calibration plot will be more variable as well.

[9]Oftentimes we are only interested in the ordering of the predictions, and not the actual probabilities. For example, if we are trying to identify the top 10% of people most likely to default on their loans, we'll just take the top 10% of predictions, and the actual probabilities are irrelevant for that goal.

14.5 Censoring and Truncation

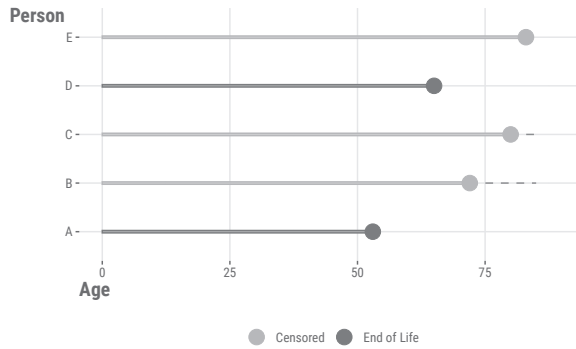


Figure 14.6: Censoring for time until death.

Sometimes, we just don't see all the data there is to see. **Censoring** is one situation where the target variable is not fully observed. This is common in techniques where the target is the 'time to an event', like death from a disease, but the event has not yet occurred for some observations in the data (thankfully!). Specifically this is called **right censoring**, and is the most common type of censoring, and depicted in Figure 14.6, where several individuals are only observed to a certain age and were still alive at that time. There is also **left censoring**, where the censoring happens from the other direction, and data before a certain point is unknown. Finally, there is **interval censoring**, where the event of interest occurs within some interval, but the exact value is unknown.

Survival analysis[10] is a common modeling technique in this situation, especially in fields informed by biostatistics, but you may also be able to keep things even more simple via something like tobit regression. In the tobit model, you assume that the target is fully observed, but that the values are censored, and you model the probability of censoring. This is a common technique in econometrics, and it allows you to keep a traditional linear model context.

Truncation is a situation where the target variable is only observed if it is above or below some value, even though we know other possibilities exist. One of the issues is that default distributional methods assume a distribution that

[10]Survival analysis is also called **event history analysis**, and is widely used in biostatistics, sociology, demography, and other disciplines where the target is the time to an event, such as death, marriage, divorce, etc.

is not bounded in the way that the data exhibits. In Figure 14.7, we restrict our data to 70 and below for practical or other reasons, but typical modeling methods predicting age would not respect that.

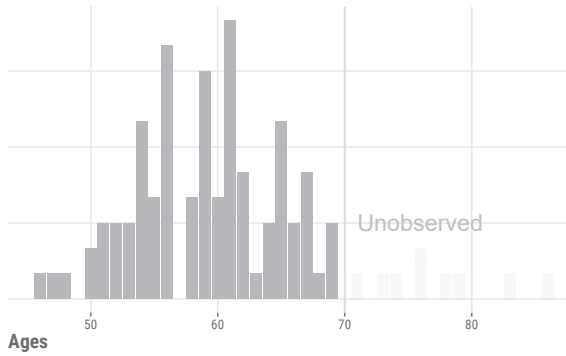


Figure 14.7: Truncation.

You could truncate predictions after the fact, but this is a bit of a hack, and often results in lumpiness in the predictions at the boundary that is rarely realistic. Alternatively, Bayesian methods allow you to model the target as a distribution with truncated distributions, and so you can model the probability of the target being above or below some value. There are also models such as **hurdle models** that might prove useful where the truncation is theoretically motivated, for example, a zero-inflated Poisson model for count data where the zero counts are due to a separate process than the non-zero counts.

> **i** Censoring vs. Truncation
>
> One way to distinguish censored and truncated data is that censored data is usually due to some external process such that the target is not observed, but could be possible (capping reported income at $1 million). Truncated data, on the other hand, is due to some internal process that prevents the target from being observed and is often derived from sample selection (we only want to model non-millionaires). We would not want predictions past the censored point to be unlikely, but we would want predictions past the truncated point to be impossible. Trickier still is that for bounded or truncated distributions that might be applied to the truncated scenario, such as folded vs. truncated distributions, they would not result in the same probability distributions even if they can be applied to the same data situation.

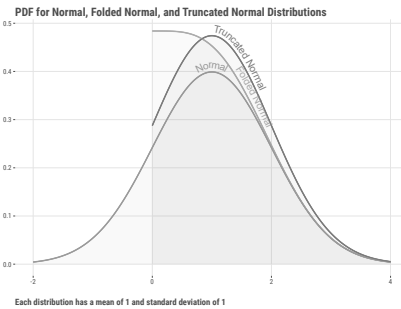


Figure 14.8: Folded and truncated distributions.

14.6 Time Series

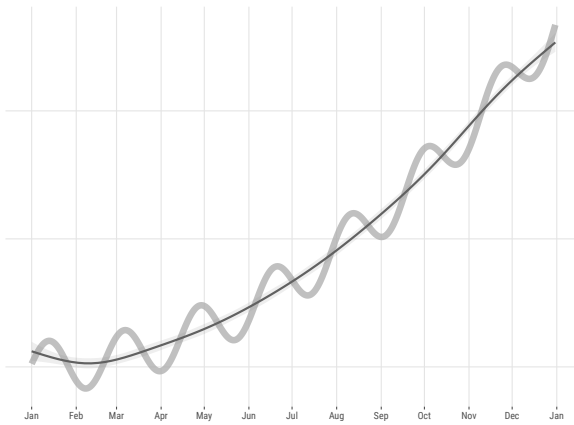


Figure 14.9: Time-series data.

Time series data is any data that incorporates values over a period of time. This could be something like a state's population over years, or the max temperature of an area over days. Time series data is very common in data science, and there are a number of ways to model such data.

14.6.1 Time-based targets

As in other settings, when the target is a value that varies over time, the most common approach is to use a linear model of some kind. Note that while the target varies over time, the features may be time-varying or not, and it is very common to have both types of features (e.g., a person's age vs. the person's race/ethnicity).

There are traditional autoregressive models that use the target's past values (**lags**) as features, for example, autoregressive moving average (**ARIMA**) models. In this case, the target is a function of its past values, and possibly the past values of other features. In these models, **forecasting**, or making predictions about future values, is the primary goal. Care will have to be taken to avoid data leakage in the training process, and to use proper standardization of the data if applicable. In some settings, common transformations such as logging can be used. In other cases, you may want to use a **differencing** approach to make the data stationary, which can be useful for some models. Differencing is the process of subtracting the previous value from the current value, and it can be done multiple times to remove trends and seasonality. Often this is done automatically as part of a hyperparameter search, but it can be done manually as well. Boosting models using lagged features can also be used for time series data and can be very effective in many situations.

Longitudinal data[11] is a special case of time series data, where the target is a function of time (e.g., date), but the data is typically grouped in some fashion and is often of a shorter sequence. An example would be a model for school performance for students over several semesters, where values are clustered within students over time. In this case, you can use some sort of time series regression, though many do not do well with shorter series. Instead, you can use a **mixed model** (Section 9.3), where you model the target as a function of time, but also include a random effect for the grouping variable, in this case, students. This could, and typically does, include both random intercepts and slopes for time. This is a very common approach in many domains and can be very effective in terms of performance as well. Mixed models can also be used for longer series, where the random effects are based on autoregressive covariance matrices. In this case, an ARIMA component is added to the linear model as a random effect to account for the time series nature of the data. This is fairly common in Bayesian contexts. Generalized additive models also work well in this setting.

Other models provide additional options for incorporating historical information, such as Bayesian methods for marketing data (Section 7.6) or reinforcement learning approaches (Section 12.3). Some models get more complex, like **recurrent neural networks** and their generalizations, and were specifically developed for sequential data. More recently, transformer-based models have

[11] You may also hear the term **panel data** in econometrics-oriented disciplines.

shown promise for time series analysis beyond text. In this area, 'foundational' models like TimeGPT, Moirai, Chronos, or TinyTimeMixers have been proposed, though they often require more data than is typically available in many common settings. In addition, they have not proven to be as effective as simpler models in many cases[12].

So, many models can be found specific to time series targets, and the choice of model will depend on the data, the questions we want to ask, and the goals we have. Most traditional models are still linear models, but traditional machine learning models, like boosting, and many neural network models have been developed that can handle time series data as well. It may be the case that you will switch to using different loss functions or metrics with time series data, such as (symmetric) mean absolute percentage error and scaled errors (Hyndman and Athanasopoulos (2021)).

> **i** Time Series vs. Longitudinal
>
> The primary distinguishing feature for referring to data as 'time series' or 'longitudinal' is the number of time points, where the latter typically has relatively few. This is arbitrary though.

14.6.2 Time-based features

When it comes to time-series features, we can apply time-specific transformations. One technique is the **fourier transform**, which can be used to decompose a time series into its component frequencies, much like how we use PCA (Section 12.2). This can be useful for identifying periodicity in the data, which can be used as a feature in a model.

In marketing contexts, some perform **adstocking** with features. This approach models the delayed effect of features over time, such that they may have their most important impact immediately, but still can impact the present target value from past values. For example, a marketing campaign might have the most significant impact immediately after it's launched, but it can still influence the target variable at later time points, albeit more weakly as the distance in time is extended. Adstocking helps capture this delayed effect without having to include multiple lagged features in the model. That said, including lagged features is also an option in this setting. In this case, you would have a feature for the current time point (t), the same feature for the previous time point (t-*1*), the feature for the time point before that (t-*2*), and so on.

[12] For example, see Nixtla's experiments with Amazon's Chronos vs. a simple combination of univariate models. NB: Amazon said that the default settings were not very good, and it would perform much better with different settings, but possibly still would not beat the simpler model.

Scaling time features

If you have the year as a feature, you can use it as a numeric feature or as a categorical feature. If you treat it as numeric, you need to consider what a zero means. In a linear model, the intercept usually represents the outcome when all features are zero. But with a feature like year, a zero year isn't meaningful in most contexts. To solve this, you can shift the values so that the earliest time point, like the first year in your data, becomes zero. This way, the intercept in your model will represent the outcome for this first time point, which is more meaningful. The same goes if you are using months or days as a numeric feature. It doesn't really matter which year/month/day is zero, just that zero refers to one of the actual time points observed. Shifting your time feature in this manner can also help with convergence for some types of models. In addition, you may want to convert the feature to represent decades, or quarters, or some other time period, to help with interpretation.

Dates and/or times can be a bit trickier. Often you can just split dates out into year, month, day, etc., and proceed with those as features. In other cases you'd want to track the time period to assess possible seasonal effects. You can use something like a **cyclic** approach (e.g., cyclic spline or sine/cosine transformation) to get at yearly or within-day seasonal effects. As mentioned, a fourier transform can also be used to decompose the time series into its component frequencies for use as model features. Time components like hours, minutes, and seconds can often be dealt with in similar ways, but you will more often deal with the periodicity in the data. For example, if you are looking at hourly data, you may want to consider the 24-hour cycle.

> **i** Calendars Are Hard
>
> *Weeks are not universal.* Some start on Sunday, others Monday. Some data contexts only consider weekdays. Some systems may have 52 or 53 weeks in a year, and dates may not be in the same week from one year to the next, etc. So use extra caution when considering weeks as a feature.

Covariance structures

In many cases you'll have features that vary over time but are not a time-oriented feature like year or month. For example, you might have a feature that is the number of people who visited a website over days. This is a time-varying feature, but it's not a time metric in and of itself.

In general, we'd like to account for the time-dependent correlations in our data, and a common way to do so is to posit a covariance structure that accounts for this in some fashion. This helps us understand how data points are related to each other over time, and requires us to estimate the correlations between observations. As a starting point, consider linear regression. In a standard

linear regression model, we assume that the samples are independent of one another, with a constant variance and no covariance.

Instead, we can also use something like a **mixed model**, where we include a random effect for each group and estimate the variance attributable to the grouping effect. By default, this ultimately assumes a constant correlation from time point to time point, but many tools allow you to specify a more complex covariance structure. A common method is to use autoregressive covariance structure that allows for correlations further apart in time to lessen. In this sense the covariance comes in as an added *random effect*, rather than being a model in and of itself as with ARIMA. Many such approaches to covariance structures are special cases of **Gaussian processes**, which are a very general technique to model time series, spatial, and other types of data.

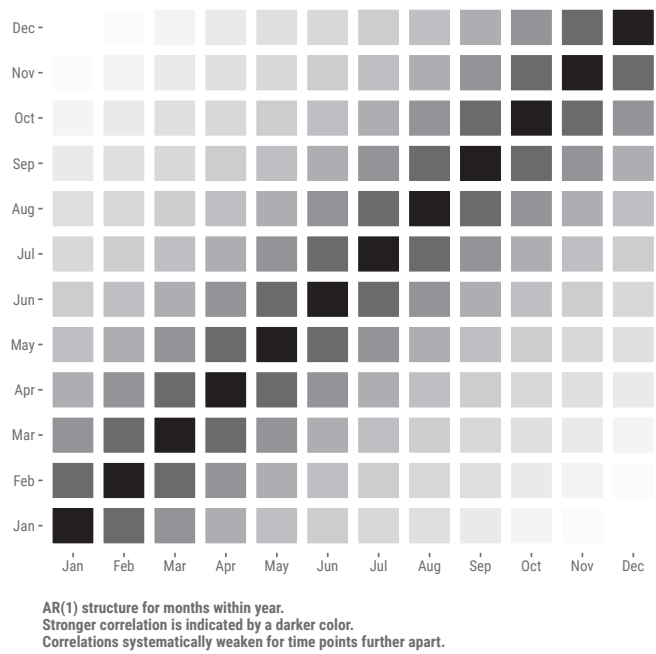


Figure 14.10: AR (1) covariance structure visualized.

14.7 Spatial Data

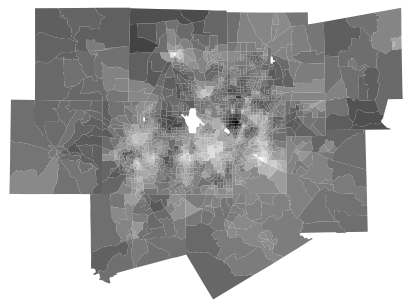


Figure 14.11: Spatial weighting applied to the Dallas-Fort Worth area census tracts.

We visited spatial data in a discussion on non-tabular data (Section 12.4.1), but here we want to talk about it from a modeling perspective, especially within the tabular domain. Say you have a target that is a function of location, such as the proportion of people voting a certain way in a county, or the number of crimes in a city. You can use a **spatial regression** model, where the target is a function of location, among other features that may or may not be spatially oriented. Two approaches already discussed may be applied in the case of having continuous spatial features, such as latitude and longitude, or discrete features like county. For the continuous case, we could use a GAM (Section 9.4) that employs a smooth interaction of latitude and longitude. For the discrete setting, we can use a mixed model (Section 9.3.2), where we include a random effect for county.

There are other traditional techniques to spatial regression, especially in the continuous spatial domain, such as using a **spatial lag**. In this case, we incorporate information about the neighborhood of an observation's location into the model. An example is shown in the previous visualization that depicts a weighted mean of neighboring values for different tracts (based on code from Walker (2023))[13]. Techniques include CAR (conditional autoregressive), SAR (spatial autoregressive), BYM, kriging, and more, and these models can be very effective. They can also be seen as a different form of random effects models very similar to those used for time-based settings via covariance structures. They can also be seen as special cases of Gaussian process regression more

[13]In the book's print version, darker colors indicate higher values, but in the web version, red represents larger relative values.

generally. So don't let the names fool you, you often will incorporate similar modeling techniques for both the time and spatial domains.

14.8 Multivariate Targets

Often you will encounter settings where the target is not a single value, but a vector of values. This is often called a **multivariate target** in statistical settings, or just the norm for deep learning. For example, you might be interested in predicting the number of people who will buy a product, the number of people who will click on an ad, and the number of people who will sign up for a newsletter. The main idea is that there is a correlation among the targets, and you want to take this into account when analyzing them simultaneously.

One model example we've already seen is the case where we have more than two categories for the target (Section 14.2.2). Some default approaches may take that input and just do a one-vs.-all, for each category, but this kind of misses the point. Others will simultaneously model the multiple targets in some way. On the other hand, it can be difficult to interpret results with multiple targets. Because of this, you'll often see results presented in terms of the respective targets anyway, and often even ignoring parameters specifically associated with such a model[14].

It is also common to have multiple targets in a regression setting, where you might want to predict multiple outcomes simultaneously. This is sometimes called **multivariate regression** and is a common technique in many fields. In this case, you can use a linear model, but you'll have to account for the correlation between the targets. This can be done with a **multivariate normal distribution**, where the targets are assumed to be normally distributed, and the covariance matrix for the targets is estimated rather than just a single variance value. This is a very common approach and is often used in mixed models as well, which offer a different way to go about the same model.

In deep learning contexts, the multivariate setting is ubiquitous. For example, if you want to classify the content of an image, you might have to predict something like different species of animals, or different types of car models. In natural language processing, you might want to predict the probability of different words in a sentence. In some cases, there are even multiple kinds of targets considered simultaneously! It can get very complex, but often in these

[14]It was common in social sciences back in the day to run a Multivariate ANOVA, and then if the result was statistically significant, and mostly because few practitioners knew what to do with the result, they would run separate ANOVAs for each target.

settings prediction performance far outweighs the need to interpret specific parameters, and so it's a good fit.

14.9 Latent Variables

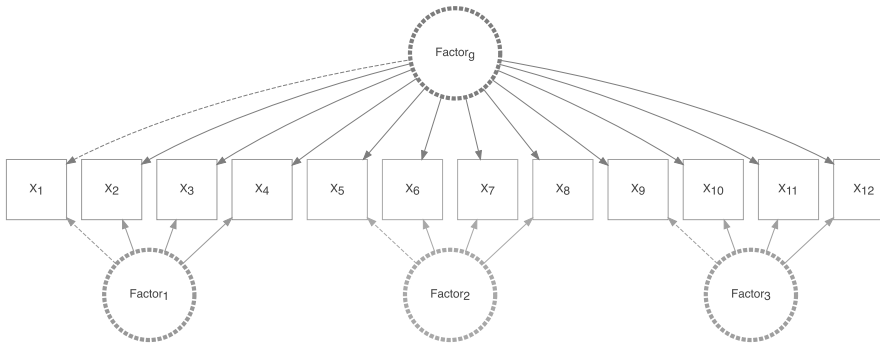


Figure 14.12: Latent variable model (bifactor).

Latent variables are a fundamental aspect of modeling, and simply put, they are variables that are not directly observed, but instead are inferred from other variables. Here are some examples of what might be called latent variables:

- The linear combination of features in a linear regression model is a latent variable, but usually we only think of it as such before the link transformation in GLMs (Chapter 8).
- The error term in any model is a latent variable representing all the unknown/unobserved/unmodeled factors that influence the target (Equation 3.3).
- The principal components in PCA (Chapter 12).
- The measurement error in any feature or target.
- The factor scores in a factor analysis model or structural equation (visualization above).
- The true target underlying the censored values (Section 14.5).
- The clusters in cluster analysis/mixture models (Section 12.2.1).
- The random effects in a mixed model (Section 9.3).
- The hidden states in a hidden Markov model.
- The hidden layers in a deep learning model (Section 11.7).

Though they may be used in different ways, it's easy to see that latent variables are very common in modeling, so it's good to get comfortable with the concept. Whether they're appropriate to your specific situation will depend on a variety

of factors, but they can be very useful in many settings, if not a required part of the modeling approach.

14.10 Data Augmentation

Data augmentation is a technique where you artificially increase the size of your dataset by creating new data points based on the existing data. This is a common technique in deep learning for computer vision, where you might rotate, flip, or crop images to create new training data. This can help improve the performance of your model, especially when you have a small dataset. Conceptually similar techniques are also available for text.

In the tabular domain, data augmentation is less common but still possible. For example, you can see it applied in class-imbalance settings (Section 14.4), where you might create new data points for the minority class to balance the dataset. This can be done by randomly sampling from the existing data points, or by creating new data points based on the existing data points. For the latter, SMOTE and many variants of it are quite common (for better or worse; see Elor and Averbuch-Elor (2022)).

Unfortunately for tabular data, these techniques are not nearly as successful as augmentation for computer vision or natural language processing, nor consistently so. Part of the issue is that tabular data is very noisy and fraught with measurement error, so in a sense, such techniques are just adding noise to the modeling process without any additional means to amplify the signal[15]. Downsampling the majority class can potentially throw away useful information. Simple random upsampling of the minority class can potentially lead to an overconfident model that doesn't generalize well. In the end, the best approach is to get more and/or better data, but as that often is not possible, hopefully more successful methods will be developed in the future for the tabular domain.

14.11 Wrapping Up

There's a lot going on with data before you ever get to modeling, and which will affect every aspect of your modeling approach. This chapter outlined common data types, issues, and associated modeling aspects, but in the end, you'll always have to make decisions based on your specific situation, and they

[15] Compare to the image settings where there is relatively little measurement error, by just rotating an image, you are still preserving the underlying structure of the data.

will often not be easy ones. These are only some of the things to consider, so be ready for surprises, and be ready to learn from them!

14.11.1 The common thread

Many of the transformations and missing data techniques can be applied in diverse modeling settings. Likewise, you may find yourself dealing with different target variable issues like imbalance or censoring, and dealing with temporal, spatial or other structures, in a variety of models. The key is to understand the data well, and to make the best decisions you can based on that knowledge.

14.11.2 Choose your own adventure

Consider revisiting a model covered elsewhere in this book in light of the data issues discussed here. For example, how might you deal with class imbalance for a boosted tree model? How would you deal with spatial structure in a neural network? How would you deal with a multivariate target in a time series model?

14.11.3 Additional resources

Here are some additional resources to help you learn more about the topics covered in this chapter.

Transformations

- About Feature Scaling and Normalization (Raschka (2014))
- What are Embeddings (Boykis (2023))

Class Imbalance

- Brief Overview (Google)
- Handling imbalanced datasets in machine learning (Rocca (2019))
- Imbalanced Outcomes: Challenges & Solutions (Clark (2025))
- A Gentle Introduction to Imbalanced Classification (Brownlee (2019))

Calibration

- Why some algorithms produce calibrated probabilities (StackExchange)
- Predicting good probabilities with supervised learning (Niculescu-Mizil and Caruana (2005))

Survival, Ordinal and Other Models

- Regression Modeling Strategies is a great resource for statistical modeling in general (Harrell (2015))
- Ordinal Regression Models in Psychology: A Tutorial is a great resource

by the author of the best Bayesian modeling package brms[16] (Bürkner and Vuorre (2019))

Time Series

- Forecasting: Principles and Practice (Hyndman and Athanasopoulos (2021))

Latent Variables

There's a ton of stuff, but one of your humble authors has an applied treatment that is not far from the conceptual approach of this text, along with a bit more general exploration. However, we'd recommend more of an 'awareness' of latent variable modeling, as you will likely be more interested in the specific application for your data and model, which can be very different from these contexts.

- Thinking about Latent Variables (Clark (2018b))
- Graphical and Latent Variable Modeling (Clark (2018a))

Data Augmentation

- What is Data Augmentation? (Amazon (2024))

Other

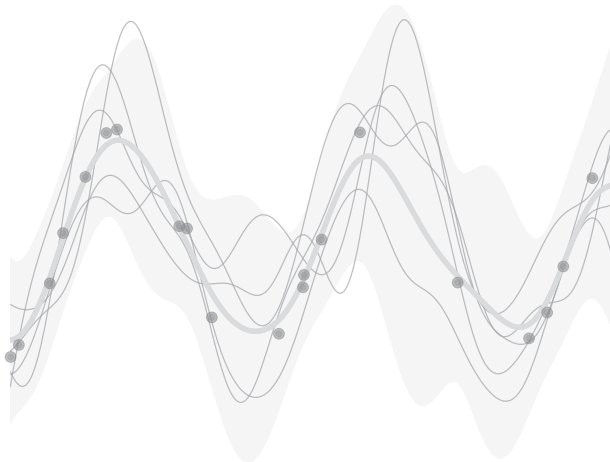
We always need to inspect the data closely and see if it matches our expectations. This is a critical part of the modeling process and can often be the most time-consuming part. There are tools that can help with the validation process, and to that end, you might look at pointblank in R or y_data_profiling in Python as options to get started.

[16] No, we're not qualifying that statement.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

15

Danger Zone



When it comes to conducting models in data science, a lot can go wrong, and in many cases it's easy to get lost in the weeds and lose sight of the bigger picture. Throughout the book, we've covered many instances in which caution is warranted in the modeling approach.

In this chapter, we'll more explicitly discuss some common pitfalls that can sneak up on you when you're working on a data science project, and others that just came to mind while we were thinking about it. The topics are based on things we've commonly seen in consulting across many academic disciplines and industries, and here we attempt to provide a very general overview. That said, it is by no means exhaustive, and you may come across additional issues in your situation. The following groups of focus attempt to reflect the content of the book as it was presented.

15.1 Linear Models and Related Statistical Endeavors

Statistical models are a powerful tool for understanding the structure and meaning in your data. They are also excellent at helping us to understand the uncertainty in our data and the aspects of the model we wish to estimate. However, there are many ways in which problems can arise with statistical models.

15.1.1 Statistical significance

One of the most common mistakes when conducting statistical linear models is simply relying too heavily on the statistical result. Statistical significance is simply not enough to determine feature importance or model performance. When complex statistical models are applied to small data, the results are typically very noisy and statistical significance can be misleading. This also means that 'big' effects can be a reflection of that noise, rather than something meaningful.

Focusing on statistical significance can lead you down other dangerous paths. For example, relying on statistical tests of assumptions instead of visualizations or practical metrics can lead you to believe that your model is valid when it is not. Using a statistical testing approach to select features can often result in incorrect choices about feature contributions, as well as poorer models.

A related issue is **p-hacking**, which occurs when you try many different models, features, or other aspects of the model until you find one that is statistically significant. This is a problem because it can reflect spurious results, and make it difficult to generalize the results of the model (overfitting). It also means you ignored null results, which can be just as informative as significant ones, a problem known as the **file drawer problem**.

15.1.2 Ignoring complexity

While techniques like standard linear/logistic regression and GLMs are valid and very useful, for many modeling contexts they may be too simple to capture the complexity of the data generating process, a form of underfitting. On the other side of the coin, many applications of statistical models ignore model assessment on a separate dataset, which can lead to overfitting. This makes generalization of such results more problematic. Those applications typically use a single model as well, and so they may not be indicative of the best approach that could be taken. It'd be better to have a few models of varying complexity to explore.

15.1.3 Using outdated techniques

If you wanted to go on a road trip, would you prefer a 1973 Ford Pinto or a Tesla Model S? If you want to browse the web, would you prefer to use a computer from the 90s and 56k modem, or a modern laptop with a high-speed internet connection? In both cases, you could potentially get to your destination or browse the web, but the experience would be much different, and you would likely have a clear preference[1]. The same goes with the models you use for your data analysis.

This is not specific to the statistical linear modeling realm, but there are many applications of statistical models that rely on outdated techniques, metrics, or other tools that solve problems that don't exist anymore. For example, using stepwise/best subset regression for feature selection is not really viable when more principled approaches like the lasso are available. Likewise, we can't really think of a case where something like MANOVA/discriminant function analysis would provide the best answer to a data problem, or where a pseudo-R^2 metric would help us understand a model better or make a decision about it.

Statistical analysis has been around a long time, and many of the techniques that have been developed are still valid, useful, and very powerful. But some reflect the limitations of the time in which they were developed. Others were an attempt to take something that was straightforward for simpler settings (e.g., linear regression) and apply to settings where it doesn't make sense (nonlinear, non-Gaussian, etc.). Even when still valid, there may be better alternatives available now.

15.1.4 Simpler is not necessarily more interpretable

Standard linear models are often used because of their interpretability, but in many of these modeling situations, interpretability can be difficult to obtain without using the same amount of effort one would for more complex models. Many statistical/linear models employ interactions, or nonlinear feature-target relationships (e.g., GLM/GAMs). If your goal is interpretability, these settings can be as difficult to interpret as features in a random forest. They still have the added benefit of more reliable uncertainty estimation. But you should not assume you will have a result as simple as a coefficient in a linear regression just because you didn't use a deep learning model.

15.1.5 Model comparison

When comparing models, especially in the statistical modeling realm, many will use a statistical test to compare them. An example would be using an ANOVA or likelihood ratio test to compare a model with and without interactions. Unfortunately, this doesn't actually tell us how the models perform under

[1] Granted, if it was a Pinto wagon, the choice could be more difficult.

realistic settings, and it comes with the usual statistical significance issues, like using an arbitrary threshold for claiming significance. You could basically claim that one terrible model is statistically better than another terrible model, but there isn't much value in that.

Some like to look at R^2 to compare models, but it has a lot of problems. People think it's more interpretable than other options, yet there is no value of 'good' you can universally apply, even in very similar scenarios. It can arbitrarily increase with the number of features whether they are actually predictive or not, and it doesn't tell you how well the model will perform on new data. It can also simply reflect that you have time-series data, as you are just witnessing spurious correlations over time. In short, you can use it to get a sense of how your predictions *correlate* with the target, but that can be a fairly limited assessment.

The following plot shows 250 simulations with a sample size of 100 and 40 completely meaningless features used in a linear regression. The R^2 values would all suggest the model is somewhat useful, with an average of ~.4. The adjusted R^2 values average zero, which is correct, but they can only average that by being negative, which is a meaningless value. Many of the adjusted values still get into higher areas that would be viable for some domains[2].

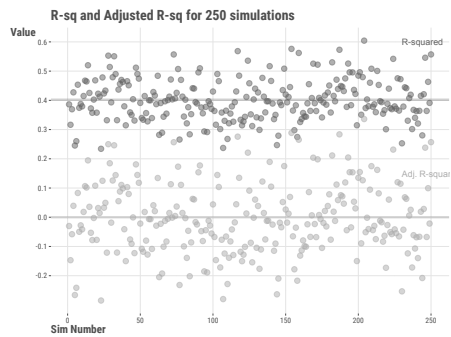


Figure 15.1: The problem of R^2.

Other commonly used metrics, like AIC, might be better in theory for model comparison. But they approximate the model selection one would get through cross-validation, so why not just do the cross-validation as due diligence? Furthermore, as long as you are using those metrics only on the training data, you probably aren't getting a good idea of how the model will generalize (Section 10.4).

[2]Also, adjusted R^2 would not be *practically* different from R^2 except for very small data situations.

Garden of Forking Paths

A common issue in statistical and machine learning modeling is the **garden of forking paths**. This is the idea that there are many different ways to analyze a dataset, and that the results of these analyses can be very different. When you don't have a lot of data, or when the data is complex and the data generating process is not well understood, there can be a lot of forks that lead to many different models with varying results. In these cases, the interpretation of a single model from the many that are actually employed can be misleading and can lead to incorrect conclusions about the data.

15.2 Estimation

15.2.1 What if I just tweak this...

From traditional statistical models to deep learning, the more you know about the underlying modeling process, the more apt you are to tweak some aspect of the model to try and improve performance. When you start thinking about changing optimizer options, link/activation functions, learning rates, etc., you can easily get lost in the weeds. This would be okay if you knew ahead of time it would make a big difference. However, in many, or maybe even most cases, this sort of tweaking doesn't improve model results by much, or there are ways to not have to make the choice in the first place such as through hyperparameter tuning (Section 10.7). More to the point, if this sort of 'by-hand' parameter tweaking does make a notable difference, that may suggest that you have a bigger problem with your model architecture or data.

For many tools, a lot of work has been done for you by folks who had a lot more time to work on these aspects of the model, and who will attempt to provide 'sensible defaults' which can work pretty well. There is still plenty we need to explore, and maybe a lot with more complex models such as boosting or deep learning. Even so, when you've appropriately tuned over the parameters that need it, you'll often find the results are not that different from what are otherwise notably different parameter settings.

15.2.2 Everything is fine

There is a flip side to the previous point, and that is that many assume that the default settings for complex models are good enough. We all do this when venturing into the unknown, but we do so at our own risk. Many of the more complex models have defaults geared toward a 'just works' setting rather than

a 'production' setting. For example, the default number of boosting rounds for xgboost will rarely be adequate[3]. Again, an appropriately tuned model should cover your bases.

15.2.3 Just bootstrap it!

When it comes to uncertainty estimation, many common modeling tools leave that to the user, and when the developers are pressed on how to get uncertainty estimates, they will often suggest to just bootstrap the result. While the bootstrap is a powerful tool for inference, it isn't appropriate just because you decide to use it. The suggestion to use bootstrapping is often made in the context of a complex modeling situation where it would be very (prohibitively) computationally expensive, and in other cases the properties of the results are not well understood. Other methods of prediction inference, such as conformal prediction, may be better suited to the task. In general, if a package developer suggests you bootstrap because their package doesn't have any means of uncertainty estimation, you should be cautious. If it's the obvious option, it should be included in the package.

While we're at it, another common suggestion is to use a quantile regression (Section 9.5) approach to get prediction intervals. This is a valid option in some cases, but it's not clear how appropriate it is for complex models or for certain types of outcomes, and modeling tools for predicting quantiles are not typically available for a given model implementation.

15.3 Machine Learning

15.3.1 General ML modeling issues

We see a lot of issues with machine learning approaches, and many of them are the same as those that come up with statistical models, but some are more unique to the machine learning world. A starting point is that many forget to create a baseline model, and instead jump right into a complicated model. This is a problem because it is hard to improve performance if you don't know what a good baseline score is. So create that baseline model and iterate from there.

A related point is that many will jump into machine learning without fully investigating the data. Standard exploratory data analysis (EDA) is a prerequisite for *any* modeling and can go a long way toward saving time and effort

[3]The number is actually dependent on other parameters, like whether early stopping is used, the number of classes, etc.

in the modeling process. It's here you'll find problematic cases and features, and can explore ways to deal with it.

When choosing a model or set of models, one should have a valid reason for the choice. Some less stellar reasons include using a model just because it seems popular in machine learning. And as mentioned with other types of models, you want to avoid using older methods that really don't perform well in most situations compared to others[4].

15.3.2 Classification

Machine learning is not synonymous with a classification problem, but this point seems to be lost on many. As an example, many will split their target just so they can do classification, when the target is a more expressive continuous variable. This is a problem because you are unnecessarily diminishing the reliability of the target score, and losing information about it. This can lead to a well-known statistical issue: **attenuation of the correlation** between variables, which the following demonstrates.

Python

```
import numpy as np
import pandas as pd

def simulate_binarize(
    N = 1000,
    correlation = .5,
    num_simulations = 100,
    bin_y_only = False
):
    correlations = []

    for i in range(num_simulations):
        # Simulate two variables with the given correlation
        xy = np.random.multivariate_normal(
            mean = [0, 0],
            cov = [[1, correlation], [correlation, 1]],
            size = N
        )

        # binarize on median split
```

[4]As we mentioned in the statistical section, many older methods are still valid and useful. But it's not clear what would be gained by using things like a basic support vector machine or knn-regression related to more recently developed or other techniques that have shown more flexibility.

```
        if bin_y_only:
            x_bin = xy[:, 0]
        else:
            x_bin = np.where(xy[:, 0] >= np.median(xy[:, 0]), 1, 0)
        y_bin = np.where(xy[:, 1] >= np.median(xy[:, 1]), 1, 0)

        raw_correlation = np.corrcoef(xy[:, 0], xy[:, 1])[0, 1]
        binarized_correlation = np.corrcoef(x_bin, y_bin)[0, 1]

        correlations.append({
            'sim': i,
            'raw_correlation': raw_correlation,
            'binarized_correlation': binarized_correlation
        })

    cors = pd.DataFrame(correlations)
    return cors

simulate_binarize(correlation = .25, num_simulations = 5)
```

R

```
simulate_binarize = function(
    N = 1000,
    correlation = .5,
    num_simulations = 100,
    bin_y_only = FALSE
) {
    correlations = list()

    for (i in 1:num_simulations) {
        # Simulate two variables with the given correlation

        xy = MASS::mvrnorm(
            n = N,
            mu = c(0, 0),
            Sigma = matrix(c(1, correlation, correlation, 1),
            nrow = 2),
            empirical = FALSE
        )

        # binarize on median split
```

```
        if (bin_y_only) {
            x_bin = xy[, 1]
        } else {
            x_bin = ifelse(xy[, 1] >= median(xy[, 1]), 1, 0)
        }

        y_bin = ifelse(xy[, 2] >= median(xy[, 2]), 1, 0)

        raw_correlation = cor(xy[, 1], xy[, 2])
        binarized_correlation = cor(x_bin, y_bin)

        correlations[[i]] = tibble(
            sim = i,
            raw_correlation,
            binarized_correlation
        )
    }

    cors =  bind_rows(correlations)
    cors
}

simulate_binarize(correlation = .25, num_simulations = 5)
```

The following plot shows the case where we only binarize the target variable for 500 simulations. The true correlation between the raw and binarized variables is .25, .5, or .75, but the correlation in the binarized case is notably less. This is because the binarization process has removed the correlation between the variables.

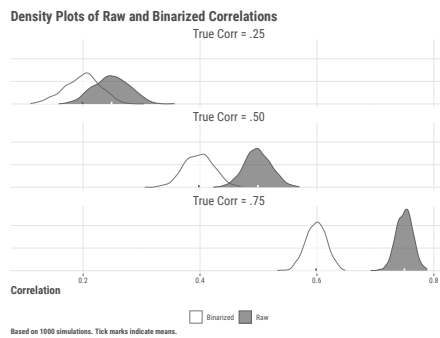


Figure 15.2: Density plots of raw and binarized correlations.

Common issues with ML classification don't end here, however. Another problem is that many will use a simple .5 cutoff for binary classification, when it is probably not the best choice in most classification settings. Related to this, many only focus on accuracy as a metric for performance. Others are more useful in many situations, or just add more information to assess the model. Each metric has its own pros and cons, so you should evaluate your model's performance with a suite of metrics.

15.3.3 Ignoring uncertainty

It is very common in ML practice to ignore uncertainty in predictions or metrics. This is a problem because there is always uncertainty, and acknowledging that it exists can help one have better expectations of performance. This is especially true when you are using a model in a production setting, where the model's performance can have real-world consequences.

It is often computationally difficult to get uncertainty estimates for many of the black-box techniques that are popular in ML. Some might suggest that there is enough data such that uncertainty is not needed, but this would have to be demonstrated in some fashion. Furthermore, there is always increased uncertainty for prediction on new data and for smaller subsets of the population we might be interested in. In general, there are ways to get uncertainty estimates for these models, e.g., bootstrapping, conformal prediction, and simulation, and it is often worth the effort to do so.

15.3.4 Hyperfocus on feature importance

Researchers and businesses often have questions about which features in an ML model are important. Yet this can be a difficult question to answer, and the answer is often not practically useful. For example, most models used in ML are going to have interactions, so the importance of any single feature is likely going to depend on other features in the model. If you can't disentangle the effects of one feature from another, then trying to talk about a single feature's relative worth is often a misguided endeavor, even if you use an importance metric that tries to account for the interaction.

Even if we can deem a variable 'important', this doesn't imply a causal relationship, and it doesn't mean that the variable is the best of the features you have. In addition, other metrics, which might be just as valid, may provide a different rank ordering of importance.

What's more, just because an importance metric may deem a feature as not important, that doesn't mean it has no effect on the target. It may be that the feature is correlated with other features that are more important, and so the metric is just reflecting that. It may also just mean that the importance metric is not well suited to assessing that particular feature's contribution.

As we have seen (Section 5.9), the reality is that multiple valid measures of importance can come to different conclusions about the relative importance of a feature, even within the same model setting. One should be very cautious in how they interpret these.

> **SHAP for Feature Importance**
>
> SHAP values are meant to assess *local*, i.e., observation level, feature contributions to a prediction. They are also used as *global* features of importance in many ML contexts, even though they are not meant to be used this way. Doing so can be misleading, and often, average SHAP values will just reflect the distribution of the feature more than its importance. SHAP can also be notably inconsistent with other metrics even in simple settings.

15.3.5 Other common pitfalls

A few other common pitfalls in ML modeling include:

- Forgetting that the data is more important than your modeling technique. You will almost always get more mileage out of improving your data than you will out of improving your model.
- Ignoring data leakage. Letting training data leak into the test set. As a simple example, consider if we use random validation splits with time-based data. This would allow the model to train with future data it will ultimately be assessed on. That may be an obvious example, but there are many more subtle ways this can happen. Data leakage gives your model an unfair advantage when it is time for testing, leading you to believe that your model is doing better than it really is.
- Forgetting you will ultimately need to be able to explain your model to someone else. The only good model is a useful one; if you can't explain it to someone, you can't expect others to trust you with it or your results.
- Assuming that grid search is good enough for all or even most cases. Not only is it computationally expensive, but you can easily miss valid tuning parameter values that are outside of the grid. Many other methods are available that more efficiently search the space and are as easy to implement.
- Thinking deep learning will solve all your problems. If you are dealing with standard, tabular data, at present deep learning will often just increase computational complexity and time, with no guarantee of increased performance, and often does notably worse. Hopefully this will change in the future, but for now, you should not expect major performance gains.
- Comparing models on different datasets. If you run different models on

separate data, there is no objective way to compare them. As an example, the accuracy may be higher on one dataset just because the baseline rate is much higher.

The list goes on. In short, many of the pitfalls in ML modeling are the same as those in statistical modeling, but there are some unique to or more common in the ML world. The most important thing to remember is that due diligence is key when conducting any modeling exercise, and ML doesn't change that. You should always be able to explain and defend your model choices and results to someone else.

15.4 Causal Inference

Causal inference and modeling is hard. Very hard.

15.4.1 The hard work is done before data analysis

The most important part of causal modeling is the conceptualization of the problem and the general design of the study to answer the specific questions related to that problem. You have to think very hard about the available data, what variables may be confounders, which effects may be indirect, and many other aspects of the process you want to measure. A causal model is the one you draw up, possibly before you even start collecting data, and it is the one you use to guide your data collection and ultimately your data modeling process.

15.4.2 Models can't prove causal relationships

Causal modeling focuses on addressing issues like confounding, selection bias, and measurement error, which can skew interpretations about cause and effect. While predictive accuracy is key in some scenarios, understanding these issues is crucial for making valid causal claims.

A common mistake in modeling is assuming that a model can prove causality. You can have a very performant model, but the model results cannot prove that one variable causes another just because it is highly predictive. There is also nothing in the estimation process that can magically extract a causal relationship even if it exists. Reality is even more complex than our models, and no model can account for every possibility. Causal modeling attempts to account for some of these issues, but it is limited by our own biases in thinking about a particular problem.

Predictive features in a model might reflect a true causal link, act as stand-ins for one, or merely reflect spurious associations. Conversely, true causal effects of a feature may not be large, but it doesn't mean they are unimportant. Assuming you have done the hard work of developing the causal structure beforehand, model results can provide more confidence in your ultimate causal conclusions, and that is very useful, despite lingering uncertainties.

15.4.3 Random assignment is not enough

Many believe experimental design is the gold standard for making causal claims, and it is certainly a good way to control for various aspects that can make causal claims difficult. Consider a randomized control trial (RCT) where you assign people to a treatment or control group. The left panel in Figure 15.3 shows the overall treatment effect, where the main effect would suggest a causal conclusion of no treatment effect. However, the right panel shows the same treatment effect across another group factor, and it is clear that the treatment effect is not the same across groups.

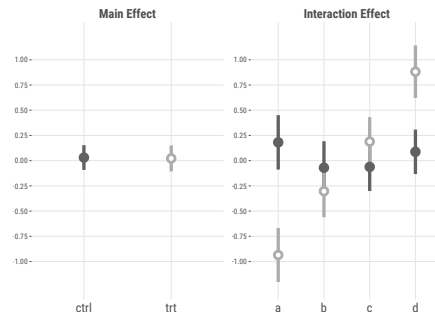


Figure 15.3: Main effect vs. interaction.

So random assignment cannot save us from misunderstanding the causal mechanisms at play. Other issues to think about are that the treatment may be implemented poorly, participants may not be compliant, or the treatment may not even be well defined, and these are not uncommon situations. This comes back to having the causal structure understood as best as you can before any analysis.

15.4.4 Ignoring causal issues

Causal modeling is concerned with things like confounding, selection bias, measurement error, reverse causality and more. These are all issues that can lead to incorrect causal conclusions. A lot of this can be ignored when predictive performance is of primary importance, and some can be ignored when we

are not interested in making causal claims. But when you are interested in making causal claims, you will have some work to do in order for your model to help you make said claims, regardless of the modeling technique you choose to implement. And it doesn't hurt to be concerned about these issues in noncausal situations.

15.5 Data

When it comes to data, plenty can go wrong before even starting with any modeling attempt. Let's take a look at some issues that can regularly arise.

15.5.1 Transformations

Many models will fail miserably without some sort of scaling or transformation of the data. A few techniques, like tree-based approaches, do not benefit, but practically all others do. At the very least, models will converge faster and possibly be more interpretable. However, you should generally not use transformations that would lose the expressivity of the data, because as we noted with binarization (Section 15.3.2), some can do more harm than good. But you should always consider the need for transformations, and not just assume that the data is in a form that is ready for modeling.

15.5.2 Measurement error

Measurement error is a common issue in data collection, and it can lead to biased estimates and reduce our ability to detect meaningful feature-target relationships. Generally speaking, the reliability of a feature or target is its ability to measure what it's supposed to, while measurement error reflects its failure to do so. There is no perfectly measured variable, and measurement error can come from a variety of sources and be difficult to assess. But it is important to try and understand how well your data reflects the construct it is supposed to. If you can't correct for a measurement problem, for example, by finding better data, you should at least be aware of the issue and consider how they might affect your results. There is a saying about squeezing blood from a stone, or putting lipstick on a pig, or something like that, and it applies here. If your data is poor, your model won't save it.

15.5.3 Simple imputation techniques

Imputation may be required when you have missing data, but it can be done in ways that don't help your model. Simple imputation techniques, like using the mean or modal category, can produce faulty, or at best, noisier, results.

First you should consider why you want to keep a feature that doesn't have a lot of data. Do you even trust the values that are present? If you really need to impute, use an actual model to do so, but recognize that the resulting value has uncertainty associated with it. There are practical problems with implementing techniques to incorporate the uncertainty (Section 14.3.4), so there is no free lunch there. But at least having a better imputation model will provide a better guess than a mean, and still better is to use a model that would handle the missing values natively, like tree-based methods that can split on the missingness.

15.5.4 Outliers are real!

One common practice in modeling is to drop or modify values considered as "outliers". However, extreme values in the target variable are often a natural part of the data. Assuming there is no actual error in recording them, often, a simple transformation can address the issue. If extremes persist after modeling, it indicates that the model is unable to capture the underlying data structure, rather than an inherent problem with the data itself. Additionally, even values that may not appear extreme can still have large residuals, so it's important not to solely focus on just the most extreme observed values.

In terms of features, extreme values can cause strange effects, but often they reflect a data problem (e.g., incorrect values) or can be resolved using the transformations you should already be considering (e.g., taking the log). In other cases, they don't really cause any modeling problems at all. And again, some techniques are fairly robust to feature extremes, like tree-based methods.

15.5.5 Big data isn't always as big as you think

Consider a model setting with 100,000 samples. Is this large? Let's say you have a rare outcome that occurs 1% of the time. This means you have 1000 samples where the outcome label you're interested in is present. Now consider a categorical feature (A) that has four categories, and one of those categories is relatively small, say 5% of the data, or 5000 cases, and you want to interact it with another categorical feature (B), one whose categories are all equally distributed. Assuming no particular correlation between the two, you'd be down to ~1% of the data for the least category of A across the levels of B. Now if there is an actual interaction effect on the target, some of those interaction cells may have only a dozen or so positive target values. Odds are pretty good that you don't have enough data to make a reliable estimate of that effect unless it is extremely large.

Oh wait, did you want to use cross-validation also? A simple random CV approach might result in some validation sets with no positive values in those interaction groups at all! Don't forget that you may have already split your 100,000 samples into training and test sets, so you have even less data to start

with! The following table shows the final cell count for a dataset with these properties.

Start N	Train N	A p	B p	5cv	Final Cell p	Cell N	Target N in Cell
100,000	80,000	0.05	0.25	0.20	0.0025	200	2

The point is that it's easy to forget that large data can get small very quickly due to class imbalance, interactions, etc. There is not much you can do about this, but you should not be surprised when these situations are not very revealing in terms of your model results.

15.6 Wrapping Up

Though we've covered many common issues in modeling here, there are plenty more ways we can trip ourselves up. The important thing to remember is that we're all prone to making and repeating mistakes in modeling. But awareness and effort can go a long way, and we can more easily avoid these problems with practice. The main thing is to try and do better each time, and learn from any mistakes you do make.

15.6.1 The common thread

Many of the issues here are model-agnostic and could creep into any modeling exercise you undertake.

15.6.2 Choose your own adventure

If you've made it through the previous chapters, there's only one place to go. But you might revisit some of those in light of the common problems we've discussed here.

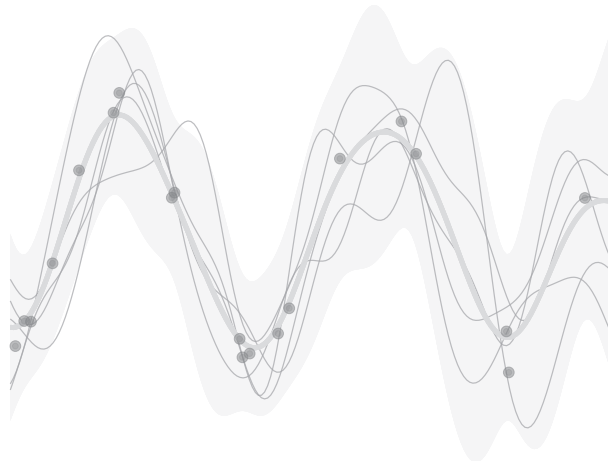
15.6.3 Additional resources

Mostly we recommend the same resources we did in the corresponding sections of the previous chapters. However, a couple of others to consider are:

- Shalizi (2015) (start with the fantastic concluding comment)
- Questionable Practices in Machine Learning (Leech et al. 2024)

16

Parting Thoughts



As we wrap things up, let's revisit some of the key points we've covered in this text and talk more about the modeling process in general.

16.1 How to Think About Models

When we first started our discussion of models in data science (Chapter 2), we talked about how a model is a simplified representation of reality. They start as ideas based on our intuition or experience, and they can sometimes be very simple ones. But at some point we start to think of them more formally, as a step toward testing those ideas in the real world. For statistics, machine learning, and data science more generally, models are then put into mathematical equations that give us a common language to reference them by. This does not have to be complex though. As an example, most of the models you've seen so far can be expressed as follows:

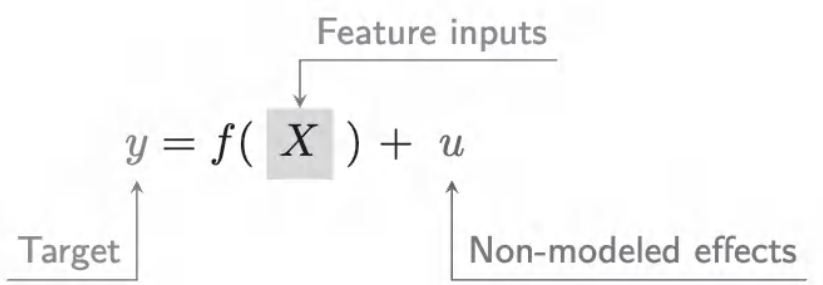


Figure 16.1: Generic model.

In words, this equation says that the target variable y is a function of the feature inputs X, along with anything else that we don't include in that set. This is a basic form of a model, and it's the same for linear regression, logistic regression, and even random forests and neural networks[1].

To aid our understanding beyond the math, we try to visually express models in a variety of ways[2], as in the following images.

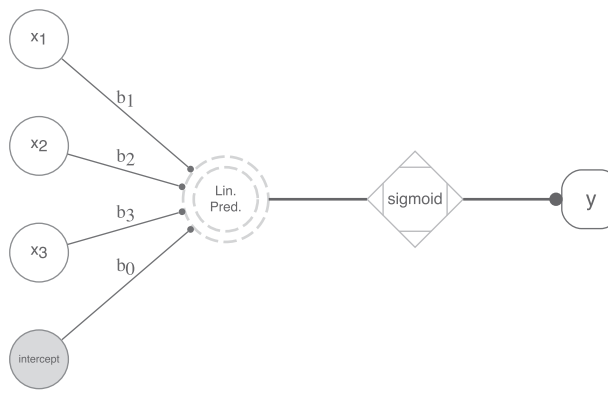


Figure 16.2: Logistic Regression Model

[1]Neural networks are a bit different in that they can be thought of as a series of (typically nested) functions that are applied to the data, but they can still be expressed in this form e.g., $h(g(f(X)))$. The functions are more complex, and the parameters are estimated in a different way than typical tabular data models, but the basic idea is the same.

[2]The LDA model depicted from Wikipedia was one of the early machine learning models for understanding natural language, and in particular to extract topics from text. It was a lot of fun to play with these, but it took a lot of pre-processing of text to get them to work at all, and they were performed pretty poorly in practice. That model may look like something peculiar, but it's not much more than a flexible PCA on a matrix of word counts, or from another perspective, a Bayesian multinomial model.

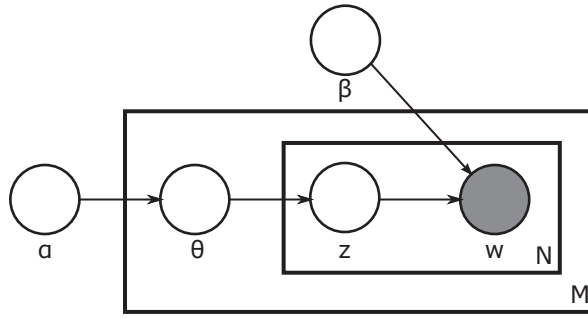


Figure 16.3: Plate Notation for Latent Dirichlet Allocation

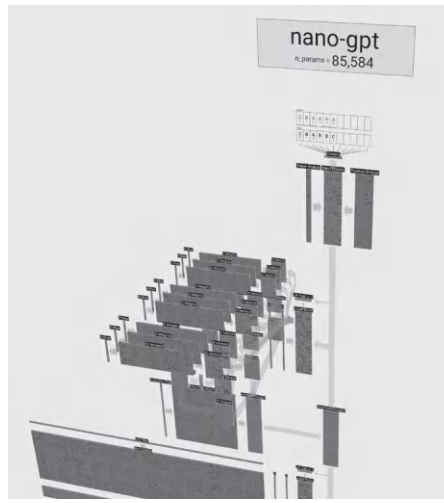


Figure 16.4: Just part of the Nano GPT model

But even now these models are still at the idea stage, and we ultimately need to see how they work in the world, make predictions, and help us to make informed decisions. We've seen how to do this with linear models of various forms, and more unusual model implementations in the form of tree-based models, and even highly complex neural networks. These are the tools that allow us to take our ideas and turn them into something that can be used to make decisions, and that's the real power of using models in data science.

16.2 More Models

When choosing a model, there's a lot at your disposal, and we've only scratched the surface of what's out there. Here are a few more models that you may encounter in your data science journey:

Statistical Models

In the statistical realm there are many more models that focus on different target distributions and types. For instance, we might use a beta distribution for targets between 0 and 1, ordinal logistic regression for ordinal targets, or survival models for time-to-event outcomes. Some models are field-specific, like two-stage least squares in econometrics. Beyond these, specific implementations will be found for time series (ARIMA, state space models), spatial data (kriging, CAR), and other special target considerations. Most of these models are essentially linear models with slight modifications.

Nonlinear models are another realm, which are a bit different from the nonlinear aspects of GLMs, GAMs, or deep learning. These models assume a specific (nonlinear) functional form and can be used to explore relationships that are not well captured by standard linear models. Examples range from something as simple as a polynomial regression or logistic growth model, to more complex biological and epidemiological models. These approaches are not as flexible as GAMs, or as predictive as neural networks, but they can potentially be useful in the right context.

In addition, there are 'multivariate' techniques like PCA, factor analysis, and similar ones which are still pretty widely used. There are also cases where the primary target is multivariate in nature, meaning a standard regression with multiple outcomes. These are more common within some areas like economics and psychology.

Machine Learning

In a purely machine learning context, you may find other models beyond those just mentioned in the statistical realm. However, as we have mentioned several times at this point, potentially any model can be used with machine learning, including statistical models. The machine learning context prioritizes prediction, and many models used would not usually produce standard statistical output like coefficients and uncertainty estimates by default. Examples include support vector machines, k-nearest neighbors regression, and other techniques. Most of these traditional 'machine learning models' have fallen out of favor due to their inflexibility with heterogeneous data types, and/or poor performance or efficiency compared to more modern approaches. However, even then, their spirit may live on in modern approaches.

You'll also find models that focus on ranking, either with an outcome of ranks requiring a specific loss function (e.g., LambdaRank), or where ranking is used to simplify decision-making through post-estimation ranking of predictions (e.g., decile ranking, uplift modeling). In addition, you can find machine learning techniques extended to survival, ordinal, and other situations that are more common in the statistical realm.

Other areas of machine learning, like reinforcement learning, recommender systems, network analysis, and unsupervised learning techniques, provide more options that might be useful. Plenty is left for you to explore here as well!

Deep Learning

When it comes to deep learning, it seems there is a new model every day, and it's hard to keep up. In general, convolutional neural networks are still the go-to for many types of computer vision tasks, while transformers are commonly used for natural language processing, but both have been applied to the other domain with success. Many 'foundational' models have been developed that allow you to apply pretrained models to your specific problem, and form the basis of modern AI. For tabular data as we've focused on here, you'll typically see some variant of MLPs, often with embeddings for categorical features. Some have attempted transformers and CNNs here as well, but results are mixed.

The deep learning landscape also includes models like deep graphical networks, and deep Q learning for reinforcement learning, specific models for image segmentation (e.g., SAM), recurrent neural network variants for time-series data, and generative adversarial networks for a variety of tasks. Some specific techniques are falling out of favor, as transformer-based architectures are being applied to seemingly everything. But the field is dynamic, and it remains to be seen which methods will prevail in the long run.

> **i** List of Models
>
> You can find a list of some specific models for each of these categories in the appendix (web only). It is by no means an exhaustive list, but it should give you a good starting point for exploring additional models once you're finished here.

16.3 Families of Models

While there are many models out there, even if we restrict the discussion to tabular data, we can group them in a fairly simple way that would cover most of the standard problems you'll come across.

GLM and Related: Interpretable Insights

Here we have standard linear models with a focus on interpretability. Basically anything you'd find in a traditional stats or econometrics textbook would belong to this 'family'.

- Includes: GLM, survival, ordinal, time-series, other distributions (beta, tweedie) - Best for: small data situations (samples and features), a baseline model, a causal model, post-model analysis of the results from more complex models - Primary strength: ease of estimation, interpretability, uncertainty estimation - Primary weakness: relatively poor prediction, may not capture natural data complexity without additional work

Penalized Regression and Friends: Predictive Progress

This family encompasses techniques that could be used as stepping stones toward machine learning. These include linear models enhanced with regularization, and advanced statistical models that deliberately incorporate nonlinearities and other complexities. Moreover, our emphasis begins to shift more to prediction in this context, though these models still provide relatively easier interpretation compared to the next group.

- Includes: lasso/ridge, mixed models, GAMs, Bayesian
- Best for: small to large data, possibly a relatively large number of features (esp. lasso), baseline model
- Primary strength: increased predictive capability while maintaining interpretability
- Primary weakness: interpretability can decrease, estimation difficulty can start to arise (convergence issues, uncertainty)

Trees and Nets: Champion's Choice

This family includes tree-based models and neural networks, which are almost exclusively focused on predictive performance by default and represent a significant increase in complexity and computational requirements.

- Includes random forests, gradient boosting, neural networks ('basis function models')
- Best for: prediction/performance
- Primary strength: prediction, ability to handle potentially very large data and numbers of features

- Primary weakness: interpretability and uncertainty estimation

Thinking about families or groups of models can do a lot to help demystify the modeling process. You could come up with other schemas within a specific data domain or group of models, there's no solid rule here. But it can be helpful to compartmentalize the models so that you don't get overwhelmed by what are often minor details that won't significantly impact the practical application.

The differences between the model families are not substantial, particularly between the first two. Specific models may only differ in the likelihood function, the penalty term, or just a shift in focus. The third group is a bit different, but it mostly just extends the application of nonlinear and interaction effects we can implement from the first groups, allowing for more computational capacity and flexibility. But if you're new to modeling or dabbling in a new area, we think this grouping can quickly help you understand what you're looking at and what you might want to use for a given problem. As you do more modeling, you'll likely come up with your own.

16.3.1 A simple modeling toolbox

In practice, just a handful of techniques from this text can provide a lot of modeling power. Here's a simple toolbox that can cover a lot of the ground you'd need in a typical data science project:

- **Penalized Regression**: Lasso, ridge, GAMs, mixed models and similar methods keep things linear while increasing predictive power and accommodating more features than their non-penalized counterparts. If you need to focus more on the explanatory and statistical side of things, you can use the standard GLM.
- **Boosting/Tree-Based Models**: At the time of this writing, boosting methods consistently deliver the best predictive performance for tabular data, and they are quite computationally efficient relative to deep learning techniques. That's reason enough to know how to use them and keep them handy.
- **Basic Deep Learning Model**: A 'simple' deep learning model that incorporates embeddings for categorical and text features is a very powerful tool[3]. Additionally, using a deep learning approach can be integrated with other DL models that process different types of data, such as images or text, to enhance predictive performance. We're still working toward an implementation of deep learning that can handle any tabular data we throw at it, but we're not quite there yet.

Besides the models, it's crucial to understand how to evaluate your models (cross-validation, metrics), how to interpret them (coefficients, SHAP, feature

[3]Some seem to think that deep learning is only deep learning if it's a transformer-based model or a convolutional neural network. However 'deep' is not formally defined, and we more simply see a deep learning model as just a model with multiple layers.

importance, uncertainty), and how to manage the data you're working with. While we've discussed many topics in the text, there's always more to learn, and more to practice.

> **The Tweener**
>
> Despite getting an occasional shout-out, GAMs appear to still be quite underrated in the data science community, probably because the tools in Python to implement and explore them are relatively lacking[4]. On the plus side, the tools in R are excellent, though they can take some getting used to.
> GAMs are a great way to handle nonlinear relationships, interactions, and add penalization, all in a way that can be more interpretable than boosting or a neural network. They can be used in a wide variety of data situations and can be a great way to get more predictive power while staying within a traditional linear model setting. If you're looking for a way to get a bit more out of your linear models without diving into deep learning, GAMs are a great place to start and are often a tough model to beat for those who know how to use them.

16.4 How to Choose?

So how should we choose a specific model for our data? People love to say that 'all models are wrong, but some are useful'[5]. We prefer to think of this a bit differently. There is no (necessarily) wrong model to use to answer your question, and there's no guarantee that you would come to a different *practical* conclusion from using a simple correlation than you would from a complex neural network. But some models can be more useful depending on the context and the question you're asking.

In the end, nothing says you can't use multiple models to answer your question, and in fact, this is often a good idea assuming you have the time and resources to do so. As we've talked about, you can use a simple model to get a baseline, and then use a more complex model to see if you can improve on that. You can use a model that's easy to interpret to get a sense of what's going on, and

[4]Until Python can go from model to visualizing the marginal effect with uncertainty in two or three lines of code (even with a Bayesian implementation), possibly on millions of observations in a few seconds, and even visualizing the derivatives (also with uncertainty estimates), it's not going to be as easy to use as R for GAMs. But here's hoping the current efforts continue there.

[5]George Box, a famous statistician, said this in 1976.

then use a model that's better at prediction. Even when your primary focus is prediction, you can often combine models to potentially get a better result.

And that's the main thing – you don't have to restrict yourself when it comes to modeling in data science, and you shouldn't. The key is to understand what you're trying to do, and to use the right tools for the job.

16.5 Choose Your Own Adventure

We've covered a lot of ground in this text, and we hope you've learned something new along the way. But there's so much more out there for you to continue to explore. We hope that you'll be able to take what you've learned here and apply it to your own work, and that you'll continue to learn and grow as a data scientist.

So where do you go from here? The world of data science is vast – choose your own adventure!

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

A

Acknowledgments

This work was supported by no grants or grad students, and it was done entirely in the spare time of the authors after their long days of work and teaching. We do not recommend this approach, but it appears we eventually got it done! But we could not have done this without the support of our families, who have been patient and understanding throughout the process.

In particular, Michael wishes to thank his wife Xilin, whose insights, humor, support, and especially patience, have been invaluable throughout the process. He'd also like to thank Rich Herrington, whose encouragement was the catalyst for his shift from psychology to data science many years ago. Michael also thanks his fellow consultants at the University of Michigan whose thoughtful discussions shaped much of his analytical thinking, as well as the folks at Strong Analytics and OneSix who supported his transition from academia to industry and made him much better at everything he does in the realm of data science. And finally, he would like to thank the many mentors in the statistical, machine learning, deep learning/AI, and programming communities, who never knew they were such a large influence but did much to expand his knowledge in many ways over the years.

Seth is grateful for the support and understanding of his wife Megan, and his fantastic kids, who never cease to bring him joy. He'd also like to thank his colleagues and students at Notre Dame who have supported his teaching antics and research shenanigans. He hopes that this book provides at least some evidence that he knows what he's talking about.

Together we'd like to thank those who helped with the book by providing comments and feedback, specifically: Malcolm Barrett, Isabella Gehment, Demetri Pananos, and Chelsea Parlett-Pellereti. Their insights and suggestions were invaluable in making this book a lot better. We'd also like to thank others who took the time for a quick glance to see what was going on and provide a brief comment here and there. Every contribution was appreciated.

Finally, we'd like to thank you, the reader, for taking the time to peruse this book. We hope you find it useful and informative, and that it helps you in your data science journey. If you have any feedback, please feel free to reach out to us. We'd love to hear from you.

Thank you!

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

B

Matrix Operations

Addition, subtraction, multiplication, and division. These are all things you already know how to do with single numbers. What happens, though, if you want to multiply two different matrices together. Does that simple, 'scalar' operation still translate if you have a $2x3$ matrix and a $3x2$ matrix? If words like matrix and scalar make you break out in a sweat, then this chapter is for you!

Matrix operations, especially multiplication, are critical for understanding core aspects of how modeling actually produces all these cool results that help us discover so many interesting things. Knowing the underlying mechanics of matrix operations helps to demystify several issues that you might run into with your models. It can also help to get the gist of various articles and papers that you might come across. Before we get into any operations, though, let's make sure we are together on some concepts.

A **scalar** is a single numeric value. It might help if you think about a scalar as a single 'block'.

Figure B.1: Scalar.

Code

```
scalar_example = 1 # scalar value in r or python
```

And just like we can line blocks up on the floor, we can put our scalars together to form a **vector**. A vector is a collection of scalars with a length of n. We can also think of a vector as a single row or column of scalars.

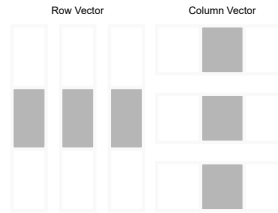


Figure B.2: Row and column vectors.

There are many ways to create a vector in R and Python. Here are a couple.

R

```
vector_example = 1:6
vector_example = c(1, 2, 3, 4, 5, 6)
vector_example = matrix(1:6, nrow = 1) # or ncol = 1
```

Python

```
import numpy as np

vector_example = range(5)
vector_example = [1, 2, 3, 4, 5] # as list
row_vector = np.array([1, 2, 3])  # create a row vector
column_vector = np.array([[1], [2], [3]])  # create a column vector
```

Now, we can take a few of our block vectors and stack them into a **matrix**, assuming the vectors are of the same size. A matrix is a two-dimensional collection of vectors, and it is the fundamental structure for tabular data and beyond.

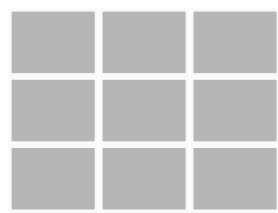


Figure B.3: Matrix.

And here is a matrix of specific values:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

If you think about most tables you've ever seen, you'll see that the simple matrix looks remarkably familiar!

R

```
matrix_example = matrix(1:6, nrow = 2, ncol = 3)
dim(matrix_example)
```

Python

```
matrix_example = np.array([[1, 2, 3], [4, 5, 6]])
matrix_example.shape
```

A matrix has two dimensions, rows and columns, which can be any size. When we talk about the dimensions of a matrix, we always make note of the rows first, followed by the columns. This matrix has two rows and three columns, so we have a 2×3, or 'two-by-three' matrix[1].

Beyond matrices, we can also have **tensors** – multi-dimensional **arrays** that generalize matrices to more than 2 dimensions. Tensors are widely used in machine learning applications, especially deep learning. For example, a 3D tensor might represent an image with dimensions for width, height, and color channels, visualized as a stack of matrices, or a cube of numbers. Even standard linear models can be conceptualized using tensor structures, such as when a third dimension extends the core data matrix to accommodate geographical regions (e.g., states) or other groupings (e.g., time points in longitudinal data).

[1]Numpy arrays/matrices are in **row major** order, while R is **column major** order. You'll note how with numpy we essentially provided two rows to the array function, which automatically created the 2×3 matrix. The R matrix is not the same, because by default it fills in the columns. If you add `by_row = TRUE`, you'd then get the same result as the numpy example. Column major is generally more intuitive for tabular data, because that's how we think of data stored in tables, and why the pandas package in Python is also column major/oriented. However, both R and Python are very flexible and more generally work in arrays. If you use both, it can take a bit to settle with one if you've used the other (especially for 'apply' functions). The reticulate package has a vignette that provides a nice overview, while the rray package actually brings the numpy approach to the R landscape.

B.1 Addition

Matrix addition, along with subtraction, is the easiest concept when dealing with matrices. There is one rule though: the matrices need to have the same dimensions. From a practical code perspective, if one is a scalar, addition of the scalar will be applied to every element in the matrix.

Let's check out these two matrices:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} \quad \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}}$$

You probably noticed that we gave each scalar within the matrix a label associated with its row and column position. We can use these to see how we will produce the new matrix.

Now, we can set this up as an addition problem to produce Matrix C:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} + \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} A_{11}+B_{11} & A_{12}+B_{12} & A_{13}+B_{13} \\ A_{21}+B_{21} & A_{22}+B_{22} & A_{23}+B_{23} \end{bmatrix}}$$

Now we can pull in the real numbers:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} + \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} 1+7 & 2+8 & 3+9 \\ 4+9 & 5+8 & 6+7 \end{bmatrix}}$$

Giving us Matrix C:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} + \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} 8 & 10 & 12 \\ 13 & 13 & 13 \end{bmatrix}}$$

First, let's create those matrices in R and Python.

R

In R, we can create a matrix with the `matrix` function or by row binding numeric vectors.

```
matrix_A = rbind(1:3, 4:6)

# The following is an equivalent
# to rbind:
# matrix_A = matrix(
#     c(1:3, 4:6),
#     nrow = 2,
#     ncol = 3,
#     byrow = TRUE
# )

matrix_B = rbind(7:9, 9:7)
```

Python

The task is just as easy in Python. We will import `numpy` and then use the `matrix` method to create the matrices:

```
import numpy as np

matrix_A = np.array([[1, 2, 3], [4, 5, 6]])

matrix_B = np.array([[7, 8, 9], [9, 8, 7]])
```

Once we have those matrices created, we can use the standard `+` to add them together:

R

```
matrix_A + matrix_B
```

```
     [,1] [,2] [,3]
[1,]    8   10   12
[2,]   13   13   13
```

```
matrix_A + 3
```

```
     [,1] [,2] [,3]
[1,]    4    5    6
[2,]    7    8    9
```

Python

Just like R, we can use `+` with those matrices.

```
matrix_A + matrix_B
```

```
array([[ 8, 10, 12],
       [13, 13, 13]])
```

```
matrix_A + 3
```

```
array([[4, 5, 6],
       [7, 8, 9]])
```

B.2 Subtraction

Take everything that you just saw with addition and replace it with subtraction! But just like addition, every matrix needs to have the same dimensions.

Here is the result:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} - \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} -6 & -6 & -6 \\ -5 & -3 & -1 \end{bmatrix}}$$

Subtracting matrices in R and Python is the same as addition, just using - instead.

R

```
matrix_A - matrix_B
```

```
     [,1] [,2] [,3]
[1,]   -6   -6   -6
[2,]   -5   -3   -1
```

```
matrix_A - 3
```

```
     [,1] [,2] [,3]
[1,]   -2   -1    0
[2,]    1    2    3
```

Python

```
matrix_A - matrix_B
```

```
array([[-6, -6, -6],
       [-5, -3, -1]])
```

```
matrix_A - 3
```

```
array([[-2, -1,  0],
       [ 1,  2,  3]])
```

B.3 Transpose

You might see a matrix denoted as A^T or A'. The superscripted T or $'$ for matrix **transpose**. If we transpose a matrix, all we are doing is flipping the rows and columns along the matrix 'main diagonal'. A visual example is much easier:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} -> \underset{\text{Matrix A transposed}}{\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}}$$

R

In R, all we need is the `t` function:

```
t(matrix_A)
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Python

In Python, we can use numpy's `transpose` method:

```
matrix_A.transpose()
```

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
matrix_A.T  # shorthand
```

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

B.4 Multiplication

Now you probably have some confidence in doing matrix operations. Just as quickly as we built that confidence, it will be crushed when learning about matrix multiplication.

When dealing with matrix multiplication, we have a huge change to our previous rule. No longer do our dimensions have to be the same! Instead, the matrices need to be *conformable* – the first matrix needs to have the same number of columns as the number of rows within the second matrix. In other words, the inner dimensions must match.

Look one more time at these matrices:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} \cdot \underset{\text{Matrix B}}{\begin{bmatrix} 7_{11} & 8_{12} & 9_{13} \\ 9_{21} & 8_{22} & 7_{23} \end{bmatrix}}$$

Matrix A has dimensions of 2×3, as does Matrix B. Putting those dimensions side by side – $2 \times 3 * 2 \times 3$ – we see that our inner dimensions are 3 and 2 and do not match.

What if we *transpose* Matrix B?

$$\underset{\text{Matrix B}^T}{\begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix}}$$

Now we have something that works!

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} \cdot \underset{\text{Matrix B}^T}{\begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix}} = \underset{\text{Matrix C}}{\begin{bmatrix} . & . \\ . & . \end{bmatrix}}$$

Now we have a $2 \times 3 * 3times2$ matrix multiplication problem! The resulting matrix will have the same dimensions as our two matrices' outer dimensions: 2×2.

Here is how we will get a 2×2 matrix:

$$\underset{\text{Matrix A}}{\begin{bmatrix} 1_{11} & 2_{12} & 3_{13} \\ 4_{21} & 5_{22} & 6_{23} \end{bmatrix}} \cdot \underset{\text{Matrix B}^T}{\begin{bmatrix} 7_{11} & 9_{12} \\ 8_{21} & 8_{22} \\ 9_{31} & 7_{32} \end{bmatrix}} =$$

$$\underset{\text{Matrix C}}{\begin{bmatrix} (A_{11} * B_{11}) + (A_{12} * B_{21}) + (A_{13} * B_{31}) & (A_{11} * B_{12}) + (A_{12} * B_{22}) + (A_{13} * B_{32}) \\ (A_{21} * B_{11}) + (A_{22} * B_{21}) + (A_{23} * B_{31}) & (A_{21} * B_{12}) + (A_{22} * B_{22}) + (A_{23} * B_{32}) \end{bmatrix}}$$

That might look like a horrible mess and likely isn't easy to commit to memory. Instead, we'd like to show you a way that might make it easier to remember how to multiply matrices. It also gives a nice representation of why your matrices need to be conformable.

We can leave Matrix A exactly where it is, flip Matrix B^T, and stack it right on top of Matrix A:

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ 7_b & 8_b & 9_b \\ \\ 1_a & 2_a & 3_a \\ 4_a & 5_a & 6_a \end{bmatrix}$$

Now, we can let those rearranged columns from Matrix B^T 'fall down' through the rows of Matrix A:

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ \\ 1_a * 7_b & 2_a * 8_b & 3_a * 9_b \\ 4_a & 5_a & 6_a \end{bmatrix} = \underset{\text{Matrix C}}{\begin{bmatrix} 50 & . \\ . & . \end{bmatrix}}$$

Adding those products together gives us 50 for C_{11}.

Let's move that row down to the next row in the Matrix A, multiply, and sum the products.

$$\begin{bmatrix} 9_b & 8_b & 7_b \\ \\ 1_a & 2_a & 3_a \\ 4_a * 7_b & 5_a * 8_b & 6_a * 9_b \end{bmatrix} = \underset{\text{Matrix C}}{\begin{bmatrix} 50 & . \\ 122 & . \end{bmatrix}}$$

We have 122 for C_{21}. That first column from Matrix B^T won't be used any more, but now we need to move the second column through Matrix A.

$$\begin{bmatrix} 1_a * 9_b & 2_a * 8_b & 3_a * 7_b \\ 4_a & 5_a & 6_a \end{bmatrix} = \begin{matrix} \text{Matrix C} \\ \begin{bmatrix} 50 & 46 \\ 122 & . \end{bmatrix} \end{matrix}$$

That gives us 46 for C_{12}.

And finally:

$$\begin{bmatrix} 1_a & 2_a & 3_a \\ 4_a * 9_b & 5_a * 8_b & 6_a * 7_b \end{bmatrix} = \begin{matrix} \text{Matrix C} \\ \begin{bmatrix} 50 & 46 \\ 122 & 118 \end{bmatrix} \end{matrix}$$

We have 118 for C_{22}.

Now that you know how these work, you can see how easy it is to handle these tasks in R and Python.

R

In R, we need to use a fancy operator: `%*%`. This is just R's matrix multiplication operator. We will also use the transpose function: `t`.

```
matrix_A %*% t(matrix_B)
```

```
     [,1] [,2]
[1,]   50   46
[2,]  122  118
```

Python

In Python, we can just use the regular multiplication operator and the transpose method:

```
matrix_A @ matrix_B.T
```

```
array([[ 50,  46],
       [122, 118]])
```

You can see that whether we do this by hand, R, or Python, we come up with the same answer! While these small matrices can definitely be done by hand, we will always trust the computer to handle larger matrices. The main thing is to understand the mechanics behind the operation.

> **Element-wise Multiplication**
>
> Matrix multiplication is not the same as **element-wise** multiplication. Element-wise multiplication is when you multiply each element in one matrix by the corresponding element in another matrix. This is done with the $*$ operator in R and Python. The matrices must have identical dimensions for this. As with addition and subtraction, if one matrix is a scalar, the operation is automatically applied to every element in the matrix.

B.5 Division

Though addition, subtraction, and multiplication are all pretty straightforward, matrix division is not. In fact, there really isn't such a thing as matrix division, we just use matrix multiplication in a particular way. This is similar to how we can divide two numbers, for example, a/b, but we can also multiply by the reciprocal, $a * (1/b)$. In matrix terms, this would look something like:

$$AB^{-1}$$

While that may also seem straightforward on the surface, **matrix inversion** is not. The basic idea is that we are looking for a matrix that, when multiplied by the original matrix like B, gives us the identity matrix. The **identity matrix** is a matrix that has 1s along the diagonal and 0s everywhere else.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Another caveat is that not all matrices have inverses. If the **determinant** of a matrix is 0, then it does not have an inverse. Technically, only square matrices can have inverses, but not all square matrices have inverses. We can, however, get a **pseudo-inverse** for nonsquare matrices.

R

```
matrix_B_inv = MASS::ginv(matrix_B)
round(matrix_B %*% matrix_B_inv)
```

```
     [,1] [,2]
[1,]    1    0
```

```
[2,]    0    1
```

Python

```
matrix_B_inv = np.linalg.pinv(matrix_B)
(matrix_B @ matrix_B_inv).round()
```

```
array([[ 1., -0.],
       [-0.,  1.]])
```

More to the point, when would we do this? In the world of modeling, we might use matrix inversion to solve a system of equations. For example, this can be implemented in linear regression, where we are trying to find the coefficients that minimize the error in our model. That problem has an analytical solution that involves matrix inversion.

$$\beta = (X^T X)^{-1} X^T y$$

Let's see this for ourselves. We will create a simple linear regression model and solve for the coefficients using matrix inversion.

R

```
set.seed(123)
x = rnorm(100)
y = 2*x + rnorm(100)
X = cbind(1, x)

beta = MASS::ginv(t(X) %*% X) %*% t(X) %*% y

tibble(
  ours = beta[,1],
  standard = coef(lm(y ~ x))
)
```

```
# A tibble: 2 x 2
    ours standard
   <dbl>    <dbl>
1 -0.103   -0.103
2  1.95     1.95
```

Python

```
import statsmodels.api as sm
import pandas as pd

np.random.seed(123)
x = np.random.normal(size = 100)
y = 2*x + np.random.normal(size = 100)
X = np.c_[np.ones(100), x]

beta = np.linalg.pinv(X.T @ X) @ X.T @ y
beta
```

```
array([-0.01908575,  1.98340745])
```

```
model_sm = sm.OLS(y, X)
results_sm = model_sm.fit()
coefficients_sm = results_sm.params

pd.DataFrame({
    'ours': beta,
    'standard': coefficients_sm
})
```

```
       ours  standard
0 -0.019086 -0.019086
1  1.983407  1.983407
```

B.6 Summary

While matrix operations are not something we explicitly do everyday data science, it is always lurking behind the scenes. Having a grasp of the underlying model mechanics helps demystify the modeling process, and can greatly expand a data scientist's abilities when you have to dive into matrix operations for model building. Whether linear regression or deep learning, matrix operations are at the core of almost every model you come across.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

C

Dataset Descriptions

All data can be found in the book's repo. Depending on when you access it, there may be more or less data available. We'll try to clean it up to make it more clear eventually, but it's easiest to use the code in the demonstrations to download the data directly.

C.1 Movie Reviews

The movie reviews dataset was a fun way to use an LLM to create movie titles and reviews in a specific way, as well as other features. With features in hand, we then generated a rating outcome with specific feature-target relationships. It has 1000 rows and the following columns:

- `title`: The title of the movie
- `review_year`: The year the review was written
- `age`: The age of the reviewer
- `children_in_home`: The number of children in the reviewer's home
- `education`: The educational level of the reviewer (Post-Graduate, Completed College, Completed High School)
- `gender`: The gender of the reviewer (Male or Female)
- `work_status`: The work status of the reviewer (Employed, Retired, Unemployed, Student)
- `genre`: The genre of the movie
- `release_year`: The year the movie was released
- `length_minutes`: The length of the movie in minutes
- `season`: The season the movie was released (e.g., Fall, Winter)
- `total_reviews`: The total number of reviews for the movie
- `rating`: The rating of the movie
- `review_text`: The text of the review
- `word_count`: The number of words in the review
- `review_year_0`: The review year starting from 0
- `release_year_0`: The release year starting from 0
- `*_sc`: Scaled (standardized) versions of age, length_minutes, total_reviews, and word_count

- `rating_good`: A binary version of rating, where 1 is a good rating (>= 3) and 0 is a bad rating (<3)

Link:

- https://tinyurl.com/moviereviewsdata

Repo File:

- `data/movie_reviews.csv`

Table C.1: Movie Reviews Dataset (String)

variable	empty	n_unique
title	0.0	100.0
education	0.0	3.0
gender	0.0	2.0
work_status	0.0	4.0
genre	0.0	8.0
season	0.0	4.0
review_text	0.0	442.0

Table C.2: Movie Reviews Dataset (Numeric)

variable	mean	sd	min	med	max
review_year	2015.8	5.1	2000.0	2017.0	2022.0
age	46.9	18.3	18.0	47.0	80.0
children_in_home	0.4	0.7	0.0	0.0	3.0
release_year	2008.1	9.6	1983.0	2010.0	2020.0
length_minutes	121.0	11.5	98.0	120.0	147.0
total_reviews	4921.7	2837.9	374.0	4464.0	9926.0
rating	3.1	0.6	1.0	3.1	5.0
word_count	10.3	5.1	2.0	9.0	32.0
rating_good	0.6	0.5	0.0	1.0	1.0

C.2 World Happiness Report

The World Happiness Report is a survey of the state of global happiness that ranks countries by how 'happy' their citizens perceive themselves to be. You can also find additional details in their supplemental documentation. Our 2018 data is from what was originally reported at that time (figure 2.2 in the corresponding report), and it also contains a life ladder score from the most recent survey, which is similar and very highly correlated.

The datasets contain the following columns:

- `country`: The country name
- `year`: The year of the survey

- `life_ladder`: The happiness score
- `log_gdp_per_capita`: The log of GDP per capita
- `social_support`: The social support score
- `healthy_life_expectancy_at_birth`: The healthy life expectancy at birth
- `freedom_to_make_life_choices`: The freedom to make life choices score
- `generosity`: The generosity score
- `perceptions_of_corruption`: The perceptions of corruption score
- `positive_affect`: The positive affect score
- `negative_affect`: The negative affect score
- `confidence_in_national_government`: The confidence in national government score
- `happiness_score`: The happiness score
- `dystopia_residual`: The dystopia residual score (difference from a 'least happy' country)

In addition, there are standardized/scaled versions of the features, which are suffixed with `_sc`.

Links:

- All years: https://tinyurl.com/worldhappinessallyears
- 2018: https://tinyurl.com/worldhappiness2018

Repo Files:

- `data/world_happiness_all_years.csv`
- `data/world_happiness_2018.csv`

Table C.3: World Happiness Report Dataset (All Years)

variable	n_missing	mean	sd	min	med	max
year	0.0	2014.2	4.7	2005.0	2014.0	2022.0
happiness_score	0.0	5.5	1.1	1.3	5.4	8.0
log_gdp_per_capita	20.0	9.4	1.2	5.5	9.5	11.7
social_support	13.0	0.8	0.1	0.2	0.8	1.0
healthy_life_expectancy_at_birth	54.0	63.3	6.9	6.7	65.0	74.5
freedom_to_make_life_choices	33.0	0.7	0.1	0.3	0.8	1.0
generosity	73.0	0.0	0.2	−0.3	0.0	0.7
perceptions_of_corruption	116.0	0.7	0.2	0.0	0.8	1.0
positive_affect	24.0	0.7	0.1	0.2	0.7	0.9
negative_affect	16.0	0.3	0.1	0.1	0.3	0.7

Table C.4: World Happiness Report Dataset (2018)

variable	mean	sd	min	med	max
life_ladder	5.6	1.1	2.7	5.5	7.9
log_gdp_per_capita	9.3	1.2	6.6	9.5	11.5
social_support	0.8	0.1	0.5	0.8	1.0
healthy_life_expectancy_at_birth	64.7	6.8	48.2	66.7	75.0
freedom_to_make_life_choices	0.8	0.1	0.4	0.8	1.0
generosity	0.0	0.2	−0.3	0.0	0.5
perceptions_of_corruption	0.7	0.2	0.2	0.8	1.0
positive_affect	0.7	0.1	0.4	0.7	0.9
negative_affect	0.3	0.1	0.2	0.3	0.5
confidence_in_national_government	0.5	0.2	0.1	0.5	1.0
happiness_score	5.4	1.1	3.3	5.4	7.6
dystopia_residual	2.0	0.5	0.3	1.9	2.9

C.3 Heart Disease UCI

This classic dataset comes from the UCI ML repository. We took a version from Kaggle, and features and target were renamed to be more intelligible. Here is a brief description from UCI:

> This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. In particular, the Cleveland database is the only one that has been used by ML researchers to date. The "goal" field refers to the presence of heart disease in the patient. It is integer valued from 0 (no presence) to 4. Experiments with the Cleveland database have concentrated on simply attempting to distinguish presence (values 1,2,3,4) from absence (value 0).

- `age`: Age in years
- `male`: 'yes' or 'no'
- `chest_pain_type`: 'typical', 'atypical', 'non-anginal', 'asymptomatic'
- `resting_bp`: Resting blood pressure (mm-Hg)
- `cholesterol`: Serum cholesterol (mg/dl)
- `fasting_blood_sugar`: '> 120 mg/dl' or '<= 120 mg/dl'
- `resting_ecg`: 'normal', 'left ventricular hypertrophy', 'ST-T wave abnormality'
- `max_heart_rate`: Maximum heart rate achieved
- `exercise_induced_angina`: 'yes' or 'no'
- `st_depression`: ST depression induced by exercise relative to rest
- `slope`: 'upsloping', 'flat', 'downsloping'
- `num_major_vessels`: Number of major vessels (0-3) colored by fluoroscopy
- `thalassemia`: 'normal', 'fixed defect', 'reversible defect'
- `heart_disease`: 'yes' or 'no'

Links:

- Processed: https://tinyurl.com/heartdiseaseprocessed
- Numeric features only: https://tinyurl.com/heartdiseaseprocessednumeric

Repo Files:

- `data/heart_disease_processed.csv`
- `data/heart_disease_processed_numeric_sc.csv`

Table C.5: Heart Disease UCI Dataset (String)

variable	empty	n_unique
chest_pain_type	0.0	4.0
fasting_blood_sugar	0.0	2.0
resting_ecg	0.0	3.0
exercise_induced_angina	0.0	2.0
slope	0.0	3.0
thalassemia	0.0	3.0
heart_disease	0.0	2.0

Table C.6: Heart Disease UCI Dataset (Numeric)

variable	mean	sd	min	med	max
age	54.5	9.0	29.0	56.0	77.0
male	0.7	0.5	0.0	1.0	1.0
resting_bp	131.7	17.8	94.0	130.0	200.0
cholesterol	247.4	52.0	126.0	243.0	564.0
max_heart_rate	149.6	22.9	71.0	153.0	202.0
st_depression	1.1	1.2	0.0	0.8	6.2
num_major_vessels	0.7	0.9	0.0	0.0	3.0

C.4 Fish

This is a very simple dataset with a count target variable. It's also good if you want to try your hand at zero-inflated models. The background is that state wildlife biologists want to model how many fish are being caught by fishermen at a state park.

- `nofish`: We've never seen this explained. Originally 0 and 1, 0 is equivalent to livebait equals 'yes', so it may be whether the primary motivation of the camping trip is for fishing or not.
- `livebait`: Whether or not live bait was used
- `camper`: Whether or not they brought a camper
- `persons`: How many total persons on the trip
- `child`: How many children present

- `count`: Number of fish caught

Link:

- https://tinyurl.com/fishcountdata

Repo File:

- `data/fish.csv`

Table C.7: Fish Dataset (String)

variable	empty	n_unique
nofish	0.0	2.0
livebait	0.0	2.0
camper	0.0	2.0

Table C.8: Fish Dataset (Numeric)

variable	mean	sd	min	med	max
persons	2.5	1.1	1.0	2.0	4.0
child	0.7	0.9	0.0	0.0	3.0
count	3.3	11.6	0.0	0.0	149.0

References

These references tend to be more functional than academic, and hopefully will be more practically useful to you as well. If you prefer additional academic resources, you'll find some of those as well, but you can also look at the references within many of these for deeper or more formal dives, or just search Google Scholar for any of the topics covered.

3Blue1Brown. 2024. "How Large Language Models Work, a Visual Intro to Transformers | Chapter 5, Deep Learning." https://www.youtube.com/watch?v=wjZofJX0v4M.

Albon, Chris. 2024. "Machine Learning Notes." https://chrisalbon.com/Home.

Amazon. 2024. "What Is Data Augmentation? - Data Augmentation Techniques Explained - AWS." *Amazon Web Services, Inc.* https://aws.amazon.com/what-is/data-augmentation/.

Angelopoulos, Anastasios N., and Stephen Bates. 2022. "A Gentle Introduction to Conformal Prediction and Distribution-Free Uncertainty Quantification." arXiv. https://doi.org/10.48550/arXiv.2107.07511.

Barrett, Malcolm, Lucy D'Agostino McGowan, and Travis Gerke. 2024. *Causal Inference in R.* https://www.r-causal.org/.

Bischl, Bernd, Raphael Sonabend, Lars Kotthoff, and Michel Lang, eds. 2024. *Applied Machine Learning Using Mlr3 in R.* https://mlr3book.mlr-org.com/.

Boykis, Vicki. 2023. "What Are Embeddings?" http://vickiboykis.com/what_are_embeddings/index.html.

Brownlee, Jason. 2019. "A Gentle Introduction to Imbalanced Classification." *MachineLearningMastery.com*. https://machinelearningmastery.com/what-is-imbalanced-classification/.

Brownlee, Jason. 2021. "Gradient Descent With AdaGrad From Scratch." *MachineLearningMastery.com*. https://machinelearningmastery.com/gradient-descent-with-adagrad-from-scratch/.

Bürkner, Paul-Christian, and Matti Vuorre. 2019. "Ordinal Regression Models in Psychology: A Tutorial." *Advances in Methods and Practices in Psychological Science* 2 (1): 77–101. https://doi.org/10.1177/2515245918823199.

Cawley, Gavin C., and Nicola L. C. Talbot. 2010. "On Over-Fitting in Model Selection and Subsequent Selection Bias in Performance Evaluation." *The Journal of Machine Learning Research* 11 (August):2079–2107.

Chernozhukov, Victor, Christian Hansen, Nathan Kallus, Martin Spindler, and Vasilis Syrgkanis. 2024. "Applied Causal Inference Powered by ML and AI." arXiv. http://arxiv.org/abs/2403.02467.

Clark, Michael. 2018a. *Graphical & Latent Variable Modeling*. https://m-clark.github.io/sem/.

Clark, Michael. 2018b. "Thinking about Latent Variables." https://m-clark.github.io/docs/FA_notes.html.

Clark, Michael. 2021a. *Model Estimation by Example*. https://m-clark.github.io/models-by-example/.

Clark, Michael. 2021b. "This Is Definitely Not All You Need," July. https://m-clark.github.io/posts/2021-07-15-dl-for-tabular/.

Clark, Michael. 2022a. "Deep Learning for Tabular Data," May. https://m-clark.github.io/posts/2022-04-01-more-dl-for-tabular/.

Clark, Michael. 2022b. *Generalized Additive Models*. https://m-clark.github.io/generalized-additive-models/.

Clark, Michael. 2023. *Mixed Models with R*. https://m-clark.github.io/mixed-models-with-R/.

Clark, Michael. 2025. "Imbalanced Outcomes." https://m-clark.github.io/posts/2025-04-07-class-imbalance/.

Cohen, Jacob. 2009. *Statistical Power Analysis for the Behavioral Sciences*. 2. ed., reprint. New York, NY: Psychology Press.

Cunningham, Scott. 2023. *Causal Inference The Mixtape*. https://mixtape.scunning.com/.

DataBricks. 2019. "What Is AdaGrad?" *Databricks*. https://www.databricks.com/glossary/adagrad.

Davison, A. C., and D. V. Hinkley. 1997. *Bootstrap Methods and Their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge: Cambridge University Press. https://doi.org/10.1017/CBO9780511802843.

Dobson, Annette J., and Adrian G. Barnett. 2018. *An Introduction to Generalized Linear Models*. 4th ed. New York: Chapman & Hall/CRC Press. https://doi.org/10.1201/9781315182780.

Dunn, Peter K., and Gordon K. Smyth. 2018. *Generalized Linear Models With Examples in R*. Springer.

Efron, Bradley, and R. J. Tibshirani. 1994. *An Introduction to the Bootstrap*. New York: Chapman & Hall/CRC Press. https://doi.org/10.1201/9780429246593.

Elor, Yotam, and Hadar Averbuch-Elor. 2022. "To SMOTE, or Not to SMOTE?" arXiv. https://doi.org/10.48550/arXiv.2201.08528.

Facure Alves, Matheus. 2022. "Causal Inference for The Brave and True — Causal Inference for the Brave and True." https://matheusfacure.github.io/python-causality-handbook/landing-page.html.

Fahrmeir, Ludwig, Thomas Kneib, Stefan Lang, and Brian D. Marx. 2021. *Regression: Models, Methods and Applications*. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-662-63882-8.

Faraway, Julian. 2014. *Linear Models with R*. Routledge & CRC Press. https://www.routledge.com/Linear-Models-with-R/Faraway/p/book/9781439887332.

Faraway, Julian J. 2016. *Extending the Linear Model with R: Generalized Linear, Mixed Effects and Nonparametric Regression Models, Second Edition*. 2nd ed. New York: Chapman & Hall/CRC Press. https://doi.org/10.1201/9781315382722.

Fox, John. 2015. *Applied Regression Analysis and Generalized Linear Models*. SAGE Publications.

Gelman, Andrew. 2013. "What Are the Key Assumptions of Linear Regression? | Statistical Modeling, Causal Inference, and Social Science." https://statmodeling.stat.columbia.edu/2013/08/04/19470/.

Gelman, Andrew, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. 2013. *Bayesian Data Analysis, Third Edition*. CRC Press.

Gelman, Andrew, and Jennifer Hill. 2006. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Gelman, Andrew, Jennifer Hill, Ben Goodrich, Jonah Gabry, Daniel Simpson, and Aki Vehtari. 2025. "Advanced Regression and Multilevel Models." http://www.stat.columbia.edu/~gelman/armm/.

Gelman, Andrew, Jennifer Hill, and Aki Vehtari. 2020. *Regression and Other Stories*. 1st ed. Cambridge University Press. https://doi.org/10.1017/9781139161879.

Google. 2023. "Machine Learning | Google for Developers." https://developers.google.com/machine-learning.

Google. 2024. "MLOps: Continuous Delivery and Automation Pipelines in Machine Learning | Cloud Architecture Center." *Google Cloud*. https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

Gorishniy, Yury, Ivan Rubachev, Nikolay Kartashev, Daniil Shlenskii, Akim Kotelnikov, and Artem Babenko. 2023. "TabR: Tabular Deep Learning Meets Nearest Neighbors in 2023." arXiv. https://doi.org/10.48550/arXiv.2307.14338.

Greene, William. 2017. *Econometric Analysis - 8th Edition*. https://pages.stern.nyu.edu/~wgreene/Text/econometricanalysis.htm.

Gruber, Cornelia, Patrick Oliver Schenk, Malte Schierholz, Frauke Kreuter, and Göran Kauermann. 2023. "Sources of Uncertainty in Machine Learning – A Statisticians' View." arXiv. https://doi.org/10.48550/arXiv.2305.16703.

Hardin, James W., and Joseph M. Hilbe. 2018. *Generalized Linear Models and Extensions*. Stata Press.

Harrell, Frank E. 2015. *Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis*. 2nd ed. Springer Series in Statistics. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-19425-7.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2017. *Elements of Statistical Learning: Data Mining, Inference, and Prediction. 2nd Edition.* https://hastie.su.domains/ElemStatLearn/.

Hernán, Miguel A. 2018. "The C-Word: Scientific Euphemisms Do Not Improve Causal Inference From Observational Data." *American Journal of Public Health* 108 (5): 616–19. https://doi.org/10.2105/AJPH.2018.304337.

Howard, Jeremy. 2024. "Practical Deep Learning for Coders - Practical Deep Learning." *Practical Deep Learning for Coders*. https://course.fast.ai/.

Hyndman, Rob, and George Athanasopoulos. 2021. *Forecasting: Principles and Practice (3rd Ed)*. https://otexts.com/fpp3/.

Ivanova, Anna A, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. 2020. "Comprehension of Computer Code Relies Primarily on Domain-General Executive Brain Regions." Edited by Andrea E Martin, Timothy E Behrens, William Matchin, and Ina Bornkessel-Schlesewsky. *eLife* 9 (December):e58906. https://doi.org/10.7554/eLife.58906.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning*. Vol. 103. Springer Texts in Statistics. New York, NY: Springer New York. https://doi.org/10.1007/978-1-4614-7138-7.

Jiang, Lili. 2020. "A Visual Explanation of Gradient Descent Methods (Momentum, AdaGrad, RMSProp, Adam)." *Medium*. https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c.

Koenker, Roger. 2005. *Quantile Regression*. Vol. 38. Cambridge university press. https://books.google.com/books?hl=en&lr=&id=WjOdAgAAQBAJ&oi=fnd&pg=PT12&dq=koenker+quantile+regression&ots=CQFHSt5o-W&sig=G1TpKPHo-BRdJ8qWcBrIBI2FQAs.

Kuhn, Max, and Kjell Johnson. 2023. *Applied Machine Learning for Tabular Data*. https://aml4td.org/.

Kuhn, Max, and Julia Silge. 2023. *Tidy Modeling with R*. https://www.tmwr.org/.

Künzel, Sören R., Jasjeet S. Sekhon, Peter J. Bickel, and Bin Yu. 2019. "Metalearners for Estimating Heterogeneous Treatment Effects Using Machine Learning." *Proceedings of the National Academy of Sciences* 116 (10): 4156–65. https://doi.org/10.1073/pnas.1804597116.

LeCun, Yann, and Ishan Misra. 2021. "Self-Supervised Learning: The Dark Matter of Intelligence." https://ai.meta.com/blog/self-supervised-learning-the-dark-matter-of-intelligence/.

Leech, Gavin, Juan J. Vazquez, Misha Yagudin, Niclas Kupper, and Laurence Aitchison. 2024. "Questionable Practices in Machine Learning." arXiv. https://doi.org/10.48550/arXiv.2407.12220.

LeNail, Alexander. 2024. "LeNet." https://alexlenail.me/NN-SVG/LeNet.html.

Masis, Serg. 2023. "Interpretable Machine Learning with Python - Second Edition." *Packt*. https://www.packtpub.com/product/interpretable-machine-learning-with-python-second-edition/9781803235424.

McCullagh, P. 2019. *Generalized Linear Models*. 2nd ed. New York: Routledge. https://doi.org/10.1201/9780203753736.

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *The Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1007/BF02478259.

McElreath, Richard. 2020. *Statistical Rethinking*. Routledge & CRC Press. https://www.routledge.com/Statistical-Rethinking-A-Bayesian-Course-with-Examples-in-R-and-STAN/McElreath/p/book/9780367139919.

Molnar, Christoph. 2023. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/.

Molnar, Christoph. 2024. *Introduction To Conformal Prediction With Python*. https://christophmolnar.com/books/conformal-prediction/.

Murphy, Kevin P. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press. https://mitpress.mit.edu/9780262018029/machine-learning/.

Murphy, Kevin P. 2023. *Probabilistic Machine Learning*. MIT Press. https://mitpress.mit.edu/9780262046824/probabilistic-machine-learning/.

Navarro, Danielle. 2018. *Learning Statistics with R*. https://learningstatisticswithr.com.

Neal, Radford M. 1996. "Priors for Infinite Networks." In *Bayesian Learning for Neural Networks*, edited by Radford M. Neal, 29–53. New York, NY: Springer. https://doi.org/10.1007/978-1-4612-0745-0_2.

Nelder, J. A., and R. W. M. Wedderburn. 1972. "Generalized Linear Models." *Royal Statistical Society. Journal. Series A: General* 135 (3): 370–84. https://doi.org/10.2307/2344614.

Niculescu-Mizil, Alexandru, and Rich Caruana. 2005. "Predicting Good Probabilities with Supervised Learning." In *Proceedings of the 22nd International Conference on Machine Learning - ICML '05*, 625–32. Bonn, Germany: ACM Press. https://doi.org/10.1145/1102351.1102430.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12 (85): 2825–30. http://jmlr.org/papers/v12/pedregosa11a.html.

Power, Alethea, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. 2022. "Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets." arXiv. https://doi.org/10.48550/arXiv.2201.02177.

Raschka, Sebastian. 2014. "About Feature Scaling and Normalization." https://sebastianraschka.com/Articles/2014_about_feature_scaling.html.

Raschka, Sebastian. 2022a. "Losses Learned." https://sebastianraschka.com/blog/2022/losses-learned-part1.html.

Raschka, Sebastian. 2022b. *Machine Learning with PyTorch and Scikit-Learn*. https://sebastianraschka.com/books/machine-learning-with-pytorch-and-scikit-learn/.

Raschka, Sebastian. 2023a. *Build a Large Language Model (From Scratch)*. https://www.manning.com/books/build-a-large-language-model-from-scratch.

Raschka, Sebastian. 2023b. *Machine Learning Q and AI*. https://nostarch.com/machine-learning-q-and-ai.

Rasmussen, Carl Edward, and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning*. The MIT Press. https://doi.org/10.7551/mitpress/3206.001.0001.

Roback, Paul, and Julie Legler. 2021. *Beyond Multiple Linear Regression*. https://bookdown.org/roback/bookdown-BeyondMLR/.

Robins, J. M., M. A. Hernán, and B. Brumback. 2000. "Marginal Structural Models and Causal Inference in Epidemiology." *Epidemiology (Cambridge, Mass.)* 11 (5): 550–60. https://doi.org/10.1097/00001648-200009000-00011.

Rocca, Baptiste. 2019. "Handling Imbalanced Datasets in Machine Learning." *Medium*. https://towardsdatascience.com/handling-imbalanced-datasets-in-machine-learning-7a0e84220f28.

Rovine, Michael J, and Douglas R Anderson. 2004. "Peirce and Bowditch." *The American Statistician* 58 (3): 232–36. https://doi.org/10.1198/000313004X964.

Schmidhuber, Juergen. 2022. "Annotated History of Modern AI and Deep Learning." arXiv. https://doi.org/10.48550/arXiv.2212.11279.

Shalizi, Cosma. 2015. "F-Tests, R2, and Other Distractions." https://www.stat.cmu.edu/~cshalizi/mreg/15/.

StatQuest with Josh Starmer. 2019a. "Gradient Descent, Step-by-Step." https://www.youtube.com/watch?v=sDv4f4s2SB8.

StatQuest with Josh Starmer. 2019b. "Stochastic Gradient Descent, Clearly Explained!!!" https://www.youtube.com/watch?v=vMh0zPT0tLI.

StatQuest with Josh Starmer. 2021. "Bootstrapping Main Ideas!!!" https://www.youtube.com/watch?v=Xz0x-8-cgaQ.

UCLA Advanced Research Computing. 2023. "FAQ: What Are Pseudo R-Squareds?" https://stats.oarc.ucla.edu/other/mult-pkg/faq/general/faq-what-are-pseudo-r-squareds/.

VanderWeele, Tyler J. 2012. "Invited Commentary: Structural Equation Models and Epidemiologic Analysis." *American Journal of Epidemiology* 176 (7): 608. https://doi.org/10.1093/aje/kws213.

Vig, Jesse. 2019. "Deconstructing BERT, Part 2: Visualizing the Inner Workings of Attention." *Medium*. https://towardsdatascience.com/deconstructing-bert-part-2-visualizing-the-inner-workings-of-attention-60a16d86b5c1.

Walker, Kyle E. 2023. *Analyzing US Census Data*. https://walker-data.com/census-r.

Weed, Ethan, and Danielle Navarro. 2021. *Learning Statistics with Python — Learning Statistics with Python*. https://ethanweed.github.io/pythonbook/landingpage.html.

Wikipedia. 2023. "Relationships Among Probability Distributions." *Wikipedia*. https://en.wikipedia.org/wiki/Relationships_among_probability_distributions.

Wood, Simon N. 2017. *Generalized Additive Models: An Introduction with R, Second Edition*. 2nd ed. Boca Raton: Chapman & Hall/CRC Press. https://doi.org/10.1201/9781315370279.

Wooldridge, Jeffrey M. 2012. *Introductory Econometrics: A Modern Approach*. 5th edition. Mason, OH: Cengage Learning.

Ye, Han-Jia, Huai-Hong Yin, and De-Chuan Zhan. 2024. "Modern Neighborhood Components Analysis: A Deep Tabular Baseline Two Decades Later." arXiv. https://doi.org/10.48550/arXiv.2407.03257.

Zhang, Aston, Zack Lipton, Mu Li, and Alex Smola. 2023. "Dive into Deep Learning — Dive into Deep Learning 1.0.3 Documentation." https://d2l.ai/index.html.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Index