

Statistics for Data Scientists and Analysts

Statistical approach to data-driven decision
making using Python

Dipendra Pant

Suresh Kumar Mukhiya



Statistics for Data Scientists — and — Analysts

Statistical approach to data-driven decision
making using Python



Dipendra Pant

Suresh Kumar Mukhiya

bpb

Statistics for Data Scientists and Analysts

*Statistical approach to data-
driven decision
making using Python*

Dipendra Pant
Suresh Kumar Mukhiya



www.bpbonline.com

First Edition 2025

Copyright © BPB Publications, India

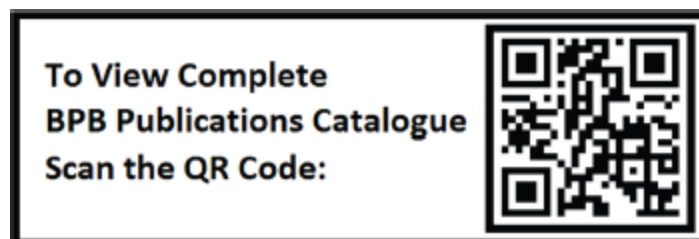
ISBN: 978-93-65897-128

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

*My dad **Mahadev Pant** and mom **Nanda Pant***

My family members and my PhD Supervisor

- Dipendra Pant

My wife and children

- Suresh Kumar Mukhiya

About the Authors

- **Dipendra Pant** is a Ph.D. candidate in Computer Science at the Norwegian University of Science and Technology (NTNU), Norway's leading technical university. He holds Bachelor's and Master's degrees in Computer Engineering from Nepal, where he received the Chancellor's Gold Medal from Kathmandu University for top Master's grades. Before relocating to Norway, Dipendra gained experience in both academia and industry in Nepal and has published multiple high-quality research articles.
- **Suresh Kumar Mukhiya** is a Senior Software Engineer at Tryg Frorsikring Norge in Norway. He holds a Ph.D. in Computer Science from Høgskulen på Vestlandet HVL, Norway. He has extensive knowledge and experience in academia and the software industry, and have authored multiple books and high quality research articles.

About the Reviewer

- ❖ **Dushyant Sengar** is a senior consulting leader in the data science, AI, and financial services domain. His areas of expertise include credit risk modeling, customer and loyalty analytics, model risk management (MRM), ModelOps-driven product development, analytics strategies, and operations. He has managed analytics delivery and sales in Retail, Loyalty, and Banking domains at leading Analytics consulting firms globally where he was involved in practice development, delivery, training, and team building.

Sengar has authored/co-authored 10+ books, peer-reviewed scientific publications, and media articles in industry publications and has presented as an invited speaker and participant at several national and international conferences.

He has strong hands-on experience in data science (methods, strategies, and best practices) as well as in cross-functional team leadership, product strategy, people, program, and budget management. He is an active reader and passionate about helping organizations and individuals realize their full potential with AI.

Acknowledgements

We would like to express our sincere gratitude to everyone who contributed to the completion of this book.

First and foremost, we extend our heartfelt appreciation to our family for their unwavering support and encouragement throughout this journey. Their love has been a constant source of motivation.

We are especially grateful to Laxmi Bhatta and Øystein Nytrø for their invaluable support and motivation during the writing process.

We thank BPB Publications for arranging the reviewers, editors, and technical experts.

Last but not least, we want to express our gratitude to the readers who have shown interest in our work. Your support and encouragement are deeply appreciated.

Thank you to everyone who has played a part in making this book a reality.

Preface

In an era where data is the new oil, the ability to extract meaningful insights from vast amounts of information has become an essential skill across various industries. Whether you are a seasoned data scientist, a statistician, a researcher, or someone beginning their journey in the world of data, understanding the principles of statistics and how to apply them using powerful tools like Python is crucial.

This book was born out of our collective experience in academia and industry, where we recognized a significant gap between theoretical statistical concepts and their practical application using modern programming languages. We noticed that while there are numerous resources available on either statistics or Python programming, few integrate both in a hands-on, accessible manner tailored for data analysis and statistical modeling.

"Statistics for Data Scientists and Analysts" is our attempt to bridge this gap. Our goal is to provide a comprehensive guide that not only explains statistical concepts but also demonstrates how to implement them using Python's rich ecosystem of libraries such as NumPy, Pandas, Matplotlib, Seaborn, SciPy, and scikit-learn. We believe that the best way to learn is by doing, so we've included numerous examples, code snippets, exercises, and real-world datasets to help you apply what you've learned immediately.

Throughout this book, we cover a wide range of topics—from the fundamentals of descriptive and inferential statistics to advanced subjects like time series analysis,

survival analysis, and machine learning techniques. We've also dedicated a chapter to the emerging field of prompt engineering for data science, acknowledging the growing importance of AI and language models in data analysis.

We wrote this book with a diverse audience in mind. Whether you have a background in Python programming or are new to the language, we've structured the content to be accessible without sacrificing depth. Basic knowledge of Python and statistics will be helpful but is not mandatory. Our aim is to equip you with the skills to explore, analyze, and visualize data effectively, ultimately empowering you to make informed decisions based on solid statistical reasoning.

As you embark on this journey, we encourage you to engage actively with the material. Try out the code examples, tackle the exercises, and apply the concepts to your own datasets. Statistics is not just about numbers; it's a lens through which we can understand the world better.

We are excited to share this knowledge with you and hope that this book becomes a valuable resource in your professional toolkit.

Chapter 1: Foundations of Data Analysis and Python -

In this chapter, you will learn the fundamentals of statistics and data, including their definitions, importance, and various types and applications. You will explore basic data collection and manipulation techniques. Additionally, you will learn how to work with data using Python, leveraging its powerful tools and libraries for data analysis.

Chapter 2: Exploratory Data Analysis - This chapter introduces Exploratory Data Analysis (EDA), the process of examining and summarizing datasets using techniques like descriptive statistics, graphical displays, and clustering methods. EDA helps uncover key features, patterns, outliers,

and relationships in data, generating hypotheses for further analysis. You'll learn how to perform EDA in Python using libraries such as pandas, NumPy, SciPy, and scikit-learn. The chapter covers data transformation, normalization, standardization, binning, grouping, handling missing data and outliers, and various data visualization techniques.

Chapter 3: Frequency Distribution, Central Tendency, Variability - Here, you will learn how to describe and summarize data using descriptive statistical techniques such as frequency distributions, measures of central tendency (mean, median, mode), and measures of variability (range, variance, standard deviation). You will use Python libraries like pandas, NumPy, SciPy, and Matplotlib to compute and visualize these statistics, gaining insights into how data values are distributed and how they vary.

Chapter 4: Unraveling Statistical Relationships - This chapter focuses on measuring and examining relationships between variables using covariance and correlation. You will learn how these statistical measures assess how two variables vary together or independently. The chapter also covers identifying and handling outliers—data points that significantly differ from the rest, which can impact the validity of analyses. Finally, you will explore probability distributions, mathematical functions that model data distribution and the likelihood of various outcomes.

Chapter 5: Estimation and Confidence Intervals - In this chapter, you will delve into estimation techniques, focusing on constructing confidence intervals for various parameters and data types. Confidence intervals provide a range within which the true population parameter is likely to lie with a certain level of confidence. You will learn how to calculate margin of error and determine sample sizes to assess the accuracy and precision of your estimates.

Chapter 6: Hypothesis and Significance Testing - This chapter introduces hypothesis testing and significance tests using Python. You will learn how to perform and interpret hypothesis tests for different parameters and data types, assessing the reliability and validity of results using p-values, significance levels, and statistical power. The chapter covers common tests such as t-tests, chi-square tests, and ANOVA, equipping you with the skills to make informed decisions based on statistical evidence.

Chapter 7: Statistical Machine Learning - Here, you will learn how to implement various supervised learning techniques for regression and classification tasks, as well as unsupervised learning techniques for clustering and dimensionality reduction. Starting with the basics—training and testing data, loss functions, evaluation metrics, and cross-validation—you will implement models like linear regression, logistic regression, decision trees, random forests, and support vector machines. Using scikit-learn library you will build, train, and evaluate these models on real-world datasets.

Chapter 8: Unsupervised Machine Learning - This chapter introduces unsupervised machine learning techniques that uncover hidden patterns in unlabeled data. We begin with clustering methods—including K-means, K-prototype, hierarchical clustering, and Gaussian mixture models—that group similar data points together. Next, we delve into dimensionality reduction techniques like Principal Component Analysis and Singular Value Decomposition, which simplify complex datasets while retaining essential information. Finally, we discuss model selection and evaluation strategies tailored for unsupervised learning, equipping you with the tools to assess and refine your models effectively.

Chapter 9: Linear Algebra, Nonparametric Statistics, and Time Series Analysis - In this chapter, you will explore advanced topics including linear algebra operations, nonparametric statistical methods that don't assume a specific data distribution, and time series analysis concepts for dealing with time-to-event data.

Chapter 10: Generative AI and Prompt Engineering - This chapter introduces Generative AI and the concept of prompt engineering in the context of statistics and data science. You will learn how to write accurate and efficient prompts for AI models, understand the limitations and challenges associated with Generative AI, and explore tools like the GPT-4 API. This knowledge will help you effectively utilize Generative AI in data science tasks while avoiding common pitfalls.

Chapter 11: Real World Statistical Applications - In the final chapter, you will apply the concepts learned throughout the book to real-world data science projects. Covering the entire lifecycle from data cleaning and preprocessing to modeling and interpretation, you will work on projects involving statistical analysis of banking data and health data. This hands-on experience will help you implement data science solutions to practical problems, illustrating workflows and best practices in the field.

Code Bundle and Coloured Images

Please follow the link to download the **Code Bundle** and the **Coloured Images** of the book:

<https://rebrand.ly/68f7c9>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Statistics-for-Data-Scientists-and-Analysts>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Foundations of Data Analysis and Python

Introduction

Structure

Objectives

Environment setup

Software installation

Launch application

Basic overview of technology

Python

pandas

NumPy

Sklearn

Matplotlib

Statistics, data and its importance

Types of data

Qualitative data

Quantitative data

Level of measurement

Nominal data

Ordinal data

Discrete data

Continuous data

Interval data

- Ratio data*
- Distinguishing qualitative and quantitative data*
- Univariate, bivariate, and multivariate data
 - Univariate data and univariate analysis*
 - Bivariate data*
 - Multivariate data*
- Data sources, methods, populations, and samples
 - Data source*
 - Collection methods*
 - Population and sample*
- Data preparation tasks
 - Data quality*
 - Cleaning*
 - Missing values*
 - Imputation*
 - Duplicates*
 - Outliers*
- Wrangling and manipulation
- Conclusion

2. Exploratory Data Analysis

- Introduction
- Structure
- Objectives
- Exploratory data analysis and its importance
- Data aggregation
 - Mean*
 - Median*
 - Mode*
 - Variance*

- Standard deviation*
- Quantiles*
- Data normalization, standardization, and transformation
 - Data normalization*
 - Normalization of NumPy array*
 - Normalization of pandas data frame*
 - Data standardization*
 - Standardization of NumPy array*
 - Standardization of data frame*
 - Data transformation*
- Data binning, grouping, encoding
 - Data binning*
 - Data grouping*
 - Data encoding*
- Missing data, detecting and treating outliers
- Visualization and plotting of data
 - Line plot*
 - Pie chart*
 - Bar chart*
 - Histogram*
 - Scatter plot*
 - Stacked area plot*
 - Dendrograms*
 - Violin plot*
 - Word cloud*
 - Graph*
- Conclusion

3. Frequency Distribution, Central Tendency, Variability

Introduction

Structure

Objectives

Measure of frequency

Frequency tables and distribution

Relative and cumulative frequency

Measure of central tendency

Measures of variability or dispersion

Measure of association

Covariance and correlation

Chi-square

Cramer's V

Contingency coefficient

Measures of shape

Skewness

Kurtosis

Conclusion

4. Unravelling Statistical Relationships

Introduction

Structure

Objectives

Covariance

Correlation

Outliers and anomalies

Probability

Probability distribution

Uniform distribution

Normal distribution

Binomial distribution

Poisson distribution

Array and matrices

Use of array and matrix

Conclusion

5. Estimation and Confidence Intervals

Introduction

Structure

Objectives

Point and interval estimate

Standard error and margin of error

Confidence intervals

Types and interpretation

Confidence interval and t-test relation

Confidence interval and p-value

Confidence interval for mean

Confidence interval for proportion

Confidence interval for differences

Confidence interval estimation for diabetes data

Confidence interval estimate in text

Conclusion

6. Hypothesis and Significance Testing

Introduction

Structure

Objectives

Hypothesis testing

Steps of hypothesis testing

Types of hypothesis testing

Significance testing

Steps of significance testing

Types of significance testing

Role of p-value and significance level

Statistical tests

Z-test

T-test

Chi-square test

One-way ANOVA

Two-way ANOVA

Hypothesis and significance testing in diabetes dataset

Sampling techniques and sampling distributions

Conclusion

7. Statistical Machine Learning

Introduction

Structure

Objectives

Machine learning

Understanding machine learning

Role of data, algorithm, statistics

Inference, prediction and fitting models to data

Supervised learning

Fitting models to independent data

Linear regression

Logistic regression

Fitting models to dependent data

Linear mixed effect model

Decision tree

- Random forest*
- Support vector machine*
- K-nearest neighbor*
- Model selection and evaluation
 - Evaluation metrics and model selection for supervised*
- Semi-supervised and self-supervised learnings
- Semi-supervised techniques
- Self-supervised techniques
- Conclusion

8. Unsupervised Machine Learning

- Introduction
- Structure
- Objectives
- Unsupervised learning
 - K-means*
 - K-prototype*
 - Hierarchical clustering*
 - Gaussian mixture models*
 - Principal component analysis*
 - Singular value decomposition*
 - DBSCAN*
 - t-distributed stochastic neighbor embedding*
 - Apriori*
 - Eclat*
 - FP-Growth*
- Model selection and evaluation
 - Evaluation metrics and model selection for unsupervised*

Conclusion

9. Linear Algebra, Nonparametric Statistics, and Time Series Analysis

Introduction

Structure

Objectives

Linear algebra

Nonparametric statistics

Rank-based tests

Goodness-of-fit tests

Independence tests

Kruskal-Wallis test

Bootstrapping

Survival analysis

Time series analysis

Conclusion

10. Generative AI and Prompt Engineering

Introduction

Structure

Objectives

Generative AI

Understanding generative AI

Large language model

Prompt engineering and types of prompts

Open-ended prompts versus specific prompts

Zero-shot, one-shot, and few-shot learning

Zero-shot

One-shot
Few-shot
Chain-of-thought
Self-consistency
Generated knowledge
Retrieval augmented generation
Using LLM and generative AI models
Setting up GPT-4 in Python using the OpenAI API
Best practices for building effective prompts
Industry-specific use cases
Conclusion

11. Real World Statistical Applications

Introduction

Structure

Objectives

Project I: Implementing data science and statistical analysis on banking data

Part 1: Exploratory data analysis

Part 2: Statistical testing

Part 3: Analyze the credit card risk

Part 4: Predictive modeling

Part 5: Use the predictive model above Part 4. Feed it user input and see predictions

Project II: Implementing data science and statistical analysis on health data

Part 1: Exploratory data analysis

Part 2: Statistical analysis

Part 3: Inferential statistics

Part 4: Statistical machine learning

Conclusion

Index

CHAPTER 1

Foundations of Data Analysis and Python

Introduction

In today's data-rich landscape, data is much more than a collection of numbers or facts, it's a powerful resource that can influence decision-making, policy formation, product development, and scientific discovery. To turn these raw inputs into meaningful insights, we rely on statistics, the discipline dedicated to collecting, organizing, summarizing, and interpreting data. Statistics not only helps us understand patterns and relationships but also guides us in making evidence-based decisions with confidence. This chapter examines fundamental concepts at the heart of data analysis. We'll explore what data is and why it matters, distinguish between various types of data and their levels of measurement, and consider how data can be categorized as univariate, bivariate, or multivariate. We'll also highlight different data sources, clarify the roles of populations and samples, and introduce crucial data preparation tasks including cleaning, wrangling, and manipulation to ensure data quality and integrity.

For example, consider you have records of customer purchases at an online store everything from product categories and prices to transaction dates and customer demographics. Applying statistical principles and effective data preparation techniques to this information can reveal purchasing patterns, highlight which product lines drive the most revenue, and suggest targeted promotions that improve the shopping experience.

Structure

In this chapter, we will discuss the following topics:

- Environment setup
- Software installation
- Basic overview of technology
- Statistics, data, and its importance
- Types of data
- Levels of measurement
- Univariate, bivariate, and multivariate data
- Data sources, methods, population, and samples
- Data preparation tasks
- Wrangling and manipulation

Objectives

By the end of this chapter, readers will learn the basics of statistics and data, such as, what they are, why they are important, how they vary in type and application, and the basic data collection and manipulation techniques. Moreover, this chapter explains different level of measurements, data analysis techniques, its source, collection methods, their quality and cleaning. You will also learn how to work with data using Python, a powerful and popular programming language that offers many tools and

libraries for data analysis.

Environment setup

To set up the environment and to run the sample code for statistics and data analysis in Python, the three options are as follows:

- Download and install Python from <https://www.python.org/downloads/>. Other packages need to be installed explicitly on top of Python. Then, use any **integrated development environment (IDE)** like visual studio code to execute Python code.
- You can also use Anaconda, a Python distribution designed for large-scale data processing, predictive analytics, and scientific computing. The Anaconda distribution is the easiest way to code in Python. It works on Linux, Windows, and Mac OS X. It can be downloaded from <https://www.anaconda.com/distribution/>.
- You can also use cloud services, which is the easiest of all options but requires internet connectivity to use. Cloud providers like Microsoft Azure Notebooks, GitHub Code Spaces and Google Collaboratory are very popular. Following are a few links:
 - **Microsoft Azure Notebooks:** <https://notebooks.azure.com/>
 - **GitHub Codespaces:** Create a GitHub account from <https://github.com/join> then, once logged in, create a repository from <https://github.com/new>. Once the repository is created, open the repository in the codespace by using the following instructions:
<https://docs.github.com/en/codespaces/developing-in-codespaces/creating-a-codespace-for-a->

repository.

- **Google Collaboratory:** Create a Google account, open <https://colab.research.google.com/>, and create a new notebook.

Azure Notebook GitHub Codespace and Google Collaboratory are cloud-based and easy-to-use platforms. To run and set up an environment locally, install the Anaconda distribution on your machine and follow the software installation instructions.

Software installation

Now, let us look at the steps to install Anaconda to run the sample code and tutorials on the local machine as follows:

1. Download the Anaconda Python distribution from the following link: <https://www.anaconda.com/download>
2. Once the download is complete, run the setup to begin the installation process.
3. Once the Anaconda application has been installed, click **Close** and move to the next step to launch the application.

Check Anaconda installation instructions in the following:

<https://docs.anaconda.com/free/anaconda/install/index.html>

Launch application

Now, let us launch the installed Anaconda navigator and the JupyterLab in it.

Following are the steps:

1. After installing the Anaconda navigator, open any Anaconda navigator and then install and launch JupyterLab.
2. This will start the Jupyter server listening on port 8888.

Usually, a pop-up window comes with a default browser, but you can also start the JupyterLab application on any web browser, Google Chrome preferred, and go to the following URL:

<http://localhost:8888/>

3. A blank notebook is launched in a new window. You can write Python code on it.

4. Select the cell and press run to execute the code.

The environment is now ready to write, run and execute tutorials.

Basic overview of technology

Python, NumPy, pandas, Sklearn, Matplotlib will be used in most of the tutorials. Let us have a look at them in the following section.

Python

To know more about Python and installation you can refer to the following link:

<https://www.python.org/about/gettingstarted/>. Execute Python-version in terminal or command prompt, and if you see the Python version as output, you are good to go, else, install Python. There are different ways to install Python packages on Jupyter Notebook, depending on the package manager you use and the environment you work in, as follows:

- If you use **pip** as your package manager, you can install packages directly from a code cell in your notebook by typing **!pip install <package_name>** and running the cell. Then replace **<package_name>** with the name of the package you want to install.
- If you use **conda** as your package manager, you can

install packages from a JupyterLab cell by typing **!conda install <package_name> --yes** and running the cell. The **--yes** flag is to avoid prompts that asks for confirmation.

- If you want to install a specific version of Python for your notebook, you can use the ipykernel module to create a new kernel with that version. For example, if you have *Python 3.11* and pip installed on your machine, you can type **!pip3.11 install ipykernel** and **!python3.11 -m ipykernel install -user** in two separate code cells and run them. Then, you can select *Python 3.11* as your kernel from the kernel menu.

Further tutorials will be based on the JupyterLab.

pandas

pandas is mainly used for data analysis and manipulation in Python. More can be read at:

<https://pandas.pydata.org/docs/>

Following are the ways to install pandas:

- In Jupyter Notebook, execute **pip install pandas**
- In the conda environment, execute **conda install pandas --yes**

NumPy

NumPy is a Python package for numerical computing, multi-dimensional array, and math computation. More can be read at <https://pandas.pydata.org/docs/>.

Following are the ways to install NumPy:

- In Jupyter Notebook, execute **pip install pandas**
- In the conda environment, execute **conda install pandas -yes**

Sklearn

Sklearn is a Python package that provides tools for machine learning, such as data preprocessing, model selection, classification, regression, clustering, and dimensionality reduction. Sklearn is mainly used for predictive data analysis and building machine learning models. More can be read at <https://scikit-learn.org/0.21/documentation.html>.

Following are the ways to install Sklearn:

- In Jupyter Notebook, execute **pip install scikit-learn**
- In the conda environment, execute **conda install scikit-learn -yes**

Matplotlib

Matplotlib is mainly used to create static, animated, and interactive visualizations (plots, figures, and customized visual style and layout) in Python. More can be read at <https://matplotlib.org/stable/index.html>.

Following are the ways to install Matplotlib:

- In Jupyter notebook, excute **pip install matplotlib**
- In the conda environment, execute **conda install matplotlib --yes**

Statistics, data and its importance

As established, statistics is a disciplined approach that enables us to derive insights from diverse forms of data. By applying statistical principles, we can understand what is happening around us quantitatively, evaluate claims, avoid misleading interpretations, produce trustworthy results, and support data-driven decision-making. Statistics also equips us to make predictions and deepen our understanding of the subjects we study.

Data, in turn, serves as the raw material that fuels statistical analysis. It may take various forms—numbers, words,

images, sounds, or videos—and provides the foundational information needed to extract useful knowledge and generate actionable insights. Through careful examination and interpretation, data leads us toward new discoveries, informed decisions, and credible forecasts.

Ultimately, data and statistics are interdependent. Without data, statistics has no basis for drawing conclusions; without statistics, raw data remains untapped and lacks meaning. When combined, they answer fundamental WH questions—Who, What, When, Where, Why, and How—with clarity and confidence, guiding our understanding and shaping the decisions we make.

Types of data

Data can be in different form and type but generally it can be divided into two types, that is, qualitative and quantitative.

Qualitative data

Qualitative data cannot be measured or counted in numbers. Also known as categorical data, it is descriptive, interpretation-based, subjective, and unstructured. It describes the qualities or characteristics of something. It helps to understand the reasoning behind it by asking why, how, or what. It includes nominal and ordinal data. For example, gender of person, race of a person, smartphone brand, hair color type, marital status, and occupation of a person.

Tutorial 1.1: To implement creating a data frame consisting of only qualitative data.

To create a data frame with pandas, **import pandas as pd**, then use the **DataFrame()** function and pass a data source, such as a dictionary, list, or array, as an argument.

```

1. #Import the pandas library to create a pandas DataFrame
2. import pandas as pd
3. # Sample qualitative data
4. qualitative_data = {
5.     'Name': ['John', 'Alice', 'Bob', 'Eve', 'Michael'],
6.     'City': ['New York', 'Los Angeles', 'Chicago', 'San Francisco', 'Miami'],
7.     'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
8.     'Occupation': ['Engineer', 'Artist', 'Teacher', 'Doctor', 'Lawyer'],
9.     'Race': ['Black', 'White', 'Asian', 'Indian', 'Mongolian'],
10.    'Smartphone Brand': ['Apple', 'Samsung', 'Xiaomi', 'Apple', 'Google']
11. }
12. # Create the DataFrame
13. qualitative_df = pd.DataFrame(qualitative_data)
14. # Prints the created DataFrame
15. print(qualitative_df)

```

Output:

```

1.   Name  City  Gender Occupation  Race
   Smartphone Brand
2. 0  John  New York  Male   Engineer  Black
   Apple
3. 1  Alice  Los Angeles  Female  Artist   White
   Samsung
4. 2  Bob   Chicago   Male   Teacher  Asian   Xiaomi
5. 3  Eve   San Francisco  Female  Doctor   Indian
   Apple
6. 4  Michael  Miami   Male   Lawyer
   Mongolian  Google

```

Row consisting of numbers 0, 1, 2, 3, and 4 is the index column, not part of the qualitative data. To exclude it from output, hide the index column using **to_string()** as follows:

```
1. print(qualtitative_df.to_string(index=False))
```

Output:

1.	Name	City	Gender	Occupation	Race	Smartphone Brand
2.	John	New York	Male	Engineer	Black	Apple
3.	Alice	Los Angeles	Female	Artist	White	Samsung
4.	Bob	Chicago	Male	Teacher	Asian	Xiaomi
5.	Eve	San Francisco	Female	Doctor	Indian	Apple
6.	Michael	Miami	Male	Lawyer	Mongolian	Google

While we often think of data in terms of numbers, many other forms such as images, audio, videos, and text they can also represent quantitative information when suitably encoded (e.g., pixel intensity values in images, audio waveforms, or textual features like word counts).

Tutorial 1.2: To implement accessing and creating a data frame consisting of the image data.

In this tutorial, we'll work with the open-source Olivetti faces dataset, which consists of grayscale face images collected at AT&T Laboratories Cambridge between April 1992 and April 1994. Each face is represented by numerical pixel values, making them a form of quantitative data. By organizing this data into a DataFrame, we can easily manipulate, analyze, and visualize it for further insights.

To create a data frame consisting of the *Olivetti faces*

dataset, you can use the following steps:

1. Fetch the **Olivetti faces** dataset from sklearn using the **sklearn.datasets.fetch_olivetti_faces** function. This will return an object that holds the data and some metadata.
2. Use the **pandas.DataFrame** constructor to create a data frame from the data and the feature names. You can also add a column for the target labels using the target and **target_names** attributes of the object.
3. Use the pandas method to display and analyze the data frame. For example, you can use **df.head()**, **df.describe()**, **df.info()**.

```
1. import pandas as pd
2. # Import datasets from the sklearn library
3. from sklearn import datasets
4. # Fetch the Olivetti faces dataset
5. faces = datasets.fetch_olivetti_faces()
6. # Create a dataframe from the data and feature names
7. df = pd.DataFrame(faces.data)
8. # Add a column for the target labels
9. df["target"] = faces.target
10. # Display the first 3 rows of the dataframe
11. print(f"{df.head(3)}")
12. # Print new line
13. print("\n")
14. # Display the first image in the dataset
15. import matplotlib.pyplot as plt
16. plt.imshow(df.iloc[0, :-1].values.reshape(64, 64), cmap="gray")
17. plt.title(f"Image of person {df.iloc[0, -1]}")
```

18. plt.show()

Quantitative data

Quantitative data is measurable and can be expressed numerically. It is useful for statistical analysis and mathematical calculations. For example, if you inquire about the number of books people have read in a month, their responses constitute quantitative data. They may reveal that they have read, let us say, **three books, zero books, or ten books**, providing information about their reading habits. Quantitative data is easily comparable and allows for calculations. It can provide answers to questions such as **How many? How much? How often?** and **How fast?**

Tutorial 1.3: To implement creating a data frame consisting of only quantitative data is as follows:

```
1. #Import the pandas library to create a pandas DataFrame
   e
2. import pandas as pd
3. quantitative_df = pd.DataFrame({
4.     "price": [300000, 250000, 400000, 350000, 450000],
5.     "distance": [10, 15, 20, 25, 30],
6.     "height": [170, 180, 190, 160, 175],
7.     "weight": [70, 80, 90, 60, 75],
8.     "salary": [5000, 6000, 7000, 8000, 9000],
9.     "temperature": [25, 30, 35, 40, 45],
10. })
11. # Print the DataFrame without index
12. print(quantitative_df.to_string(index=False))
```

Output:

```
1. price distance height weight salary temperature
2. 300000      10    170     70    5000         25
```

3.	250000	15	180	80	6000	30
4.	400000	20	190	90	7000	35
5.	350000	25	160	60	8000	40
6.	450000	30	175	75	9000	45

Tutorial 1.4: To implement accessing and creating a data frame by loading the tabular **iris** data.

Iris tabular dataset contains 150 samples of iris flowers with four features, that is, sepal length, sepal width, petal length, and petal width and three classes, that is, setosa, versicolor, and virginica. The sepal length, sepal width, petal length, petal width, and target (class) are columns of the table¹.

To create a data frame consisting of the **iris** dataset, you can use the following steps:

1. First, you need to load the **iris** dataset from sklearn using the **sklearn.datasets.load_iris** function. This will return a bunch object that holds the data and some metadata.
2. Next, you can use the **pandas.DataFrame** constructor to create a data frame from the data and the feature names. You can also add a column for the target labels using the target and **target_names** attributes of the bunch object.
1. Finally, you can use the panda method to display and analyze the data frame. For example, you can use **df.head()**, **df.describe()**, **df.info()** as follows:

```

1. import pandas as pd
2. # Import dataset from sklearn
3. from sklearn import datasets
4. # Load the iris dataset
5. iris = datasets.load_iris()
6. # Create a dataframe from the data and feature nam

```



```

es
7. df = pd.DataFrame(iris.data, columns=iris.feature_names)
8. # Add a column for the target labels
9. df["target"] = iris.target
10. # Display the first 5 rows of the dataframe
11. df.head()

```

Level of measurement

Level of measurement is a way of classifying data based on how precise it is and what we can do with it. Generally, they are four, that is, nominal, ordinal, interval and ratio. Nominal is a category with no inherent order, such as colors. Ordinal is a category with a meaningful order, such as education levels. Interval is equal intervals but no true zero, such as temperature in degrees Celsius, and ratio are equal intervals with a true zero, such as age in years.

Nominal data

Nominal data is qualitative data that does not have a natural ordering or ranking. For example, gender, religion, ethnicity, color, brand ownership of electronic appliances, and person's favorite meal.

Tutorial 1.5: To implement creating a data frame consisting of qualitative nominal data, is as follows:

```

1. #Import the pandas library to create a pandas Dataframe
2. import pandas as pd
3. nominal_data = {
4.     "Gender": ["Male", "Female", "Male", "Female", "Male"],
5.     "Religion": ["Hindu", "Muslim", "Christian", "Buddhist"]

```

```

    ", "Jewish"],
6.   "Ethnicity": ["Indian", "Pakistani", "American", "Chinese", "Israeli"],
7.   "Color": ["Red", "Green", "Blue", "Yellow", "White"],
8.   "Electronic Appliances Ownership": ["Samsung", "LG", "Apple", "Huawei", "Sony"],
9.   "Person Favorite Meal": ["Biryani", "Kebab", "Pizza", "Noodles", "Falafel"],
10.  "Pet Preference": ["Dog", "Cat", "Parrot", "Fish", "Hamster"]
11. }
12. # Create the DataFrame
13. nominal_df = pd.DataFrame(nominal_data)
14. # Display the DataFrame
15. print(nominal_df)

```

Output:

```

1. Gender  Religion  Ethnicity  Color  Electronic Appliance
   s Ownership \
2. 0  Male    Hindu    Indian    Red                      Samsu
   ng
3. 1  Female  Muslim  Pakistani  Grezn
   LG
4. 2  Male  Christian  American  Blue                      App
   le
5. 3  Female  Buddhist  Chinese  Yellow                      Hu
   awei
6. 4  Male    Jewish  Israeli  Whie                      Sony

7.
8.  Person Favorite Meal Pet Preference
9. 0                      Biryani        Dog
10. 1                      Kebab        Cat

```

11. 2	Pizza	Parrot
12. 3	Noodles	Fish
13. 4	Falafel	Hamster

Ordinal data

Ordinal data is qualitative data that has a natural ordering or ranking. For example, student ranking in class (1st, 2nd, or 3rd), educational qualification (high school, undergraduate, or graduate), satisfaction level (bad, average, or good), income level range, level of agreement (agree, neutral, or disagree).

Tutorial 1.6: To implement creating a data frame consisting of qualitative ordinal data is as follows:

```

1. import pandas as pd
2. ordinal_data = {
3.     "Student Rank in a Class": ["1st", "2nd", "3rd", "4th",
4.     "5th"],
5.     "Educational Qualification": ["Graduate", "Undergrad
6.     uate", "High School", "Graduate", "Undergraduate"],
7.     "Satisfaction Level": ["Good", "Average", "Bad", "Aver
8.     age", "Good"],
9.     "Income Level Range": ["80,000-100,000", "60,000-
10.    80,000", "40,000-60,000", "100,000-120,000", "50,000-
11.    70,000"],
12.     "Level of Agreement": ["Agree", "Neutral", "Disagree"
13.     , "Neutral", "Agree"]
14. }
15. ordinal_df = pd.DataFrame(ordinal_data)
16. print(ordinal_df)

```

Output:

```

1. Student Rank in a Class Educational Qualification Sati
2. sfaction Level \
3. 0 1st Graduate Good

```

3.	1	2nd	Undergraduate	Average
4.	2	3rd	High School	Bad
5.	3	4th	Graduate	Average
6.	4	5th	Undergraduate	Good
7.				
8.	Income Level Range Level of Agreement			
9.	0	80,000-100,000	Agree	
10.	1	60,000-80,000	Neutral	
11.	2	40,000-60,000	Disagree	
12.	3	100,000-120,000	Neutral	
13.	4	50,000-70,000	Agree	

Discrete data

Discrete data is quantitative data, integers or whole numbers, they cannot be subdivided into parts. For example, total number of students present in a class, cost of a cell phone, number of employees in a company, total number of players who participated in a competition, days in a week, number of books in a library, etc. For example, number of coins in a jar, it can only be a whole number like 1,2,3 and so on.

Tutorial 1.7: To implement creating a data frame consisting of quantitative discrete data is as follows:

```

1. import pandas as pd
2. discrete_data = {
3.     "Students": [25, 30, 35, 40, 45],
4.     "Cost": [500, 600, 700, 800, 900],
5.     "Employees": [100, 150, 200, 250, 300],
6.     "Players": [50, 40, 30, 20, 10],
7.     "Week": [7, 7, 7, 7, 7]
8. }
```

```
9. discrete_df = pd.DataFrame(discrete_data)
10. discrete_df
```

Output:

```
1. Students Cost Employees Players Week
2. 0 25 500 100 50 7
3. 1 30 600 150 40 7
4. 2 35 700 200 30 7
5. 3 40 800 250 20 7
6. 4 45 900 300 10 7
```

Continuous data

Continuous data is quantitative data that can take any value (including fractional value) within a range and have no gaps between them. No gaps mean that if a person's height is 1.75 meters, there is always a possibility of height being between 1.75 and 1.76 meters, such as 1.751 or 1.755 meters.

Interval data

Interval data is quantitative numerical data with inherent order. They always have an arbitrary zero, an arbitrary zero meaning no meaningful zero, chosen by convention, not by nature. For example, a temperature of zero degrees Fahrenheit does not mean that there is no heat or temperature, here, zero is an arbitrary zero point. For example, temperature (Celsius or Fahrenheit), GMAT score (200-800), SAT score (400-1600).

Tutorial 1.8: To implement creating a data frame consisting of quantitative interval data is as follows:

```
1. import pandas as pd
2. interval_data = {
3.     "Temperature": [10, 15, 20, 25, 30],
4.     "GMAT_Score": [600, 650, 700, 750, 800],
```

```

5. "SAT_Score (400 - 1600)": [1200, 1300, 1400, 1500, 1600],
6. "Time": ["9:00", "10:00", "11:00", "12:00", "13:00"]
7. }
8. interval_df = pd.DataFrame(interval_data)
9. # Print DataFrame as it is without print() also
10. interval_df

```

Output:

```

1. Temperature GMAT_Score SAT_Score (400 - 1600) Time
2. 0 10 600 1200 9:00
3. 1 15 650 1300 10:00
4. 2 20 700 1400 11:00
5. 3 25 750 1500 12:00
6. 4 30 800 1600 13:00

```

Ratio data

Ratio data is naturally, numerical ordered data with an absolute, where zero is not arbitrary but meaningful. For example, height, weight, age, tax amount has true zero point that is fixed by nature, and they are measured on a ratio scale. Zero height means no height at all, like a point in space. There is nothing shorter than zero height. Zero tax amount means no tax at all, like being exempt. There is nothing lower than zero tax amount.

Tutorial 1.9: To implement creating a data frame consisting of quantitative ratio data is as follows:

```

1. import pandas as pd
2. ratio_data = {
3.     "Height": [170, 180, 190, 200, 210],
4.     "Weight": [60, 70, 80, 90, 100],
5.     "Age": [20, 25, 30, 35, 40],

```

```
6.     "Speed": [80, 90, 100, 110, 120],
7.     "Tax Amount": [1000, 1500, 2000, 2500, 3000]
8. }
9. ratio_df = pd.DataFrame(ratio_data)
10. ratio_df
```

Output:

```
1. Height Weight Age Speed Tax Amount
2. 0 170 60 20 80 1000
3. 1 180 70 25 90 1500
4. 2 190 80 30 100 2000
5. 3 200 90 35 110 2500
6. 4 210 100 40 120 3000
```

Tutorial 1.10: To implement loading the ratio data in a JSON format and displaying it.

Sometimes, data can be in **JSON**. The data used in the following Tutorial 1.10 is in **JSON** format. In that case **json.loads()** method can load it. **JSON** is a text format for data interchange based on JavaScript as follows:

```
1. # Import json
2. import json
3. # The JSON string:
4. json_data = """
5. [
6. {
7.     "Height": 170,
8.     "Weight": 60,
9.     "Age": 20,
10.    "Speed": 80,
11.    "Tax Amount": 1000
12. },
13. {
```

```
14.     "Height": 180,
15.     "Weight": 70,
16.     "Age": 25,
17.     "Speed": 90,
18.     "Tax Amount": 1500
19. },
20. {
21.     "Height": 190,
22.     "Weight": 80,
23.     "Age": 30,
24.     "Speed": 100,
25.     "Tax Amount": 2000
26. },
27. {
28.     "Height": 200,
29.     "Weight": 90,
30.     "Age": 35,
31.     "Speed": 110,
32.     "Tax Amount": 2500
33. },
34. {
35.     "Height": 210,
36.     "Weight": 100,
37.     "Age": 40,
38.     "Speed": 120,
39.     "Tax Amount": 3000
40. }
41. ]
42. ""
43. # Convert to Python object (list of dicts):
```



```
44. data = json.loads(json_data)
```

```
45. data
```

Output:

1. [{ 'Height': 170, 'Weight': 60, 'Age': 20, 'Speed': 80, 'Tax Amount': 1000},
2. { 'Height': 180, 'Weight': 70, 'Age': 25, 'Speed': 90, 'Tax Amount': 1500},
3. { 'Height': 190, 'Weight': 80, 'Age': 30, 'Speed': 100, 'Tax Amount': 2000},
4. { 'Height': 200, 'Weight': 90, 'Age': 35, 'Speed': 110, 'Tax Amount': 2500},
5. { 'Height': 210, 'Weight': 100, 'Age': 40, 'Speed': 120, 'Tax Amount': 3000}]

Distinguishing qualitative and quantitative data

As discussed above, qualitative data describes the quality or nature of something, such as color, shape, taste, or opinion etc. whereas quantitative data is the data that measures the quantity or amount of something, such as length, weight, speed, or frequency. Qualitative data can be further classified as nominal (categorical) or ordinal (ranked). Quantitative data can be further classified as discrete (countable) or continuous (measurable). The following methods are used to understand if data is qualitative or quantitative in nature.

dtype(): It is used to check the data types of the data frame.

Tutorial 1.11: To implement **dtype()** to check the datatypes of the different features or column in a data frame, as follows:

1. `import pandas as pd`
2. *# Create a data frame with qualitative and quantitative*

columns

```
3. df = pd.DataFrame({
4.     "age": [25, 30, 35], # a quantitative column
5.     "gender": ["female", "male", "male"], # a qualitative
   column
6.     "hair color": ["black", "brown", "white"], # a qualitati
   ve column
7.     "marital status": ["single", "married", "divorced"], #
   a qualitative column
8.     "salary": [5000, 6000, 7000], # a quantitative column
9.     "height": [6, 5.7, 5.5], # a quantitative column
10.    "weight": [60, 57, 55] # a quantitative column
11. })
12. # Print the data frame
13. print(df)
14. # Print the data types of each column using dtype()
15. print(df.dtypes)
```

Output:

```
1.  age  gender  hair color  marital status  salary  height  w
   eight
2.  0   25  female    black        single   5000    6.0    60
3.  1   30   male    brown        married   6000    5.7    57
4.  2   35   male    white        divorced   7000    5.5    55
5.  age                int64
6.  gender              object
7.  hair color          object
8.  marital status      object
9.  salary              int64
10. height              float64
11. weight              int64
```

describe(): You can also use the describe method from pandas to generate descriptive statistics for each column. This method will only show statistics for quantitative columns by default, such as mean, standard deviation, minimum, maximum, etc.

You need to specify **include='O'** as an argument to include qualitative columns. This will show statistics for qualitative columns, such as count, unique values, top values, and frequency. As you can see, the descriptive statistics for qualitative and quantitative columns are different, reflecting the nature of the data.

Tutorial 1.12: To implement **describe()** in the data frame used in *Tutorial 1.11* of **dtype()**, is as follows:

1. *# Print the descriptive statistics for quantitative columns*
2. `print(df.describe())`
3. *# Print the descriptive statistics for qualitative columns*
4. `print(df.describe(include='O'))`

Output:

1.	age	salary	height	weight	
2.	count	3.0	3.0	3.000000	3.000000
3.	mean	30.0	6000.0	5.733333	57.333333
4.	std	5.0	1000.0	0.251661	2.516611
5.	min	25.0	5000.0	5.500000	55.000000
6.	25%	27.5	5500.0	5.600000	56.000000
7.	50%	30.0	6000.0	5.700000	57.000000
8.	75%	32.5	6500.0	5.850000	58.500000
9.	max	35.0	7000.0	6.000000	60.000000
10.	gender	hair	color	marital	status
11.	count	3	3	3	
12.	unique	2	3	3	
13.	top	male	black	single	
14.	freq	2	1	1	

value_counts(): To count unique values in a data frame, the **value_counts()** is used. It also displays the data type **dtype**. The **dtype** displays is the data type of the values in the series object returned by the **value_counts** method.

Tutorial 1.13: To implement **value_count()** to count unique value in a data frame as follows:

```
1. # To count the values in `gender` column
2. print(df['gender'].value_counts())
3. print("\n")
4. # To count the values in `age` column
5. print(df['age'].value_counts())
```

Output:

```
1. gender
2. male    2
3. female  1
4. Name: count, dtype: int64
5.
6. age
7. 25    1
8. 30    1
9. 35    1
```

In above *Tutorial 1.13* of **value_counts()**, the values are counts of each unique value in the gender column of the data frame, and the data type is **int64**, which means 64-bit integer.

is_numeric_dtype(), is_string_dtype(): These functions from the **pandas.api.types** module can help you determine if a column contains numeric or string (object) data.

Tutorial 1.14: To implement checking the numeric and string data type of a data frame column with **is_numeric_dtype()** and **is_string_dtype()** functions is as follows:

1. *# Import module for data type checking and inference.*
2. `import pandas.api.types as ptypes`
3. *# Checks if the column 'hair color' in df is of the string dtype and prints the result*
4. `print(f"Is string?: {ptypes.is_string_dtype(df['hair color'])}")`
5. *# Checks if the column 'weight' in df is of the numeric dtype and prints the result*
6. `print(f"Is numeric?: {ptypes.is_numeric_dtype(df['weight'])}")`
7. *# Checks if the column 'salary' in df is of the string dtype and prints the result*
8. `print(f"Is string?: {ptypes.is_string_dtype(df['salary'])}")`

Output:

1. Is string?: `True`
2. Is numeric?: `True`
3. Is string?: `False`

Also, in *Tutorial 1.14* we can use a for loop to check it for all the columns iteratively as follows:

1. *# Check the data types of each column using is_numeric_dtype() and is_string_dtype()*
2. `for col in df.columns:`
3. `print(f"{col}:")`
4. `print(f"Is numeric? {ptypes.is_numeric_dtype(df[col])}")`
5. `print(f"Is string? {ptypes.is_string_dtype(df[col])}")`
6. `print()`

info(): It describes the data frame with a column name, the number of not null values, and the data type of each column.

Tutorial 1.15: To implement **info()** to view the information

about a data frame is as follows:

```
1. df.info()
```

The output will display the summary consisting of column names, **non-null** count values, data types of each column, and many more.

```
1. RangeIndex: 3 entries, 0 to 2
```

```
2. Data columns (total 7 columns):
```

```
3. #   Column      Non-Null Count  Dtype
```

```
4. ---  ---
```

```
5. 0   age          3 non-null    int64
```

```
6. 1   gender        3 non-null    object
```

```
7. 2   hair color     3 non-null    object
```

```
8. 3   marital status 3 non-null    object
```

```
9. 4   salary         3 non-null    int64
```

```
10. 5   height         3 non-null    float64
```

```
11. 6   weight         3 non-null    int64
```

```
12. dtypes: float64(1), int64(3), object(3)
```

```
13. memory usage: 296.0+ bytes
```

head() and tail(): **head()** displays the data frame from the top, and the **tail()** displays it from the last or bottom.

Tutorial 1.16: To implement **head()** and **tail()** to view the top and bottom rows of data frame respectively.

head() displays the data frame with the first few rows. Inside the parenthesis of **head()**, we can define the number of rows we want to view. For example, to view the first ten rows of the data frame, write **head(10)**. Same with **tail()** also as follows:

```
1. # View the first few rows of the DataFrame
```

```
2. print(df.head())
```

```
3. # View first 2 rows of the DataFrame
```

```
4. df.head(1)
```

The output will display the data frame from the top, and **head(1)** will display the topmost row of data frame as follows:

```
1. age gender hair color marital status salary height weight
2. 0 25 female black single 5000 6.0 60
3. 1 30 male brown married 6000 5.7 57
4. 2 35 male white divorced 7000 5.5 55
5. age gender hair color marital status salary height weight
6. 0 25 female black single 5000 6.0 60
```

Tutorial 1.17: To implement **tail()** and display the bottom most rows of data frame is as follows:

```
1. # Import display function from IPython module to display the dataframe
2. from IPython.display import display
3. # View the last few rows of the DataFrame
4. display(df.tail())
5. # View last 2 rows of the DataFrame
6. display(df.tail(1))
```

In output **tail(1)** will display the bottommost row of data frame as follows:

```
1. age gender hair color marital status salary height weight
2. 0 25 female black single 5000 6.0 60
3. 1 30 male brown married 6000 5.7 57
4. 2 35 male white divorced 7000 5.5 55
5. age gender hair color marital status salary height weight
6. 2 35 male white divorced 7000 5.5 55
```

Other methods: Besides function described above there

are few other methods in Python that are useful to distinguish qualitative and quantitative data. They are as follows:

- **select_dtypes(include='__')**: It is used to select columns with specific datatype that is, number and object.

1. *# Select and display DataFrame with only numeric values*

2. `display(df.select_dtypes(include='number'))`

The output will display only the numeric column of the data frame as follows:

1. `age salary height weight`

2. `0 25 5000 6.0 60`

3. `1 30 6000 5.7 57`

4. `2 35 7000 5.5 55`

To select only object data types, an object is used. In pandas, objects are the column containing strings or mixed types of data. It is the default data type for columns that have text or arbitrary Python objects as follows:

1. *# Select and display DataFrame with only object values in the same cell(display() displays DataFrame in same cell)*

2. `display(df.select_dtypes(include='object'))`

Output will include only object type columns as follows:

1. `gender hair color marital status`

2. `0 female black single`

3. `1 male brown married`

4. `2 male white divorced`

- **groupby(): groupby()** is used to group based on column as follows:

1. *# Group by gender*

2. `df.groupby("gender")`

After grouping based on the column name, it can be used to display the summary statistics of a grouped data frame. **describe()** method on the **groupby()** object can be used to find summary statistics of each group as follows:

1. `# Describe dataframe summary statistics by gender`
2. `df.groupby('gender').describe()`

Further to print the count of each group, you can use the **size()** or **count()** method on the **groupby()** object as follows:

1. `# Print count of group object with size`
2. `print(df.groupby('gender').size())`
3. `# Print count of group object with count`
4. `print(df.groupby('gender').count())`

Output:

```
1. gender
2. female    1
3. male      2
4. dtype: int64
5. age  hair color  marital status  salary  height  weight
6. gender
7. female    1    1              1      1      1      1
8. male      2    2              2      2      2      2
```

- **groupby().sum():** **groupby().sum()** groups data and then display sum in each group as follows:

1. `# Group by gender and hair color and calculate the sum of each group`
2. `df.groupby(["gender", "hair color"]).sum()`

- **columns:** **columns** display column names. Sometimes, through descriptive columns names, types of data can

be distinguished. So, displaying column name can be useful as follows:

1. *# Displays all column names.*

2. `df.columns`

- **type():** **type()** is used to display the type of a variable. It can be used to determine the type of a single variable as follows:

1. *# Declare variable*

2. `x = 42`

3. `y = "Hello"`

4. `z = [1, 2, 3]`

5. *# Print data types*

6. `print(type(x))`

7. `print(type(y))`

8. `print(type(z))`

Tutorial 1.18: To implement **read_json()**, to read and view nobel prize dataset in JSON format.

Let us load a *nobel prizdataset*² and see what kind of data it contains. The *Tutorial 1.18* flattens nested **JSON** data structures into a data frame as follows:

1. `import pandas as pd`

2. *# Read the json file from the direcotory*

3. `json_df = pd.read_json("/workspaces/ImplementingStatisticsWithPython/data/chapter1/prize.json")`

4. *# Convert the json data into a dataframe*

5. `data = json_df["prizes"]`

6. `prize_df = pd.json_normalize(data)`

7. *# Display the dataframe*

8. `prize_df`

To see what type of data **prize_df** contains use **info()** and **head()**, is as follows:

1. `prize_df.info()`

```
2. prize_df.head()
```

Alternatively, to *Tutorial 1.18*, the *nobel prize dataset*³ can be accessed directly by sending the request as shown in the following code:

```
1. import pandas as pd
2. # Send HTTP requests using Python
3. import requests
4. # Get the json data from the url
5. response = requests.get("https://api.nobelprize.org/v1/
    prize.json")
6. data = response.json()
7. # Convert the json data into a dataframe
8. prize_json_df = pd.json_normalize(data, record_path="p
    rizes")
9. prize_json_df
```

Tutorial 1.19: To implement **read_csv()**, to read and view nobel prize dataset in CSV format is as follows:

```
1. import pandas as pd
2. # Read the json file from the direcotory
3. prize_csv_df = pd.read_csv("/workspaces/Implementing
    StatisticsWithPython/data/chapter1/prize.csv")
4. # Display the dataframe
5. prize_csv_df
```

Tutorial 1.20: To implement use of NumPy and to read diabetes dataset in CSV files.

Most common ways are using **numpy.loadtxt()** and using **numpy.genfromtxt()**. **numpy.loadtxt()** assumes that the file has no missing values, no comments, and no headers and uses whitespace as the delimiter by default. We can change the delimiter to a comma by passing **delimiter = ','** as a parameter. Here, the CSV file has one header row, which is a string, so we use **skiprows = 1** this skips the

first row of the CSV file and loads the rest of the data as a NumPy array as follows:

```
1. import numpy as np
2. arr = np.loadtxt('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv', delimiter=',', skiprows=1)
3. print(arr)
```

The **numpy.genfromtxt()** function can handle missing values, comments, headers, and various delimiters. We can use the **missing_values** parameter to specify which values to treat as missing. We can use the **comments** parameter to specify which character indicates a comment line, such as # or %. For example, if you have a CSV file named **diabetes.csv** that looks as follows:

```
1. import numpy as np
2. arr = np.genfromtxt('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv', delimiter=',', names=True, missing_values='?', dtype=None)
3. print(arr)
```

Univariate, bivariate, and multivariate data

Univariate, bivariate, and multivariate data are terms used in statistics to describe the number of variables and their relationships within a dataset. Where univariate means one, bivariate means two and multivariate is more than two. These concepts are fundamental to statistical analysis and play a crucial role in various fields, from social sciences to natural sciences, engineering and beyond.

Univariate data and univariate analysis

Univariate analysis involves observing only one variable or attribute. For example, height of students in a class, color of cars in a parking lot, or salary of employees in a

company are all univariate data. Univariate analysis analyzes only one variable column or attribute at a time. For example, analyzing only the patient height column at a time or the person's salary column.

Tutorial 1.21: To implement univariate data and univariate analysis by selecting a column or variable or attribute from the CSV dataset and compute its mean, standard deviation, frequency or distribution with other information using **describe()** as follows:

```
1. import pandas as pd
2. from IPython.display import display
3. diabeties_df = pd.read_csv('/workspaces/ImplementingS
   tatisticsWithPython/data/chapter1/diabetes.csv')
4. #To view all the column names
5. print(diabeties_df.columns)
6. # Select the column Glucose column as a DataFrame fro
   m diabeties_df DataFrame
7. display(diabeties_df[['Glucose']])
8. # describe() gives the mean,standard deviation
9. print(diabeties_df[['Glucose']].describe())
```

In the above, we selected only the **Glucose** column of the **diabeties_df** and analyzed only that column. This kind of single-column analysis is univariate analysis.

Tutorial 1.22: To further implement computation of median, mode range, frequency or distribution of variables in continuation with *Tutorial 1.21*, is as follows:

```
1. # Use mode() for computing most frequest value i.e, mo
   de
2. print(diabeties_df[['Glucose']].mode())
3. # To get range simply subtract DataFrame maximum val
   ue by the DataFrame minimum value. Use df.max() and
   df.min() for maximum and minimum value
4. mode_range = diabeties_df[['Glucose']].max() - diabeties
```

```

    _df[['Glucose']].min()
5. print(mode_range)
6. # For frequency or distribution of variables use value_counts()
7. diabetics_df[['Glucose']].value_counts()

```

Bivariate data

Bivariate data consists of observing two variables or attributes for each individual or unit. For example, if you wanted to study the relationship between the age and height of students in a class, you would collect the age and height of each student. Age and height are two variables or attributes, and each student is an individual or unit. Bivariate analysis analyzes how two different variables, columns, or attributes are related. For example, the correlation between people's height and weight or between hours worked and monthly salary.

Tutorial 1.23: To implement bivariate data and bivariate analysis by selecting two columns or variables or attributes from the CSV dataset and to describe them, as follows:

```

1. import pandas as pd
2. from IPython.display import display
3. diabetics_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')
4. #To view all the column names
5. print(diabetics_df.columns)
6. # Select two column Glucose column as a DataFrame from diabetics_df DataFrame
7. display(diabetics_df[['Glucose','Age']])
8. # describe() gives the mean,standard deviation
9. print(diabetics_df[['Glucose','Age']].describe())
10. # Use mode() for computing most frequent value i.e, mo

```

de

```
11. print(diabities_df[['Glucose']].mode())
12. # To get range simply subtract DataFrame maximum value by the DataFrame minimum value. Use df.max() and df.min() for maximum and minimum value
13. mode_range = diabities_df[['Glucose']].max() - diabities_df[['Glucose']].min()
14. print(mode_range)
15. # For frequency or distribution of variables use value_counts()
16. diabities_df[['Glucose']].value_counts()
```

Here, we compared two columns, **glucose** and **age** in **diabities_df** data frame, which involved multiple data frame columns making it bivariate analysis.

Alternatively two or more columns can be accessed using **loc[row_start:row:stop,column_start:column:stop]** or also through column index via slicing by using **iloc[row_start:row:stop,column_start:column:stop]** as follows:

```
1. # Using loc
2. diabities_df.loc[:, ['Glucose','Age']]
3. # Using iloc, column index and slicing
4. diabities_df.iloc[:,0:2]
```

Further, to compute the correlation between two variables or two columns, such as **glucose** and **age**, we can use columns along with **corr()** as follows:

```
1. diabities_df['Glucose'].corr(diabities_df['Age'])
```

Correlation is a statistical measure that indicates how two variables are related to each other. A positive correlation means that the variables increase or decrease together, while a negative correlation means that the variables move in opposite directions. A correlation value close to zero

means that there is no linear relationship between the variables.

In the context of `diabetes_df['Glucose'].corr(diabetes_df['Age'])`, the random positive correlation value of **0.26** means that there is a weak positive correlation between glucose level and age in the diabetes dataset. This implies that older people tend to have higher glucose levels than younger people but the relationship is not very strong or consistent. Correlation can be computed using different methods such as **pearson**, **kendall**, or **spearman** then specify `method = '__'` in `corr()` as follows:

```
1. diabetes_df['Glucose'].corr(diabetes_df['Age'], method='kendall')
```

Multivariate data

Multivariate data consists of observing three or more variables or attributes for each individual or unit. For example, if you want to study the relationship between the age, gender, and income of customers in a store, you would collect this data for each customer. Age, gender, and income are the three variables or attributes, and each customer is an individual or unit. In this case, the data you collect will be multivariate data because it requires observations on three variables or attributes for each individual or unit. For example, the correlation between age, gender, and sales in a store or between temperature, humidity, and air quality in a city.

Tutorial 1.24: To implement multivariate data and multivariate analysis by selecting multiple columns or variables or attributes from the CSV dataset and describe them, as follows:

```
1. import pandas as pd
2. from IPython.display import display
```



```

3. diabetics_df = pd.read_csv('/workspaces/ImplementingS
   tatisticsWithPython/data/chapter1/diabetes.csv')
4. #To view all the column names
5. print(diabetics_df.columns)
6. # Select the column Glucose column as a DataFrame fro
   m diabetics_df DataFrame
7. display(diabetics_df[['Glucose','BMI', 'Age', 'Outcome']])
8. # describe() gives the mean,standard deviation
9. print(diabetics_df[['Glucose','BMI', 'Age', 'Outcome']].de
   scribe())

```

Alternatively, multivariate analysis can be performed by describing the whole data frame as follows:

```

1. # describe() gives the mean,standard deviation
2. print(diabetics_df.describe())
3. # Use mode() for computing most frequent value i.e, mo
   de
4. print(diabetics_df.mode())
5. # To get range simply subtract DataFrame maximum val
   ue by the DataFrame minimum value. Use df.max() and
   df.min() for maximum and minimum value
6. mode_range = diabetics_df.max() - diabetics_df.min()
7. print(mode_range)
8. # For frequency or distribution of variables use value_co
   unts()
9. diabetics_df.value_counts()

```

Further, to compute the correlation between all the variables in the data frame, use **corr()** after the data frame variable name as follows:

```

1. diabetics_df.corr()

```

You can also apply various multivariate analysis techniques, as follows:

- **Principal Component Analysis (PCA):** It transforms

high-dimensional data into a smaller set of uncorrelated variables (principal components) that capture the most variance, thereby simplifying the dataset while retaining essential information. It makes easier to visualize, interpret, and model multivariate relationships

- **Library:** Scikit-learn
- **Method:** **PCA(n_components=___)**
- **Multivariate regression:** This is used to analyze the relationship between multiple dependent and independent variables.
 - **Library:** Statsmodels
 - **Method:** **statsmodels.api.OLS** for ordinary least squares regression. It allows you to perform multivariate linear regression and analyze the relationship between multiple dependent and independent variables. Regression can also be performed using scikit-learn's **LinearRegression()**, **LogisticRegression()**, and many more.
- **Cluster analysis:** This is used to group similar data points together based on their characteristics.
 - **Library:** Scikit-learn
 - **Method:** **sklearn.cluster.KMeans** for K-means clustering. It allows you to group similar data points together based on their characteristics. And many more.
- **Factor analysis:** This is used to identify underlying latent variables that explain the observed variance.
 - **Library:** FactorAnalyzer
 - **Method:** **FactorAnalyzer** for factor analysis. It allows you to perform **Exploratory Factor**

Analysis (EFA) to identify underlying latent variables that explain the observed variance.

- **Canonical Correlation Analysis (CCA):** To explore the relationship between two sets of variables.
 - **Library:** Scikit-learn
 - **Method:** **sklearn.cross_decomposition** and CCA allows you to explore the relationship between two sets of variables and find linear combinations that maximize the correlation between the two sets.

Tutorial 1.25: To implement **Principal Component Analysis (PCA)** for dimensionality reduction is as follows:

1. `import pandas as pd`
2. `# Import principal component analysis`
3. `from sklearn.decomposition import PCA`
4. `# Scales data between 0 and 1`
5. `from sklearn.preprocessing import StandardScaler`
6. `# Import matplotlib to plot visualization`
7. `import matplotlib.pyplot as plt`
8. `# Step 1: Load your dataset into a DataFrame`
9. `# Assuming you have your dataset stored in a CSV file called "data.csv", load it into a Pandas DataFrame.`
10. `data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")`
11. `# Step 2: Separate the features and the outcome variable (if applicable)`
12. `# If the "Outcome" column represents the dependent variable and not a feature, you should separate it from the features.`
13. `# If it's not the case, you can skip this step.`
14. `X = data.drop("Outcome", axis=1) # Features`

```
15. y = data["Outcome"] # Outcome (if applicable)
16. # Step 3: Standardize the features
17. # PCA is sensitive to the scale of features, so it's crucial
    to standardize them to have zero mean and unit variance.
18. scaler = StandardScaler()
19. X_scaled = scaler.fit_transform(X)
20. # Step 4: Apply PCA for dimensionality reduction
21. # Create a PCA instance and specify the number of components
    you want to retain.
22. # If you want to reduce the dataset to a certain number
    of dimensions (e.g., 2 or 3), set the 'n_components' accordingly.
23. pca = PCA(n_components=2) # Reduce to 2 principal components
24. X_pca = pca.fit_transform(X_scaled)
25. # Step 5: Explained Variance Ratio
26. # The explained variance ratio gives us an idea of how much
    information each principal component captures.
27. explained_variance_ratio = pca.explained_variance_ratio_
28. # Step 6: Visualize the Explained Variance Ratio
29. plt.bar(range(len(explained_variance_ratio)), explained_
    variance_ratio)
30. plt.xlabel("Principal Component")
31. plt.ylabel("Explained Variance Ratio")
32. plt.title("Explained Variance Ratio for Each Principal Component")
33. # Show the figure
34. plt.savefig('skew_negative.jpg',dpi=600,bbox_inches='tight')
```

```
35. plt.show()
```

PCA reduces the dimensions but it also results in some loss of information as we only retain the most important components. Here, the original 8-dimensional diabetes data set has been transformed into a new 2-dimensional data set. The two new columns represent the first and second principal components, which are linear combinations of the original features. These principal components capture the most significant variation in the data.

The columns of the data set pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, diabetes pedigree function, and age are reduced to 2 principal components because we specify **n_components=2** as shown in [Figure 1.1](#).

Output:

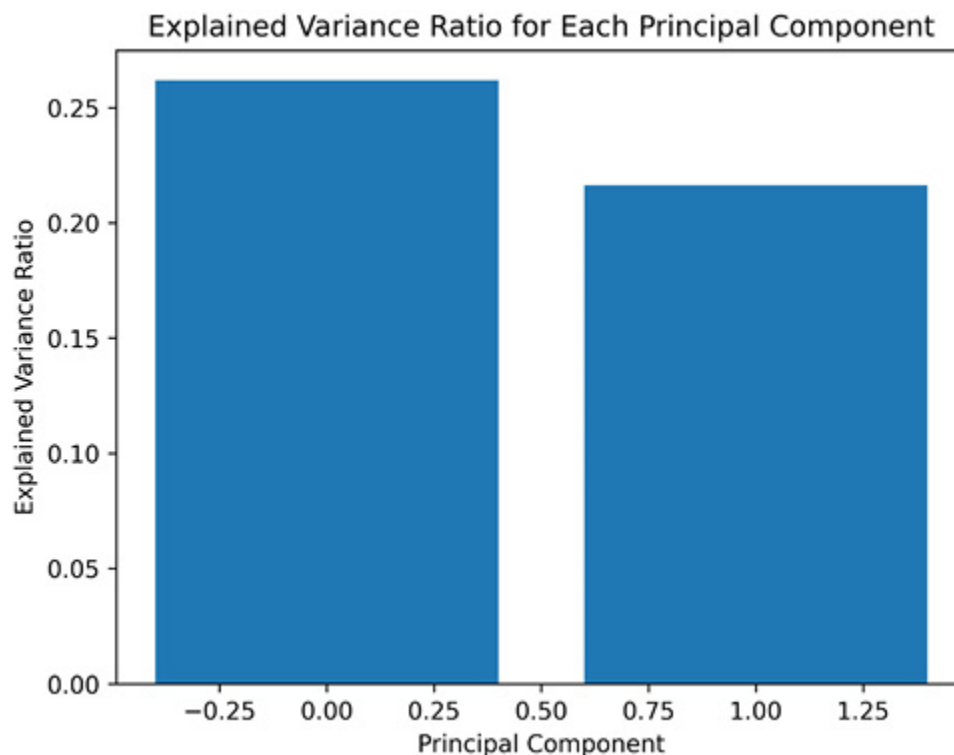


Figure 1.1: Explained variance ratio for each principal component

Following is what you can infer from these explained

variance ratios in this **diabetes dataset**:

- **The First Principal Component (PC1):** With an explained variance of 0.27, PC1 captures the largest portion of the data's variability. It represents the direction in the data space along which the data points exhibit the most significant variation. PC1 is the principal component that explains the most significant patterns in the data.
- **The Second Principal Component (PC2):** With an explained variance of 0.23, PC2 captures the second-largest portion of the data's variability. PC2 is orthogonal (uncorrelated) to PC1, meaning it represents a different direction in the data space from PC1. PC2 captures additional patterns that are not explained by PC1 and provides complementary information. PC1 and PC2 account for approximately 50% ($0.27 + 0.23$) of the total variance.

You can do similar with NumPy and JSON. Also, you can create different types of plots and charts for data analysis using Matplotlib and Seaborn libraries

Data sources, methods, populations, and samples

Data sources provide information for analysis from surveys, databases, or experiments. Collection methods determine how data is gathered, through interviews, questionnaires, or observations. Population is the entire group being studied, while samples are representative subsets used to draw conclusions with less analysis.

Data source

Data can be primary and secondary. It can be of two types, that is, statistical sources like surveys, census, experiments, and statistical reports and non-statistical

sources like business transactions, social media posts, weblogs, data from wearables and sensors, or personal records.

Tutorial 1.26: To implement reading data from different sources and view statistical and non-statistical data is as follows:

1. `import pandas as pd`
2. *# To import urllib library for opening and reading URLs*
3. `import urllib.request`
4. *# To access CSV file replace file name*
5. `df = pd.read_csv('url_to_csv_file.csv')`

To access or read data from different sources, pandas provides **read_csv()** and **read_json()** and **loadtxt()**, **genfromtxt()** in NumPy and many others. The URL can also be used like <https://api.nobelprize.org/v1/prize.json>, but it should be accessible. Most data server would need authentication to access the server.

To read JSON files replace file name in the script as follows:

1. *# To access JSON data replace file name*
2. `df = pd.read_json('your_file_name.json')`

To read XML file from a server with NumPy, you can use the **np.loadtxt()** function and pass as an argument a file object created using the **urllib.request.urlopen()** function from the **urllib.request** module. You must also specify the delimiter parameter as `<` or `>` to separate XML tags from the data values. To read XML file, replace files names with appropriate one in the script as follows:

1. *# To access and read the XML file using URL*
2. `file = urllib.request.urlopen('your_url_to_accessible_xml_file.xml')`
3. *# To open the XML file from the URL and store it in a file*

object

```
4. arr = np.loadtxt(file, delimiter='<')  
5. print(arr)
```

Collection methods

Collection methods are surveys, interviews, observations, focus groups, experiments, and secondary data analysis. It can be quantitative, based on numerical data and statistical analysis, or qualitative, based on words, images, actions, and interpretive analysis. Also, sometimes mixed methods, which combine qualitative and quantitative, can be used.

Population and sample

Population is entire group of people, items, or elements you want to study or draw conclusions about. For example, if you want to know the average score of all students in a school, the population is all students. The sample is a subset of the population from which you select and collect data. For example, 20 randomly chosen students from this school are a population sample.

Let us see an example of selecting population and sample using random modules. The **random** module **rand()** function can be utilized to randomly choose unstructured and semi-structured datasets or files. This approach ensures that each data point has an equal probability of being included in the sample, thereby minimizing selection bias and ensuring the sample's representativeness of the broader population.

Tutorial 1.27: To implement **rand()** to select items from the population, is as follows:

```
1. import random  
2. # Define population and sample size  
3. population = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
4. sample_size = 3
```


5. *# Randomly select a sample from the population*
6. `sample = random.sample(population, sample_size)`
7. `print("Sample:", sample)`

Tutorial 1.28: To implement **rand()** to select items from the patient registry data, is as follows:

1. `import pandas as pd`
2. `from IPython.display import display`
3. *# Import module to generate random numbers*
4. `import random`
5. *# Read CSV file and save as dataframe*
6. `diabities_df = pd.read_csv('/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv')`
7. *# Define the sample size*
8. `sample_size = 5`
9. *# Get the number of rows in the DataFrame*
10. `num_rows = diabities_df.shape[0]`
11. *# Generate random indices for selecting rows*
12. `random_indices = random.sample(range(num_rows), sample_size)`
13. *# Select the rows using the random indices*
14. `sample_diabities_df = diabities_df.iloc[random_indices]`
15. `display(sample_diabities_df)`

While random sampling methods like **rand()** help select a representative subset from a broader population, functions such as **train_test_split()** play a pivotal role in organizing this subset into training and testing sets, particularly in supervised learning. By systematically dividing data into dependent and independent variables and ensuring that these splits are both representative and reproducible, **train_test_split()** facilitates the development of models that perform reliably on unseen data.

Tutorial 1.29: To implement **train_test_split()** to select

items from the patient registry data population, is as follows:

```
1. # Import sklearn train_test_split
2. from sklearn.model_selection import train_test_split
3. # Define population and test size
4. population = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5. test_size = 0.2 # Proportion of data to reserve for testing
6. # Split the population into training and testing sets
7. train_set, test_set = train_test_split(population, test_size=test_size, random_state=42)
8. # Display the split
9. print("Training Set:", train_set)
10. print("Testing Set:", test_set)
```

Output:

```
1. Training Set: [6, 1, 8, 3, 10, 5, 4, 7]
2. Testing Set: [9, 2]
```

Data preparation tasks

Data preparation tasks are the early steps carried out upon having access to the data. It involves checking the quality of data, cleaning of data, data wrangling, and its manipulation described in detail.

Data quality

Data quality indicates how suitable, accurate, useful, complete, reliable, and consistent the data is for its intended use. Verifying data quality is an important step in analysis and preprocessing.

Tutorial 1.30: To implement checking the data quality of CSV file data frame, is as follows:

- Check missing values with **isna()** or **isnull()**
- Check summary with **describe()** or **info()**
- Check shape with **shape**, size with **size**, and memory usage with **memory_usage()**
- Check duplicates with **duplicated()** and remove duplicate with **drop_duplicates()**

Based on this instruction, let us see the implementation as follows:

```
1. import pandas as pd
2. diabeties_df = pd.read_csv('/workspaces/ImplementingS
   tatisticsWithPython/data/chapter1/diabetes.csv')
3. # Check for missing values using isna() or isnull()
4. print(diabeties_df.isna().sum())
5. #Describe the dataframe with describe() or info()
6. print(diabeties_df.describe())
7. # Check for the shape,size and memory usage
8. print(f'Shape: {diabeties_df.shape} Size: {diabeties_df.siz
   e} Memory Usage: {diabeties_df.memory_usage()}')
9. # Check for the duplicates using duplicated() and drop t
   hem if necessary using drop_duplicates()
10. print(diabeties_df.duplicated())
```

Now, we use synthetic transaction narrative data containing unstructured information about the nature of the transaction.

Tutorial 1.30: To implement viewing the text information in the text files (synthetic transaction narrative files), is as follows:

```
1. import pandas as pd
2. import numpy as np
3. # To import glob library for finding files and directories u
   sing patterns
4. import glob
```

```

5. # To assign the path of the directory containing the text
   files to a variable
6. path = "/workspaces/ImplementingStatisticsWithPython
   /data/chapter1/TransactionNarrative"
7. # To find all the files in the directory that have a .txt ext
   ension and store them in a list
8. files = glob.glob(path + "/*.txt")
9. # To loop through each file in the list
10. for file in files:
11.     # To open each file in read mode with utf-
       8 encoding and assign it to a file object
12.     with open(file, "r", encoding="utf-8") as f:
13.         print(f.read())

```

Output:

1. Date: 2023-08-01
2. Merchant: VideoStream Plus
3. Amount: \$9.99
4. Description: Monthly renewal of VideoStream Plus subs
cription.
5. Your subscription to VideoStream Plus has been success
fully renewed for \$9.99.

Tutorial 1.31: To implement checking the data quality of multiple **.txt** files (synthetic transaction narrative files) that contains text information as shown in *Tutorial 1.30* output and to check the quality of information in them, we use **file_size**, **line_count**, **missing_field**, as follows:

```

1. import os
2. import glob
3. def check_file_quality(content):
4.     # Check for presence of required fields
5.     required_fields = ['Date:', 'Merchant:', 'Amount:', 'De
       scription:']

```

```
6.     missing_fields = [field for field in required_fields if field
    not in content]
7.     # Calculate file size
8.     file_size = len(content.encode('utf-8'))
9.     # Count lines in the content
10.    line_count = content.count('\n') + 1
11.    # Return quality assessment
12.    quality_assessment = {
13.        "file_name": file,
14.        "file_size_bytes": file_size,
15.        "line_count": line_count,
16.        "missing_fields": missing_fields
17.    }
18.    return quality_assessment
19. # To assign the path of the directory containing the text
    files to a variable
20. path = "/workspaces/ImplementingStatisticsWithPython
    /data/chapter1/TransactionNarrative"
21. # To find all the files in the directory that have a .txt ext
    ension and store them in a list
22. files = glob.glob(path + "/*.txt")
23. # To loop through each file in the list
24. for file in files:
25.     with open(file, "r", encoding="utf-8") as f:
26.         content = f.read()
27.         print(content)
28.         quality_result = check_file_quality(content)
29.         print(f"\nQuality Assessment for {quality_result['fil
    e_name']}:")
30.         print(f"File Size: {quality_result['file_size_bytes']} b
    ytes")
```

```

31.     print(f"Line Count: {quality_result['line_count']} lin
      es")
32.     if quality_result['missing_fields']:
33.         print("Missing Fields:", ', '.join(quality_result['mi
      ssing_fields']))
34.     else:
35.         print("All required fields present.")
36.     print("=" * 40)

```

Output (Only one transaction narrative output is shown):

```

1. Date: 2023-08-01
2. Merchant: VideoStream Plus
3. Amount: $9.99
4. Description: Monthly renewal of VideoStream Plus subs
  cription.
5.
6. Your subscription to VideoStream Plus has been success
  fully renewed for $9.99.
7.
8.
9. Quality Assessment for /workspaces/ImplementingStati
  sticsWithPython/data/chapter1/TransactionNarrative/3.
  txt:
10. File Size: 201 bytes
11. Line Count: 7 lines
12. All required fields present.
13. =====
    =====

```

Cleaning

Data cleansing involves identifying and resolving inconsistencies and errors in raw data sets to improve data quality. High-quality data is critical to gaining accurate and

meaningful insights. Data cleansing also include data handling. Different ways for data cleaning or data handling are described below.

Missing values

Missing values refer to data points or observations with incomplete or absent information. For example, in a survey, if people do not answer a certain question, the related entries will be empty. Appropriate methods, like imputation or exclusion, are used to address them. If there are missing values then one way is to drop missing value as shown in *Tutorial 1.32*.

Tutorial 1.32: To implement finding the missing value and dropping them.

Let us check **prize_csv_df** data frame for null values and drop the null ones, as follows:

```
1. import pandas as pd
2. from IPython.display import display
3. # Read the prize csv file from the direcotory
4. prize_csv_df = pd.read_csv("/workspaces/Implementing
    StatisticsWithPython/data/chapter1/prize.csv")
5. # Display the dataframe null values count
6. print(prize_csv_df.isna().sum())
```

Output:

```
1. year                374
2. category            374
3. overallMotivation   980
4. laureates__id        49
5. laureates__firstname  50
6. laureates__surname   82
7. laureates__motivation 49
8. laureates__share     49
```

Since **prize_csv_df** have null values, let us drop them and view the count of null values after drop as follows:

1. `print("\n \n **** After dropping the null values in prize_csv_df****")`
2. `after_dropping_null_prize_df = prize_csv_df.dropna()`
3. `print(after_dropping_null_prize_df.isna().sum())`

Finally, after applying the above code, the output will be as follows:

1. `**** After dropping the null values in prize_csv_df****`
2. year 0
3. category 0
4. overallMotivation 0
5. laureates__id 0
6. laureates__firstname 0
7. laureates__surname 0
8. laureates__motivation 0
9. laureates__share 0
10. dtype: int64

This shows there are now zero null values in all the column.

Imputation

Imputation means to place a substitute value in place of the missing values. Like constant value imputation, mean imputation, mode imputation.

Tutorial 1.33: To implement imputing the mean value of the column **laureates__share**.

Mean imputation only imputes the mean value of numeric data types as **fillna()** expects scalar, so we cannot use the **mean()** method to fill missing values in object columns.

1. `import pandas as pd`
2. `from IPython.display import display`
3. `# Read the prize csv file from the direcotory`


```

4. prize_csv_df = pd.read_csv("/workspaces/Implementing
   StatisticsWithPython/data/chapter1/prize.csv")
5. # View the number of null values in original DataFrame
6. print("Null Value Before",prize_csv_df['laureates__share']
   .isna().sum())
7. # Calculate the mean of each column
8. prize_col_mean = prize_csv_df['laureates__share'].mean
   ()
9. # Fill missing values with column mean, inplace = True
   will replace the original DataFrame
10. prize_csv_df['laureates__share'].fillna(value=prize_col_
   mean, inplace=True)
11. # View the number of null values in the new DataFrame
12. print("Null Value After",prize_csv_df['laureates__share']
   .isna().sum())

```

Output:

1. Null Value Before 49
2. Null Value After 0

Also to fill missing values in object columns, you have to use a different strategy, such as using a constant value i.e, **df[column_name].fillna(' ')**, a mode value, or a custom function..

Tutorial 1.34: To implement imputing the mode value in the object data type column.

```

1. import pandas as pd
2. from IPython.display import display
3. # Read the prize csv file from the direcotory
4. prize_csv_df = pd.read_csv("/workspaces/Implementing
   StatisticsWithPython/data/chapter1/prize.csv")
5. # Display the original DataFrame null values in object d
   ata type columns
6. print(prize_csv_df.isna().sum())

```

```

7. # Select the object columns
8. object_cols = prize_csv_df.select_dtypes(include='object').columns
9. # Calculate the mode of each object data type column
10. col_mode = prize_csv_df[object_cols].mode().iloc[0]
11. # Fill missing values with the mode of each object data type column
12. prize_csv_df[object_cols] = prize_csv_df[object_cols].fillna(col_mode)
13. # Display the DataFrame column after filling null values in object data type columns
14. print(prize_csv_df.isna().sum())

```

Output:

```

1. year          374
2. category      374
3. overallMotivation  980
4. laureates__id    49
5. laureates__firstname  50
6. laureates__surname   82
7. laureates__motivation  49
8. laureates__share    49
9. dtype: int64
10. year          374
11. category       0
12. overallMotivation  0
13. laureates__id    49
14. laureates__firstname  0
15. laureates__surname   0
16. laureates__motivation  0
17. laureates__share    49
18. dtype: int64

```

Duplicates

Data may be duplicated or contains duplicate value. The duplicacy will affect the final statistical result. Hence, to prevent duplicacy, identifying and removing duplicates is necessary step as explained in this section. Best way to handle duplicate is to identify and remove duplicates.

Tutorial 1.35: To implement identifying and removing duplicate rows in data frame with **deduplicated()**, as follows:

1. *# Identify duplicate rows and display their index*
2. `print(prize_csv_df.duplicated().index[prize_csv_df.duplicated()])`

Since, there is no duplicate the output is empty it displays indexes of duplicates as follows:

1. `Index([], dtype='int64')`

Also, you can find the duplicate values in a specific column by using the following code:

1. `prize_csv_df.duplicated(subset=['name_of_the_column'])`

To remove duplicates, **drop()** method can be used, syntax will be **dataframe.drop(labels, axis='columns', inplace=False)**. Drop can be applied to row and index using label and index values as follows:

1. `import pandas as pd`
2. *# Create a sample dataframe*
3. `people_df = pd.DataFrame({'name': ['Alice', 'Bob', 'Charlie'], 'age': [25, 30, 35], 'gender': ['F', 'M', 'M']})`
4. *# Print the original dataframe*
5. `print("original dataframe \n", people_df)`
6. *# Drop the 'gender' column and return a new dataframe*
7. `new_df = people_df.drop('gender', axis='columns')`
8. *# Print the new dataframe*
9. `print("dataframe after drop \n", new_df)`

Output:

```
1. original dataframe
2.      name age gender
3. 0  Alice  25    F
4. 1   Bob  30    M
5. 2 Charlie 35    M
6. dataframe after drop
7.      name age
8. 0  Alice  25
9. 1   Bob  30
10. 2 Charlie 35
```

Outliers

Outliers are data points that are very different from the other data points. They can be much higher or lower than the standard range of values. For example, if the heights of ten people in centimeters are measured, the values might be as follows:

160, 165, 170, 175, 180, 185, 190, 195, 200, 1500.

Most of the heights are alike but the last measurement is much larger than the others. This data point is an outlier because it is not like the rest of the data. The best way to handle outliers is to identify outliers and then correct, resolve, or leave as needed. Ways to identify outliers are to compute mean, standard deviation, and quantile (a common approach is to compute interquartile range). Another way to identify outliers is by computing the z-score of the data points and then considering points beyond the threshold values as outliers.

Tutorial 1.36: To implement identifying outliers in a data frame with `zscore`.

Z-score measures how many standard deviations a value is from the mean. In the following code, **z_score** identifies

outliers in the laureates' share column:

```
1. import pandas as pd
2. import numpy as np
3. # Read the prize csv file from the direcotory
4. prize_csv_df = pd.read_csv("/workspaces/Implementing
   StatisticsWithPython/data/chapter1/prize.csv")
5. # Calculate mean, standard deviation and Z-
   scores for the column
6. z_scores = np.abs((prize_csv_df['laureates__share'] - pri
   ze_csv_df['laureates__share'].mean()) / prize_csv_df['lau
   reates__share'].std())
7. # Define a threshold for outliers (e.g., 4)
8. threshold = 2
9. # Display the row index of the outliers
10. print(prize_csv_df.index[z_scores > threshold])
```

Output:

```
1. Index([ 17,  18,  22,  23,  34,  35,  48,  49,  54,  55
   ,
   62,  63,
2.      73,  74,  86,  87,  97,  98, 111, 112, 144, 145,
   146, 147,
3.      168, 169, 180, 181, 183, 184, 215, 216, 242,
   243,
   249, 250,
4.      255, 256, 277, 278, 302, 303, 393, 394, 425,
   426,
   467, 468,
5.      471, 472, 474, 475, 501, 502, 514, 515, 556,
   557,
   563, 564,
6.      607, 608, 635, 636, 645, 646, 683, 684, 760,
   761,
```

```
764, 765,  
7. 1022, 1023],  
8. dtype='int64')
```

The output shows the row index of the outliers in the laureates' share column of the **prize.csv** file. Outliers are values that are unusually high or low compared to the rest of the data. The code uses a z-score to measure how many standard deviations a value is from the mean of the column. A higher z-score means a more extreme value. The code defines a threshold of two, which means that any value with a z-score greater than two is considered an outlier.

Additionally, preparing data, cleaning it, manipulating it, and doing data wrangling includes the following:

- Checking typos and spelling errors. Python provides libraries like **PySpellChecker**, **NLTK**, **TextBlob**, or **Enchant** to check typos and spelling errors.
- Data transformation is a change from one form to another desired form. It involves aggeration, conversion, normalization, and many more, they are covered in detail in *Chapter 2, Exploratory Data Analysis*.
- Handling inconsistencies which involve identifying conflicting information and resolving them. For example, the body temperature is listed as 1400 Celsius which is not correct.
- Standardize format and units of measurements to ensure consistency.
- Further data integrity and validation ensures that data is unchanged, not altered or corrupted. Data validation verifies that the data to be used is correct (use techniques like validation rules, manual review).

Wrangling and manipulation

It means making raw data usable through cleaning, transformation, or other ways. It involves cleaning, organizing, merging, filtering, sorting, aggregating, and reshaping data. Helping you analyze, organize, and improve your data for informed insights and decisions.

The various useful data wrangling and manipulation method in Python are as following:

- **Cleaning:** Some of the methods used to clean the data, along with their syntax, are as follows:

`df.dropna()`: Removing missing values.

`df.fillna()`: Filling missing values.

`df.replace()`: Replacing values.

`df.drop_duplicates()`: Removing duplicate.

`df.drop()`: Removing specific rows **or** columns.

`df.rename()`: Renaming columns.

`df.astype()`: Changing data types.

- **Transformation:** Some of the methods used for data transformation, together with their syntax, are as follows:

`df.apply()`: Applying a function.

`df.groupby()`: Grouping data.

`df.pivot_table()`: Creating pivot tables to summarize.

`df.melt()`: Unpivoting **or** melting data.

`df.sort_values()`: Sorting rows.

`df.join()`, `df.merge()`: Combining data.

- **Aggregation:** Some methods used for data aggregation, together with their syntax, are as follows:

`df.groupby().agg()`: Aggregate data using specified functions.

`df.groupby().size()`, `df.groupby().count()`, `df.groupby().mean()`: Calculating common aggregation metrics.

- **Reshape:** Some methods used for data reshape,

together with their syntax, are as follows:

`df.transpose()`: Transposing rows and columns.

`df.stack()`, `df.unstack()`: Stacking and unstacking.

- **Filtering and subset selection:** Some methods for data filtering and subset selection are as follows:

`df.loc[]`, `df.iloc[]`: Selecting subsets.

`df.query()`: Filtering data using a query.

`df.isin()`: Checking for values in a DataFrame.

`df.nlargest()`, `df.nsmallest()`: Selecting the largest or smallest values.

- **Sorting:** Some methods used for sorting are as follows:

`df.sort_values()`: Sorts a DataFrame by one or more columns.

Ascending or descending order.

`df.sort_index()`: Sorts a DataFrame based on the row index.

`sort()`: Sorts lists in ascending and descending order

- **String manipulation:** Some methods used for string manipulation are as follows:

`str.strip()`, `str.lower()`, `str.upper()`, `str.replace()`

Moreover, adding new columns, variables, statistical modeling, testing and probability distribution, and exploratory data analysis is also part of data wrangling and manipulation, which will be covered in [Chapter 2, Exploratory Data Analysis](#).

Conclusion

Statistics provides a structured framework for understanding and interpreting the world around us. It empowers us to gather, organize, analyze, and interpret information, thereby revealing patterns, testing hypotheses, and informing decisions. In this chapter, we

examined the foundations of data and statistics: from the distinction between qualitative (descriptive) and quantitative (numeric) data to the varying levels of measurement—nominal, ordinal, interval, and ratio. We also considered the scope of analysis in terms of the number of variables involved—whether univariate, bivariate, or multivariate—and recognized that data can originate from diverse sources, including surveys, experiments, and observations.

We explored how careful data collection methods—whether sampling from a larger population or studying an entire group—can significantly affect the quality and applicability of our findings. Ensuring data quality is key, as the validity and reliability of statistical results depend on accurate, complete, and consistent information. Data cleaning addresses errors and inconsistencies, while data wrangling and manipulation techniques help us prepare data for meaningful analysis.

By applying these foundational concepts, we establish a platform for more advanced techniques. In the upcoming Chapter 2, Exploratory data analysis we learn to transform and visualize data in ways that reveal underlying structures, guide analytical decisions, and communicate insights effectively, enabling us to extract even greater value from data.

1 Source: https://scikit-learn.org/stable/datasets/toy_dataset.html#iris-dataset

2 Source: <https://github.com/jdorfman/awesome-json-datasets#nobel-prize>

3 Source: <https://github.com/jdorfman/awesome-json-datasets#nobel-prize>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates,
Offers, Tech happenings around the world, New Release and
Sessions with the Authors:

[**https://discord.bpbonline.com**](https://discord.bpbonline.com)



CHAPTER 2

Exploratory Data Analysis

Introduction

Exploratory Data Analysis (EDA) is the technique of examining, understanding, and summarizing data using various methods. EDA uncovers important insights, features, characteristics, patterns, relationships, and outliers. It also generates hypotheses for the research questions and covers descriptive statistics, a graphical representation of data in a meaningful way, and data exploration in general. In this chapter, we present techniques for data aggregation, transformation, normalization, standardization, binning, grouping, data coding, and encoding, handling missing data and outliers, and the appropriate data visualization methods.

Structure

In this chapter, we will discuss the following topics:

- Exploratory data analysis and its importance
- Data aggregation
- Data normalization, standardization, and transformation
- Data binning, grouping, encoding

- Missing data, detecting and treating outliers
- Visualization and plotting of data

Objectives

By the end of this chapter, readers will learn the techniques to explore the data and to gather meaningful insight to know data well. You will acquire the skills necessary to explore data and gain insights for better understanding. You will learn different data preprocessing method and how to apply them. Further this chapter also explains data encoding, grouping, cleansing, and visualization techniques with Python.

Exploratory data analysis and its importance

EDA is a method of analyzing and summarizing data sets to discover their key characteristics, often using data visualization techniques. EDA helps you better understand the data, find patterns and outliers, test hypotheses, and check assumptions. For example, if you have a data set of home prices and characteristics, you can use EDA to explore the distribution of prices, the relationship between price and characteristics, the effect of location and neighborhood, and so on. You can also use EDA to check for missing values, outliers, or errors in the data. In data science and analytics, EDA helps prepare data for further analysis and modeling. It can help select the appropriate statistical methods or machine learning algorithms for the data, validate the results, and communicate the findings.

Python is a popular programming language for EDA, as it has many libraries and tools that support data manipulation, visualization, and computation. Some of the commonly used libraries for EDA in Python are **pandas**, **NumPy**, **Matplotlib**, **Seaborn**, **Statsmodels**, **Scipy** and **Scikit-learn**. These libraries provide functions and methods for

reading, cleaning, transforming, exploring, and visualizing data in various formats and dimensions.

Data aggregation

Data aggregation in statistics involves summarizing numerical data using statistical measures like mean, median, mode, standard deviation, or percentile. This approach helps detect irregularities and outliers, and enables effective analysis. For example, to determine the average height of students in a class, their individual heights can be aggregated using the mean function, resulting in a single value representing the central tendency of the data. To evaluate the extent of variation in student heights, utilize the standard deviation function to gather data, which will indicate how spread out the data is from the average. The practice of data aggregation in statistics can simplify and aid in comprehending large data sets.

Mean

The mean is a statistical measure used to determine the average value of a set of numbers. To obtain the mean, add all numbers and divide the sum by the number of values. For example, if you have five test scores: 80, 90, 70, 60, and 100, the mean will be as follows:

$$\text{Mean} = (80 + 90 + 70 + 60 + 100) / 5$$

The average score will be the typical score for this series of tests.

Tutorial 2.1: An example to compute the mean from a list of numbers, is as follows:

1. *# Define a list of test scores*
2. `test_scores = [80, 90, 70, 60, 100]`
3. *# Calculate the sum of the test scores*
4. `total = sum(test_scores)`

```
5. # Calculate the number of test scores
6. count = len(test_scores)
7. # Calculate the mean by dividing the sum by the count
8. mean = total / count
9. # Print the mean
10. print("The mean is", mean)
```

The Python **sum()** function takes a list of numbers and returns their sum. For instance, **sum([1, 2, 3])** equals 6. On the other hand, the **len()** function calculates the number of elements in a sequence like a string, a list, or a tuple. For example, **len("hello")** returns 5.

Output:

```
1. The mean is 80.0
```

Median

Median determines the middle value of a data set by locating the value positioned at the center when the data is arranged from smallest to largest. When there is an even number of data points, the median is calculated as the average of the two middle values. For example, among test scores: 75, 80, 85, 90, 95. To determine the median, we must sort the data and locate the middle value. In this case the middle value is 85 thus, the median is 85. If we add another score of 100 to the dataset, we now have six data points: 75, 80, 85, 90, 95, 100. Therefore, the median is the average of the two middle values 85 and 90. The average of the two values: $(85 + 90) / 2 = 87.5$. Hence, the median is 87.5.

Tutorial 2.2: An example to compute the median is as follows:

```
1. # Define the dataset as a list
2. data = [75, 80, 85, 90, 95, 100]
3. # Calculate the number of data points
```

```
4. num_data_points = len(data)
5. # Sort the data in ascending order
6. data.sort()
7. # Check if the number of data points is odd
8. if num_data_points % 2 == 1:
9.     # If odd, find the middle value (median)
10.     median = data[num_data_points // 2]
11. else:
12.     # If even, calculate the average of the two middle values
13.     middle1 = data[num_data_points // 2 - 1]
14.     middle2 = data[num_data_points // 2]
15.     median = (middle1 + middle2) / 2
16. # Print the calculated median
17. print("The median is:", median)
```

Output:

```
1. The median is: 87.5
```

The median is a useful tool for summarizing data that is skewed or has outliers. It is more reliable than the mean, which can be impacted by extreme values. Furthermore, the median separates data into two equal quartiles.

Mode

Mode represents the value that appears most frequently in a given data set. For example, consider a set of shoe sizes that is, 6, 7, 7, 8, 8, 8, 9, 10. To find the mode, count how many times each value appears and identify the value that occurs most frequently. The mode is the most common value. In this case, the mode is 8 since it appears three times, more than any other value.

Tutorial 2.3: An example to compute the mode, is as follows:

```
1. # Define the dataset as a list
2. shoe_sizes = [6, 7, 7, 8, 8, 8, 9, 10]
3. # Create an empty dictionary to store the count of each value
4. size_counts = {}
5. # Iterate through the dataset to count occurrences
6. for size in shoe_sizes:
7.     if size in size_counts:
8.         size_counts[size] += 1
9.     else:
10.        size_counts[size] = 1
11. # Find the mode by finding the key with the maximum value in the dictionary
12. mode = max(size_counts, key=size_counts.get)
13. # Print the mode
14. print("The mode is:", mode)
```

max() used in *tutorial 2.3* is a Python function that returns the highest value from an iterable such as a list or dictionary. In this instance, it retrieves the key (**shoe_sizes**) with the highest count in the **size_counts** dictionary. The **.get()** method is used in a dictionary as a key function for **max()**. It retrieves the value associated with a key. In this case, **size_counts.get** retrieves the count associated with each shoe size key. Then **max()** uses this information to determine which key (**shoe_sizes**) has the highest count, indicating the mode.

Output:

```
1. The mode is: 8
```

Variance

Variance measures the deviation of data values from their average in a dataset. It is calculated by averaging the

squared differences between each value and the mean. A high variance suggests that data is spread out from the mean, while a low variance suggests that data is tightly grouped around the mean. For example, suppose we have two sets of test scores: **A = [90, 92, 94, 96, 98]** and **B = [70, 80, 90, 100, 130]**. The mean of both sets is 94, but the variance of A is 8 and B is 424. Lower variance in A means the scores in A are more consistent and closer to the mean than the scores in B. We can use the **var()** function from the **numpy** module to see the variance in Python.

Tutorial 2.4: An example to compute the variance is as follows:

```
1. import numpy as np
2. # Define two sets of test scores
3. A = [90, 92, 94, 96, 98]
4. B = [70, 80, 90, 100, 130]
5. # Calculate and print the mean of A and B
6. print("The mean of A is", sum(A)/len(A))
7. print("The mean of B is", sum(B)/len(B))
8. # Calculate and print the variance of A and B
9. var_A = np.var(A)
10. var_B = np.var(B)
11. print("The variance of A is", var_A)
12. print("The variance of B is", var_B)
```

To compute the variance in a pandas data frame, one way is to use the **describe()** method, which returns a summary of the descriptive statistics for each column, including the variance. For example, if we have a data frame named **df**, we can use **df.describe()** to see the variance of each column. Another way is to use the **apply()** method, which applies a function to each column or row of a data frame. For example, if we want to compute the variance of each row, we can use **df.apply(np.var, axis=1)**, where **np.var** is

the NumPy function for variance and **axis=1** means that the function is applied along the row axis.

Output:

1. The mean of A is 94.0
2. The mean of B is 94.0
3. The variance of A is 8.0
4. The variance of B is 424.0

Standard deviation

Standard deviation is a measure of how much the values in a data set vary from the mean. It is calculated by taking the square root of the variance. A high standard deviation means that the data is spread out, while a low standard deviation means that the data is concentrated around the mean. For example, suppose we have two sets of test scores: **A = [90, 92, 94, 96, 98]** and **B = [70, 80, 90, 100, 110]**. The mean of both sets is 94, but the standard deviation of A is about 2.83 and the standard deviation of B is about 14.14. This means that the scores in A are more consistent and closer to the mean than the scores in B. To find the standard deviation in Python, we can use the **std()** function from the **numpy** module.

Tutorial 2.5: An example to compute the standard deviation is as follows:

1. *# Import numpy module*
2. **import** numpy **as** np
3. *# Define two sets of test scores*
4. A = [90, 92, 94, 96, 98]
5. B = [70, 80, 90, 100, 110]
6. *# Calculate and print the standard deviation of A and B*
7. std_A = np.std(A)
8. std_B = np.std(B)
9. print("The standard deviation of A is", std_A)

```
10. print("The standard deviation of B is", std_B)
```

Output:

1. The standard deviation of A is 2.82
2. The standard deviation of B is 14.14

Quantiles

A quantile is a value that separates a data set into an equal number of groups, typically four (quartiles), five (quintiles), or ten (deciles). The groups are formed by ranking the data set in ascending order, ensuring that each group contains the same number of values. Quantiles are useful for summarizing data distribution and comparing different data sets.

For example, let us consider a set of 15 heights in centimeters: **[150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178]**. To calculate the quartiles (a specific subset of quantiles) for this dataset, divide it into four equally sized groups. Q1, the first quartile, represents the median of the lower half of the data, which is 158. Q2, the second quartile, corresponds to the median of the entire data set, which is 164. Q3, the third quartile, represents the median of the upper half of the data, which is 170. The data is split into four clear groups by the quartiles: **[150, 152, 154, 156]**, **[158, 160, 162]**, **[164, 166, 168]**, and **[170, 172, 174, 176, 178]**. This separation facilitates understanding and comparison of distinct segments of the data's distribution.

Tutorial 2.6: An example to compute the quantiles is as follows:

1. *# Import numpy module*
2. **import** numpy **as** np
3. *# Define a data set of heights in centimeters*
4. heights = [150 ,152 ,154 ,156 ,158 ,160 ,162 ,164 ,166 ,
168 ,170 ,172 ,174 ,176 ,178]

```
5. # Calculate and print the quartiles of the heights
6. Q1 = np.quantile(heights ,0.25)
7. Q2 = np.quantile(heights ,0.5)
8. Q3 = np.quantile(heights ,0.75)
9. print("The first quartile is", Q1)
10. print("The second quartile is", Q2)
11. print("The third quartile is", Q3)
```

Output:

1. The first quartile is 157.0
2. The second quartile is 164.0
3. The third quartile is 171.0

Tutorial 2.7: An example to compute mean, median, mode, standard deviation, maximum, minimum value in pandas data frame.

The mean, median, mode, variance, maximum and minimum value in data frame can be computed easily with **mean()**, **median()**, **mode()**, **var()**, **max()**, **min()** respectively, as follows:

```
1. # Import the pandas library
2. import pandas as pd
3. # Import display function
4. from IPython.display import display
5. # Load the diabetes data from a csv file
6. diabetes_df = pd.read_csv(
7.     "/workspaces/ImplementingStatisticsWithPython/data/
   /chapter1/diabetes.csv")
8. # Print the mean of each column
9. print(f'Mean: \n {diabetes_df.mean()}')
10. # Print the median of each column
11. print(f'Median: \n {diabetes_df.median()}')
12. # Print the mode of each column
```

```
13. print(f'Mode: \n {diabetes_df.mode()}')
14. # Print the variance of each column
15. print(f'Varience: \n {diabetes_df.var()}')
16. # Print the standard deviation of each column
17. print(f'Standard Deviation: \n{diabetes_df.std()}')
18. # Print the maximum value of each column
19. print(f'Maximum: \n {diabetes_df.max()}')
20. # Print the minimum value of each column
21. print(f'Minimum: \n {diabetes_df.min()}')
```

Tutorial 2.8: An example to compute mean, median, mode, standard deviation, maximum, minimum value in NumPy array, is as follows:

```
1. # Import the numpy and statistics libraries
2. import numpy as np
3. import statistics as st
4. # Create a numpy array with some data
5. data = np.array([12, 15, 20, 25, 30, 30, 35, 40, 45, 50])
6. # Calculate the mean of the data using numpy
7. mean = np.mean(data)
8. # Calculate the median of the data using numpy
9. median = np.median(data)
10. # Calculate the mode of the data using statistics
11. mode_result = st.mode(data)
12. # Calculate the standard deviation of the data using numpy
13. std_dev = np.std(data)
14. # Find the maximum value of the data using numpy
15. maximum = np.max(data)
16. # Find the minimum value of the data using numpy
17. minimum = np.min(data)
18. # Print the results to the console
```

```
19. print("Mean:", mean)
20. print("Median:", median)
21. print("Mode:", mode_result)
22. print("Standard Deviation:", std_dev)
23. print("Maximum:", maximum)
24. print("Minimum:", minimum)
```

Output:

```
1. Mean: 30.2
2. Median: 30.0
3. Mode: 30
4. Standard Deviation: 11.93
5. Maximum: 50
6. Minimum: 12
```

Tutorial 2.9: An example to compute variance, quantiles, and percentiles using **var()** and **quantile** from diabetes dataset data frame, and also **describe()** to describe the data frame, is as follows:

```
1. import pandas as pd
2. from IPython.display import display
3. # Load the diabetes data from a csv file
4. diabetes_df = pd.read_csv(
5.     "/workspaces/ImplementingStatisticsWithPython/data/
   /chapter1/diabetes.csv")
6. # Calculate the variance of each column using pandas
7. variance = diabetes_df.var()
8. # Calculate the quantiles (25th, 50th, and 75th percentil
   es) of each column using pandas
9. quantiles = diabetes_df.quantile([0.25, 0.5, 0.75])
10. # Calculate the percentiles (90th and 95th percentiles) of
    each column using pandas
11. percentiles = diabetes_df.quantile([0.9, 0.95])
12. # Display the results using the display function
```

```
13. display("Variance:", variance)
14. display("Quantiles:", quantiles)
15. display("Percentiles:", percentiles)
```

This will calculate the variance, quantile and percentile of each column in the **diabetes_df** data frame.

Data normalization, standardization, and transformation

Data normalization, standardization, and transformation are methods for preparing data for analysis. They ensure that the data is consistent, comparable, and appropriate for various analytical techniques. **Data normalization** rescales feature values to a range between zero and one, helping to mitigate the impact of outliers and different scales on the data. For instance, if one feature ranges from 0 to 100, while another ranges from 0 to 10,000, normalizing them can enhance comparability.

Standardizing data is achieved by subtracting the mean and dividing by the standard deviation of a feature. This results in a more normal distribution of data centered around zero. For example, if one feature has a mean of 50 and a standard deviation of 10, standardizing it will achieve a mean of 0 and a standard deviation of 1.

Data transformation involves using a mathematical function to alter the shape or distribution of a feature and make the data more linear or symmetrical. For instance, if a feature has an uneven distribution, applying a logarithmic or square root transformation can balance it. The order of these techniques relies on the data's purpose and type. It is generally recommended to perform data transformation prior to data standardization and then data normalization. Nevertheless, specific methods may call for varying or no preprocessing. Therefore, understanding the requirements and assumptions of each technique is crucial before

implementation.

Data normalization

Standardizing and organizing data entries through normalization improves their suitability for analysis and comparison, resulting in higher quality data. Additionally, reducing the impact of outliers enhances algorithm performance, increases data interpretability, and uncovers underlying patterns among variables.

Normalization of NumPy array

We can use the **numpy.min** and **numpy.max** functions to find the minimum and maximum values of an array, and then use the formula $x_{norm} = (x_i - x_{min}) / (x_{max} - x_{min})$ to normalize each value.

Tutorial 2.10: An example to show normalization of NumPy array, is as follows:

```
1. #import numpy
2. import numpy as np
3. #create a sample dataset
4. data = np.array([10, 15, 20, 25, 30])
5. #find the minimum and maximum values of the data
6. xmin = np.min(data)
7. xmax = np.max(data)
8. #normalize the data using the formula
9. normalized_data = (data - xmin) / (xmax - xmin)
10. #print the normalized data
11. print(normalized_data)
```

Array data before normalization, is as follows:

```
1. [10 15 20 25 30]
```

Array data after normalization, is as follows:

```
1. [0.  0.25 0.5  0.75 1.  ]
```


Tutorial 2.11: An example to show normalization of the 2-Dimensional NumPy array using **MinMaxScaler**, is as follows:

Following is an easy example of data normalization in Python using the scikit-learn library. **MinMaxScaler** is a technique to rescale the values of a feature to a specified range, typically between zero and one. This can help to reduce the effect of outliers and different scales on the data. **scaler.fit_transform()** is a method that combines two steps: fit and transform. The fit step computes the minimum and maximum values of each feature in the data. The transform step applies the formula $x_{norm} = (x_i - x_{min}) / (x_{max} - x_{min})$ to each value in the data, where x_{min} and x_{max} are the minimum and maximum values of the feature.

Code:

```
1. #import numpy library for working with arrays
2. import numpy as np
3. #import MinMaxScaler class from the preprocessing module of scikit-learn library for data normalization
4. from sklearn.preprocessing import MinMaxScaler
5. #create a structured data as a 2D array with two features: x and y
6. structured_data = np.array([[100, 200], [300, 400], [500, 600]])
7. #print the normalized structured data as a numpy array
8. print("Original Data:")
9. print(structured_data)
10. #create an instance of MinMaxScaler object that can normalize the data
11. scaler = MinMaxScaler()
12. #fit the scaler to the data and transform the data to a range between 0 and 1
13. normalized_structured = scaler.fit_transform(structured
```

```
_data)
```

```
14. #print the normalized structured data as a numpy array
```

```
15. print("Normalized Data:")
```

```
16. print(normalized_structured)
```

2-Dimensional array data before normalization is as follows:

```
1. [[100 200]
```

```
2. [300 400]
```

```
3. [500 600]]
```

2-Dimensional array data after normalization is as follows:

```
1. [[0. 0.]
```

```
2. [0.5 0.5]
```

```
3. [1. 1.]]
```

One potential problem when using **MinMaxScaler** for normalization is its sensitivity to outliers and extreme values. This can distort the scaling and limit the range of transformed features, potentially impacting the performance and accuracy of machine learning algorithms that rely on feature scale or distribution. A better alternative could be using the **Standard Scaler** or the **Robust Scaler**.

Standard Scaler rescales the data to achieve a mean of zero and a standard deviation of one, which improves optimization or distance-based algorithms. Although outliers can still impact the data, there is no guarantee of a restricted range for the transformed features. Robust Scaler is robust against extreme values and outliers, as it eliminates the median and rescales the data based on the **Interquartile Range (IQR)**. However, there is no assurance of a bounded span for the transformed features.

Tutorial 2.12. An example to show normalization of the 2-Dimensional array, is as follows:

```
1. #import the preprocessing module from the scikit-learn library
```

```
2. from sklearn import preprocessing
3. #create a sample dataset with two features: x and y
4. data = [[10, 2000], [15, 3000], [20, 4000], [25, 5000]]
5. #initialize a MinMaxScaler object that can normalize the
   data
6. scaler = preprocessing.MinMaxScaler()
7. #fit the scaler to the data and transform the data to a ra
   nge between 0 and 1
8. normalized_data = scaler.fit_transform(data)
9. #print the normalized data as a numpy array
10. print(normalized_data)
```

The data before normalization, is as follows:

```
1. [[10, 2000], [15, 3000], [20, 4000], [25, 5000]]
```

The data after normalization represented between zero and one, is as follows:

```
1. [[0.      0.      ]
2. [0.33333333 0.33333333]
3. [0.66666667 0.66666667]
4. [1.      1.      ]]
```

Normalization of pandas data frame

To normalize a pandas data frame we can use the min-max scaling technique. Min-max scaling is a normalization method that rescales data to fit between zero and one. It is beneficial for variables with predetermined ranges or algorithms that are sensitive to scale. An example of min-max scaling can be seen by normalizing test scores that range from 0 to 100.

Following are some sample scores to consider:

Name	Score
Alice	80

Bob	60
Carol	90
David	40

Table 2.1: Scores of students in a class

To apply min-max scaling, we use the following formula:

$$\text{normalized value} = (\text{original value} - \text{minimum value}) / (\text{maximum value} - \text{minimum value})$$

The minimum value is 0 and the maximum value is 100, so we can simplify the formula as follows:

$$\text{normalized value} = \text{original value} / 100$$

Using this formula, we can calculate the normalized scores as follows:

Name	Score	Normalized score
Alice	80	0.8
Bob	60	0.6
Carol	90	0.9
David	40	0.4

Table 2.2: Normalized scores of students in a class

The normalized scores are now between zero and one, and they preserve the relative order and distance of the original scores.

Tutorial 2.13. An example to show normalization of data frame using **pandas** and **sklearn** library, is as follows:

1. `#import pandas and sklearn`
2. `import pandas as pd`
3. `from sklearn.preprocessing import MinMaxScaler`
4. `#create a sample dataframe with three columns: age, height, and weight`
5. `df = pd.DataFrame({`

```

6.     'age': [25, 35, 45, 55],
7.     'height': [160, 170, 180, 190],
8.     'weight': [60, 70, 80, 90]
9. })
10. #print the original dataframe
11. print("Original dataframe:")
12. print(df)
13. #create a MinMaxScaler object
14. scaler = MinMaxScaler()
15. #fit and transform the dataframe using the scaler
16. normalized_df = scaler.fit_transform(df)
17. #convert the normalized array into a dataframe
18. normalized_df = pd.DataFrame(normalized_df, columns
    =df.columns)
19. #print the normalized dataframe
20. print("Normalized dataframe:")
21. print(normalized_df)

```

The original data frame, is as follows:

```

1.   age  height  weight
2.  0   25    160     60
3.  1   35    170     70
4.  2   45    180     80
5.  3   55    190     90

```

The normalized data frame, is as follows:

```

1.   age  height  weight
2.  0  0.000000  0.000000  0.000000
3.  1  0.333333  0.333333  0.333333
4.  2  0.666667  0.666667  0.666667
5.  3  1.000000  1.000000  1.000000

```

Tutorial 2.14. An example to read a Comma Separated File (CSV) and normalize the selected column in it using **pandas**

and **sklearn** library is as follows, using the **diabetes.csv** data:

```
1. # import MinMaxScaler class from the preprocessing module of scikit-learn library for data normalization
2. from sklearn.preprocessing import MinMaxScaler
3. import pandas as pd
4. # import IPython.display for displaying the dataframe
5. from IPython.display import display
6. # read the csv file from the directory and store it as a dataframe
7. diabetes_df = pd.read_csv(
8.     "/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
9. # specify the columns to normalize, which are all the numerical features in the dataframe
10. columns_to_normalize = ['Pregnancies', 'Glucose', 'Blood Pressure',
11.     'SkinThickness', 'Insulin', 'BMI', 'Diabetes PedigreeFunction', 'Age', 'Outcome']
12. # display the unnormalized dataframe
13. display(diabetes_df[columns_to_normalize].head(4))
14. # create an instance of MinMaxScaler object that can normalize the data
15. scaler = MinMaxScaler()
16. # fit and transform the dataframe using the scaler and assign the normalized values to the same columns
17. diabetes_df[columns_to_normalize] = scaler.fit_transform(
18.     diabetes_df[columns_to_normalize])
19. # print a message to indicate the normalized structured data
20. print("Normalized Structured Data:")
```

```
21. # display the normalized dataframe
```

```
22. display(diabetes_df.head(4))
```

The output of *Tutorial 2.14* will be a data frame with normalized values in the selected columns.

Data standardization

Data standardization is a type of data transformation that adjusts data to have a mean of zero and a standard deviation of one. It helps compare variables with different scales or units and is necessary for algorithms like **Principal Component Analysis (PCA)**, **Linear Discriminant Analysis (LDA)**, or k-means clustering that require standardized data. By standardizing values, we can measure how far each value is from the mean in terms of standard deviations. This can help us identify outliers, perform hypothesis tests, or apply machine learning algorithms that require standardized data. There are different ways to standardize data like min-max normalization described in normalization of data frames, but the z-score formula remains the most widely used. This formula adjusts each value in a dataset by subtracting the mean and dividing it by the standard deviation. The formula is as follows:

$$z = (x - \mu) / \sigma$$

Where x represents the original value, μ represents the mean, and σ represents the standard deviation.

Suppose, we have a dataset of two variables: height (in centimeters) and weight (in kilograms) of five people:

Height	Weight
160	50
175	70
180	80

168	60

Table 2.3: Height and weight of peoples

The mean height is 169 cm and the standard deviation is 7.6 cm. The mean weight is 62.4 kg and the standard deviation is 11.6 kg. To standardize the data, we use the formula as follows:

$$z = (x - \mu) / \sigma$$

where x is the original value, μ is the mean, and σ is the standard deviation. Applying this formula to each value in the dataset, we get the following standardized values:

Height (z-score)	Weight (z-score)
-1.18	-1.07
0.79	0.66
1.45	1.52
-0.13	-0.21

Table 2.4: Standardized height and weight

Now, the two variables have an average of zero and a standard deviation of one, and they are measured on the same scale. The standardized values reflect the extent to which each observation deviates from the mean in terms of standard deviations.

Standardization of NumPy array

Tutorial 2.15. An example to show standardization of height and weight as a NumPy array, is as follows:

1. *# Import numpy library for numerical calculations*
2. **import** numpy **as** np
3. *# Define the data as numpy arrays*
4. height = np.array([160, 175, 180, 168, 162])
5. weight = np.array([50, 70, 80, 60, 52])


```

6. # Calculate the mean and standard deviation of each variable
7. height_mean = np.mean(height)
8. height_std = np.std(height)
9. weight_mean = np.mean(weight)
10. weight_std = np.std(weight)
11. # Define the z-score formula as a function
12. def z_score(x, mean, std):
13.     return (x - mean) / std
14. # Apply the z-score formula to each value in the data
15. height_z = z_score(height, height_mean, height_std)
16. weight_z = z_score(weight, weight_mean, weight_std)
17. # Print the standardized values
18. print("Height (z-score):", height_z)
19. print("Weight (z-score):", weight_z)

```

Output:

```

1. Height (z-score): [-1.18421053  0.78947368  1.44736842 -0.13157895 -0.92105263]
2. Weight (z-score): [-1.06904497  0.65465367  1.51887505 -0.20655562 -0.89792798]

```

Standardization of data frame

Tutorial 2.16. An example to show standardization of height and weight as a data frame, is as follows:

```

1. # Import pandas library for data manipulation
2. import pandas as pd
3. # Define the original data as a pandas dataframe
4. data = pd.DataFrame({"Height": [160, 175, 180, 168, 162],
    "Weight": [50, 70, 80, 60, 52]})

```

```

5. # Calculate the mean and standard deviation of each column
6. data_mean = data.mean()
7. data_std = data.std()
8. # Define the z-score formula as a function
9. def z_score(column):
10.     mean = column.mean()
11.     std_dev = column.std()
12.     standardized_column = (column - mean) / std_dev
13.     return standardized_column
14. # Apply the z-score formula to each column in the dataframe
15. data_z = data.apply(z_score)
16. # Print the standardized dataframe
17. print("Data (z-score):", data_z)

```

Output:

```

1. Data (z-score):   Height   Weight
2. 0 -1.060660 -0.984003
3. 1  0.707107  0.603099
4. 2  1.296362  1.396649
5. 3 -0.117851 -0.190452
6. 4 -0.824958 -0.825293

```

Data transformation

Data transformation is essential as it satisfies the requirements for particular statistical tests, enhances data interpretation, and improves the visual representation of charts. For example, consider a dataset that includes the heights of 100 students measured in centimeters. If the distribution of data is positively skewed (more students are shorter than taller), assumptions like normality and equal variances must be satisfied before conducting a t-test. A t-

test (a statistical test used to compare the means of two groups) on the average height of male and female students may produce inaccurate results if skewness violates these assumptions.

To mitigate this problem, transform the height data by taking the square root or logarithm of each measurement. Doing so will improve consistency and accuracy. Perform a t-test on the transformed data to compute the average height difference between male and female students with greater accuracy. Use the inverse function to revert the transformed data back to its original scale. For example, if the transformation involved the square root, then square the result to express centimeters. Another reason to use data transformation is to improve data visualization and understanding. For example, suppose you have a dataset of the annual income of 1000 people in US dollars that is skewed to the right, indicating that more participants are in the lower-income bracket. If you want to create a histogram that shows income distribution, you will see that most of the data is concentrated in a few bins on the left, while some outliers exist on the right side. For improved clarity in identifying the distribution pattern and range, apply a transformation to the income data by taking the logarithm of each value. This distributes the data evenly across bins and minimizes the effect of outliers. After that, plot a histogram of the log-transformed income to show the income fluctuations among individuals.

Tutorial 2.17: An example to show the data transformation of the annual income of 1000 people in US dollars, which is a skewed data set, is as follows:

1. *# Import the libraries*
2. `import numpy as np`
3. `import matplotlib.pyplot as plt`
4. *# Generate some random data for the annual income of 1000 people in US dollars*

```
5. np.random.seed(42) # Set the seed for reproducibility
6. income = np.random.lognormal(mean=10, sigma=1, size
    =1000) # Generate 1000 incomes from a lognormal distri
    bution with mean 10 and standard deviation 1
7. income = income.round(2) # Round the incomes to two
    decimal places
8. # Plot a histogram of the original income
9. plt.hist(income, bins=20)
10. plt.xlabel("Income (USD)")
11. plt.ylabel("Frequency")
12. plt.title("Histogram of Income")
13. plt.show()
```

Suppose the initial actual distribution of annual income of 1000 people in US dollars as shown in [Figure 2.1](#):

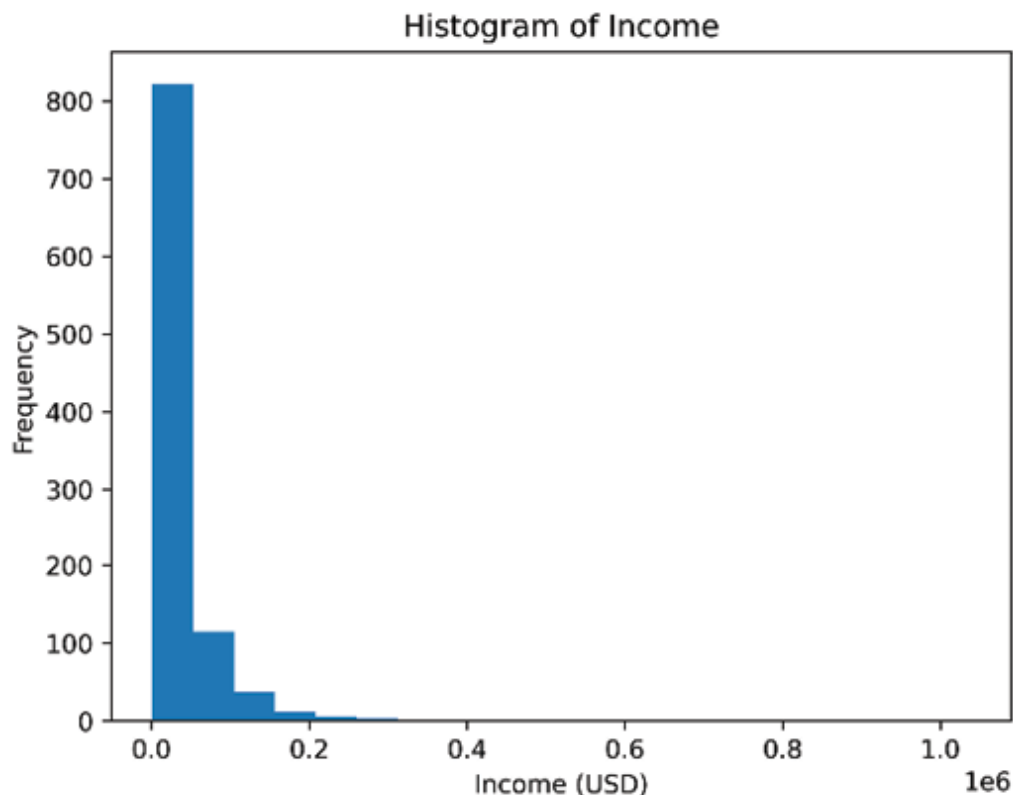


Figure 2.1: Distribution of annual income of 1000 people in US dollars

Now, let us apply the logarithmic transformation to the income:

```
1. # Apply a logarithm transformation to the income
2. log_income = np.log10(income) # Take the base 10 logarithm of each income value
3. # Plot a histogram of the transformed income
4. plt.hist(log_income, bins=20)
5. plt.xlabel("Logarithm of Income")
6. plt.ylabel("Frequency")
7. plt.title("Histogram of Logarithm of Income")
8. # Set the DPI to 600
9. plt.savefig('data_transformation2.png', dpi=600)
10. # Show the plot (optional)
11. plt.show()
```

The **log10()** function in the above code takes the base 10 logarithm of each income value. This means that it converts the income values from a linear scale to a logarithmic scale, where each unit increase on the x-axis corresponds to a 10-fold increase on the original scale. For example, if the income value is 100, the **log10** value is 2, and if the income value is 1000, the **log10** value is 3.

The **log10** function is useful for data transformation because it can reduce the skewness and variability of the data, and make it easier to compare values that differ by orders of magnitude.

Now, let us plot the histogram of income after logarithmic transformation as follows:

```
1. # Label the x-axis with the original values by using 10^x as tick marks
2. plt.hist(log_income, bins=20)
3. plt.xlabel("Income (USD)")
4. plt.ylabel("Frequency")
```

```
5. plt.title("Histogram of Logarithm of Income")
6. plt.xticks(np.arange(1, 7), ["$10", "$100", "$1K", "$10K",
    "$100K", "$1M"])
7. plt.show()
```

The histogram of logarithm of income with original values is plotted as shown in [Figure 2.2](#):

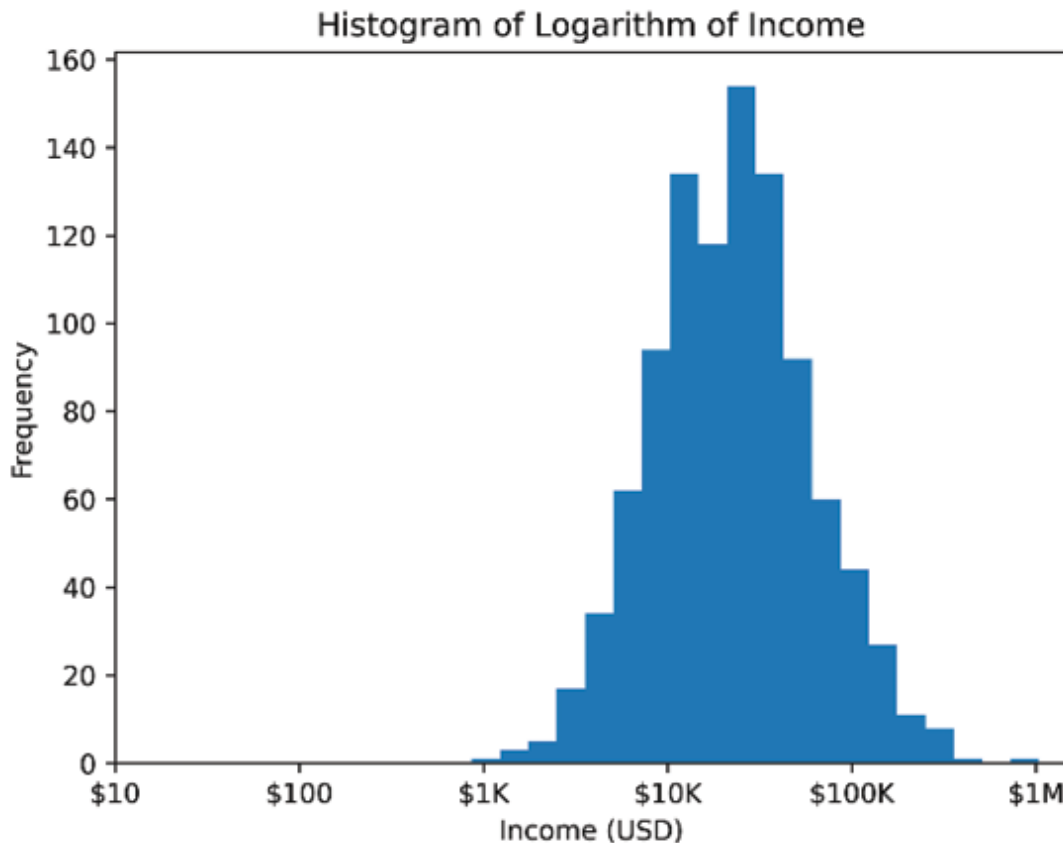


Figure 2.2: Logarithmic distribution of annual income of 1000 people in US dollars

As you can see, the data transformation made the data more evenly distributed across bins, and reduced the effect of outliers. The histogram of the log-transformed income showed a clearer picture of how income varies among people.

In unstructured data like text, normalization may involve natural language processing like convert lowercase,

removing punctuation, handling special character like whitespaces and many more. In image or audio, it may involve rescaling pixel values, extracting features.

Tutorial 2.18: An example to convert lowercase, removing punctuation, handling special character like whitespaces in unstructured text data, is as follows:

```
1. # Import the re module, which provides regular expressions on operations
2. import re
3.
4. # Define a function named normalize_text that takes a text as an argument
5. def normalize_text(text):
6.     # Convert all the characters in the text to lowercase
7.     text = text.lower()
8.     # Remove any punctuation marks (such as . , ! ?) from the text using a regular expression
9.     text = re.sub(r'^\w\s', '', text)
10.    # Remove any extra whitespace (such as tabs, newlines, or multiple spaces) from the text using a regular expression
11.    text = re.sub(r'\s+', ' ', text).strip()
12.    # Return the normalized text as the output of the function
13.    return text
14.
15. # Create a sample unstructured text data as a string
16. unstructured_text = "This is an a text for book Implementing Stat with Python, with! various punctuation marks..."
17. # Call the normalize_text function on the unstructured text and assign the result to a variable named normalized
```

```
    _text
18. normalized_text = normalize_text(unstructured_text)
19. # Print the original and normalized texts to compare the
    m
20. print("Original Text:", unstructured_text)
21. print("Normalized Text:", normalized_text)
```

Output:

1. Original Text: This is an a text for book Implementing Stat with Python, with! various punctuation marks...
2. Normalized Text: this is an a text for book implementing stat with python with various punctuation marks

Data binning, grouping, encoding

Data binning, grouping, and encoding are common data preprocessing and feature engineering techniques. They transform the original data into a format suitable for modeling or analysis.

Data binning

Data binning groups continuous or discrete values into a smaller number of bins or intervals. For example, if you have data on the ages of 100 people, you may group them into five bins: [0-20), [20-40), [40-60), [60-80), and [80-100], where [0-20) includes values greater than or equal to 0 and less than 20, [80-100] includes values greater than or equal to 80 and less than or equal to 100. Each bin represents a range of values, and the number of cases in each bin can be counted or visualized. Data binning reduces noise, outliers, and skewness in the data, making it easier to view distribution and trends.

Tutorial 2.19: A simple implementation of data binning for grouping the ages of 100 people into five bins: [0-20), [20-40), [40-60), [60-80), and [80-100] is as follows:


```

1. # Import the libraries
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. # Generate some random data for the ages of 100 people
6. np.random.seed(42) # Set the seed for reproducibility
7. ages = np.random.randint(low=0, high=101, size=100)
   # Generate 100 ages between 0 and 100
8. # Create a pandas dataframe with the ages
9. df = pd.DataFrame({"Age": ages}) # Create a dataframe
   with one column: Age
10. # Define the bins and labels for the age groups
11. bins = [0, 20, 40, 60, 80, 100] # Define the bin edges
12. labels = ["[0-20)", "[20-40)", "[40-60)", "[60-80)", "[80-
   100)"] # Define the bin labels
13. # Apply data binning to the ages using the pd.cut function
14. df["Age Group"] = pd.cut(df["Age"], bins=bins, labels=labels,
   right=False) # Create a new column with the age groups
15. # Print the first 10 rows of the dataframe
16. print(df.head(10))

```

Output:

```

1.   Age Age Group
2.  0   51  [40-60)
3.  1   92  [80-100]
4.  2   14  [0-20)
5.  3   71  [60-80)
6.  4   60  [60-80)
7.  5   20  [20-40)
8.  6   82  [80-100]

```

```
9. 7 86 [80-100]
10. 8 74 [60-80)
11. 9 74 [60-80)
```

Tutorial 2.20: An example to apply binning on diabetes dataset by grouping the ages of all the people in dataset into three bins: [< 30], [$30-60$], [$60-100$], is as follows:

```
1. import pandas as pd
2. # Read the json file from the direcotory
3. diabetes_df = pd.read_csv(
4.     "/workspaces/ImplementingStatisticsWithPython/data
   /chapter1/diabetes.csv")
5. # Define the bin intervals
6. bin_edges = [0, 30, 60, 100]
7. # Use cut to create a new column with bin labels
8. diabetes_df['Age_Group'] = pd.cut(diabetes_df['Age'],
   bins=bin_edges, labels=[
9.     '<30', '30-60', '60-100'])
10. # Count the number of people in each age group
11. age_group_counts = diabetes_df['Age_Group'].
   value_counts().sort_index()
12. # View new DataFrame with the new bin(categories) colu
   mns
13. diabetes_df
```

The output is a new data frame with **Age_Group** column consisting appropriate bin label.

Tutorial 2.21: An example to apply binning on NumPy array data by grouping the scores of students in exam into five bins based on the scores obtained: [< 60], [$60-69$], [$70-79$], [$80-89$] , [$90+$], is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Create a sample NumPy array of exam scores
```

```
4. scores = np.array([75, 82, 95, 68, 90, 85, 78, 72, 88, 93,
60, 72, 80])
5. # Define the bin intervals
6. bin_edges = [0, 60, 70, 80, 90, 100]
7. # Use histogram to count the number of scores in each bin
8. bin_counts, _ = np.histogram(scores, bins=bin_edges)
9. # Plot a histogram of the binned scores
10. plt.bar(range(len(bin_counts)), bin_counts, align='center')
11. plt.xticks(range(len(bin_edges) - 1), ['<60', '60-69', '70-79', '80-89', '90+'])
12. plt.xlabel('Score Range')
13. plt.ylabel('Number of Scores')
14. plt.title('Distribution of Exam Scores')
15. plt.savefig("data_binning2.jpg",dpi=600)
16. plt.show()
```

Output:

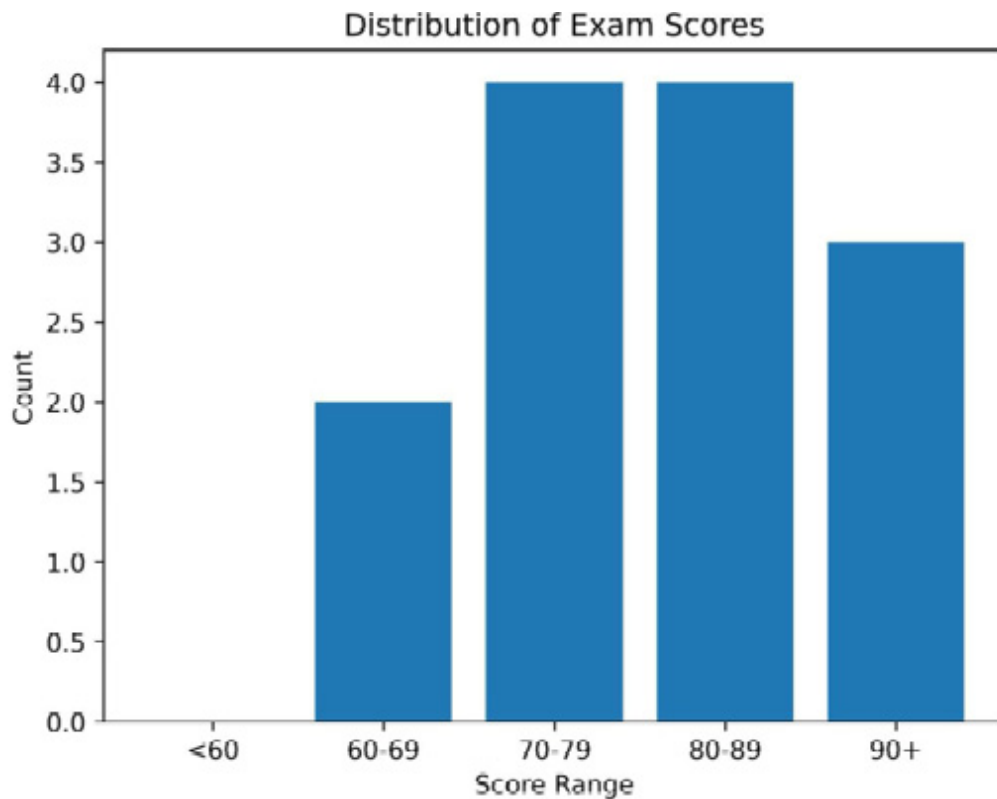


Figure 2.3: Distribution of student's exam scores in five bins

In text files, data binning can be grouping and categorizing of text data based on some criteria. To apply data binning on the text data, keep the following points in mind:

- Determine a criterion for binning. For example: Could be count of sentences in text, word count, sentiment score, topic.
- Read text and calculate the selected criteria for binning. For example: Count number of words in bins.
- Define bins based on range of values for the selected criteria. For example: Defining short, medium, long based on word count of text.
- Assign text files appropriate bin based on calculated value.
- Analyze or summarize the data in the new bins.

Some use cases of binning in text file are grouping text files based on their length, binning based on the sentiment

analysis score, topic binning by performing topic modelling, language binning if text files are in different languages, time-based binning if text files have timestamps.

Tutorial 2.22: An example showing data binning of text files using word counts in the files with three bins: [<26 words] as short [26 and 30 words (inclusive)] as medium, [>30] as long, is as follows:

```
1. # Import the os, glob, and pandas modules
2. import os
3. import glob
4. import pandas as pd
5. # Define the path of the folder that contains the files
6. path = "/workspaces/ImplementingStatisticsWithPython/
  /data/chapter1/TransactionNarrative"
7. files = glob.glob(path + "/*.txt") # Get a list of files that
  match the pattern "/*.txt" in the folder
8. # Display a the information in first file
9. file_one = glob.glob("/workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative/1.txt")
10. for file1 in file_one: # Loop through the file_one list
11.     # To open each file in read mode with utf-8 encoding and assign it to a file object
12.     with open(file1, "r", encoding="utf-8") as f1: # Open each file in read mode with utf-8 encoding and assign it to a file object named f1
13.         print(f1.read()) # Print the content of the file object
14. # Function that takes a file name as an argument and returns the word count of that file
15. def word_count(file): # Define a function named word_count that takes a file name as an argument
16.     # Open the file in read mode
```

```

17. with open(file, "r") as f: # Open the file in read mode
    and assign it to a file object named f
18.     # Read the file content
19.     content = f.read() # Read the content of the file object
    and assign it to a variable named content
20.     # Split the content by whitespace characters
21.     words = content.split() # Split the content by whitespace
    characters and assign it to a variable named words
22.     # Return the length of the words list
23.     return len(words) # Return the length of the words list
    as the output of the function
24. counts = [word_count(file) for file in files] # Use a list
    comprehension to apply the word_count function to each file
    in the files list and assign it to a variable named counts
25. binning_df = pd.DataFrame({"file": files, "count": counts})
    # Create a pandas dataframe with two columns: file and count,
    using the files and counts lists as values
26. binning_df["bin"] = pd.cut(binning_df["count"], bins=[0, 26, 30, 35])
    # Create a new column named bin, using the pd.cut function to
    group the count values into three bins: [0-26), [26-30), and
    [30-35]
27. binning_df["bin"] = pd.cut(binning_df["count"], bins=[0, 26, 30, 35],
    labels=["Short", "Medium", "Long"]) # Replace the bin values
    with labels: Short, Medium, and Long, using the labels argument
    of the pd.cut function
28. binning_df # Display the dataframe

```

Output:

The output shows a sample text file, then, the file names, the number of words in each file, and the assigned bin labels as follows:

1. Date: 2023-08-05

2. Merchant: Bistro Delight
3. Amount: \$42.75
4. Description: Dinner with colleagues - celebrating a successful project launch.
- 5.
6. Thank you for choosing Bistro Delight. Your payment of \$42.75 has been processed.
- 7.
8. file count bin
9. 0 /workspaces/ImplementingStatisticsWithPython/d... 2
5 Short
10. 1 /workspaces/ImplementingStatisticsWithPython/d... 3
0 Medium
11. 2 /workspaces/ImplementingStatisticsWithPython/d... 3
1 Long
12. 3 /workspaces/ImplementingStatisticsWithPython/d... 2
7 Medium
13. 4 /workspaces/ImplementingStatisticsWithPython/d... 3
3 Long

In unstructured data, the data binning can be used for text categorization and modelling of text data, color quantization and feature extraction on image data, audio segmentation and feature extraction on audio data.

Data grouping

Data grouping aggregates data by criteria or categories. For example, if sales data exists for different products or market regions, grouping by product type or region can be beneficial. Each group represents a subset of data that shares some common attribute, allowing for comparison of summary statistics or measures. Data grouping simplifies information, emphasizes group differences or similarities, and exposes patterns or relationships.

Tutorial 2.23: An example for grouping sales data by product and region for three different products, is as follows:

```
1. # Import pandas library
2. import pandas as pd
3. # Create a sample sales data frame with columns for product, region, and sales
4. sales_data = pd.DataFrame({
5.     "product": ["A", "A", "B", "B", "C", "C"],
6.     "region": ["North", "South", "North", "South", "North", "South"],
7.     "sales": [100, 200, 150, 250, 120, 300]
8. })
9. # Print the sales data frame
10. print("\nOriginal dataframe")
11. print(sales_data)
12. # Group the sales data by product and calculate the total sales for each product
13. group_by_product = sales_data.groupby("product").sum()
14. # Print the grouped data by product
15. print("\nGrouped by product")
16. print(group_by_product)
17. # Group the sales data by region and calculate the average sales for each region
18. group_by_region = sales_data.groupby("region").sum()
19. # Print the grouped data by region
20. print("\nGrouped by region")
21. print(group_by_region)
```

Output:

```
1. Original dataframe
```


2. product region sales

3. 0 A North 100

4. 1 A South 200

5. 2 B North 150

6. 3 B South 250

7. 4 C North 120

8. 5 C South 300

9.

10. Grouped by product

11. region sales

12. product

13. A NorthSouth 300

14. B NorthSouth 400

15. C NorthSouth 420

16.

17. Grouped by region

18. product sales

19. region

20. North ABC 370

21. South ABC 750

Tutorial 2.24: An example to show grouping of data based on age interval through binning and calculate the mean score for each group, is as follows:

1. *# Import pandas library to work with data frames*

2. **import** pandas **as** pd

3. *# Create a data frame with student data, including name, age, and score*

4. data = {'Name': ['John', 'Anna', 'Peter', 'Carol', 'David', 'Oystein', 'Hari'],

5. 'Age': [15, 16, 17, 15, 16, 14, 16],

6. 'Score': [85, 92, 78, 80, 88, 77, 89]}

7. df = pd.DataFrame(data)

```

8. # Create age intervals based on the age column, using bins of 13-16 and 17-18
9. age_intervals = pd.cut(df['Age'], bins=[13, 16, 18])
10. # Group the data frame by the age intervals and calculate the mean score for each group
11. grouped_data = df.groupby(age_intervals)
    ['Score'].mean()
12. # Print the grouped data with the age intervals and the mean score
13. print(grouped_data)

```

Output:

```

1. Age
2. (13, 16]    85.166667
3. (16, 18]    78.000000
4. Name: Score, dtype: float64

```

Tutorial 2.25: An example of grouping a scikit-learn digit image dataset based on target labels, where target labels are numbers from 0 to 9, is as follows:

```

1. # Import the sklearn library to load the digits dataset
2. from sklearn.datasets import load_digits
3. # Import the matplotlib library to plot the images
4. import matplotlib.pyplot as plt
5.
6. # Class to display and perform grouping of digits
7. class Digits_Grouping:
8.     # Contructor method to initialize the object's attributes
9.     def __init__(self, digits):
10.         self.digits = digits
11.
12.     def display_digit_image(self):

```

```
13.     # Get the images and labels from the dataset
14.     images = self.digits.images
15.     labels = self.digits.target
16.     # Display the first few images along with their label
    s
17.     num_images_to_display = 5 # You can change this
    number as needed
18.     # Plot the selected few image in a subplot
19.     plt.figure(figsize=(10, 4))
20.     for i in range(num_images_to_display):
21.         plt.subplot(1, num_images_to_display, i + 1)
22.         plt.imshow(images[i], cmap='gray')
23.         plt.title(f"Label: {labels[i]}")
24.         plt.axis('off')
25.     # Save the figure to a file with no padding
26.     plt.savefig('data_grouping.jpg', dpi=600, bbox_inches='tight')
27.     plt.show()
28.
29.     def display_label_based_grouping(self):
30.         # Group the data based on target labels
31.         grouped_data = {}
32.         # Iterate through each image and its corresponding
    target in the dataset.
33.         for image, target in zip(self.digits.images, self.digits.target):
34.             # Check if the current target value is not already
    present as a key in grouped_data.
35.             if target not in grouped_data:
36.                 # If the target is not in grouped_data, add it as
    a new key with an empty list as the value.
```

```

37.         grouped_data[target] = []
38.         # Append the current image to the list associated
with the target key in grouped_data.
39.         grouped_data[target].append(image)
40.         # Print the number of samples in each group
41.         for target, images in grouped_data.items():
42.             print(f"Target {target}: {len(images)} samples")
43.
44. # Create an object of Digits_Grouping class with the digit
s dataset as an argument
45. displayDigit = Digits_Grouping(load_digits())
46. # Call the display_digit_image method to show some ima
ges and labels from the dataset
47. displayDigit.display_digit_image()
48. # Call the display_label_based_grouping method to show
how many samples are there for each label
49. displayDigit.display_label_based_grouping()

```

Output:

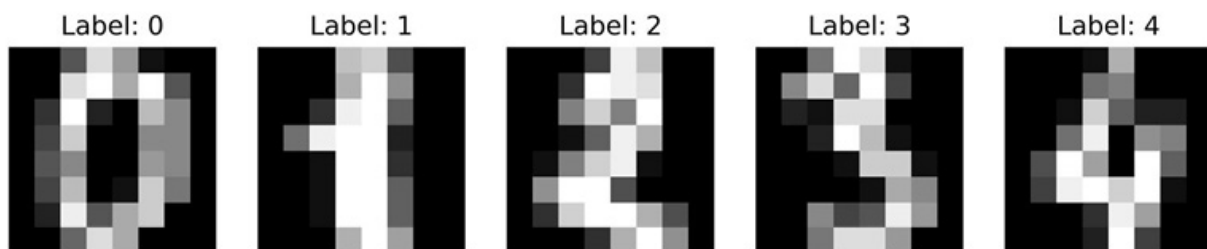


Figure 2.4: Images and respective labels of digit dataset

1. Target 0: 178 samples
2. Target 1: 182 samples
3. Target 2: 177 samples
4. Target 3: 183 samples
5. Target 4: 181 samples
6. Target 5: 182 samples

7. Target 6: 181 samples
8. Target 7: 179 samples
9. Target 8: 174 samples
10. Target 9: 180 samples

Data encoding

Data encoding converts categorical or text-based data into numeric or binary form. For example, you can encode gender data of 100 customers as 0 for male and 1 for female. This encoding corresponds to a specific value or level of the categorical variable to assist machine learning algorithms and statistical models. Encoding data helps manage non-numeric data, reduces data dimensionality, and enhances model performance. It is useful because it allows us to convert data from one form to another, usually for the purpose of transmission, storage, or analysis. Data encoding can help us prepare data for analysis, develop features, compress data, and protect data.

There are several techniques for encoding data, depending on the type and purpose of the data as follows:

- **One-hot encoding:** This technique converts categorical variables, which have a finite number of discrete values or categories, into binary vectors of 0s and 1s. Each category is represented by a unique vector where only one element is 1 and the rest are 0. Appropriate when ordinality is important. One-hot encoding generates a column for every unique category variable value, and binary 1 or 0 values indicate the presence or absence of each value in each row. This approach encodes categorical data in a manner that facilitates comprehension and interpretation by machine learning algorithms. Nevertheless, it expands data dimensions and produces sparse matrices.

Tutorial 2.26: An example of applying one-hot encoding in gender and color, is as follows:

```

1. import pandas as pd
2. # Create a sample dataframe with 3 columns: name, gender and color
3. df = pd.DataFrame({
4.     'name': ['Alice', 'Eve', 'Lee', 'Dam', 'Eva'],
5.     'gender': ['F', 'F', 'M', 'M', 'F'],
6.     'color': ['yellow', 'green', 'green', 'yellow', 'pink']
7. })
8. # Print the original dataframe
9. print("Original dataframe")
10. print(df)
11. # Apply one hot encoding on the gender and color columns using pandas.get_dummies()
12. df_encoded = pd.get_dummies(df, columns=['gender', 'color'], dtype=int)
13. # Print the encoded dataframe
14. print("One hot encoded dataframe")
15. df_encoded

```

Tutorial 2.27: An example of applying one-hot encoding in object data type column in data frame using UCI adult dataset, is as follows:

```

1. import pandas as pd
2. import numpy as np
3. # Read the json file from the direcotory
4. diabetes_df = pd.read_csv(
5.     "/workspaces/ImplementingStatisticsWithPython/data/chapter2/Adult_UCI/adult.data")
6.
7. # Define a function for one hot encoding
8. def one_hot_encoding(diabetes_df):
9.     # Identify columns that are categorical to apply one h

```

ot encoding in them only

```
10. columns_for_one_hot = diabetes_df.select_dtypes(include="object").columns
11. # Apply one hot encoding to the categorical columns
12. diabetes_df = pd.get_dummies(
13.     diabetes_df, columns=columns_for_one_hot, prefix=
columns_for_one_hot, dtype=int)
14. # Display the transformed dataframe
15. print(display(diabetes_df.head(5)))
16.
17. # Call the one hot encoding method by passing dataframe as argument
18. one_hot_encoding(diabetes_df)
```

- **Label coding:** This technique assigns a numeric value to each category of a categorical variable. The numerical values are usually sequential integers starting from 0. Appropriate when order is important. The transformed variable will have numerical values instead of categorical values. Its drawback is the loss of information about the similarity or difference between categories.

Tutorial 2.28: An example of applying label encoding for categorical variables, is as follows:

```
1. import pandas as pd
2. # Create a data frame with name, gender, and color columns
3. df = pd.DataFrame({
4.     'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane', 'Bo'],
5.     'gender': ['F', 'M', 'M', 'M', 'F', 'F', 'M'],
6.     'color': ['red', 'blue', 'green', 'yellow', 'pink', 'red', 'blue']
7. })
```

```

8. # Convert the gender column to a categorical variable and assign numerical codes to each category
9. df['gender_label'] = df['gender'].astype('category').cat.codes
10. # Convert the color column to a categorical variable and assign numerical codes to each category
11. df['color_label'] = df['color'].astype('category').cat.codes
12. # Print the data frame with the label encoded columns
13. print(df)

```

- **Binary encoding:** Binary coding converts categorical variables into fixed-length binary codes. Performing a binary search on sorted categories records the comparison result as 1 or 0. Each unique category is assigned an integer value, which is then converted into binary code. This reduces the number of columns necessary to describe categorical data, unlike one-hot encoding, which requires a new column for each unique category. However, binary encoding has certain downsides, such as the creation of ordinality or hierarchy within categories that did not previously exist, making interpretation and analysis more challenging.

Tutorial 2.29: An example of applying binary encoding for categorical variables using **category_encoders** package from **pip**, is as follows:

```

1. # Import pandas library and category_encoders library
2. import pandas as pd
3. import category_encoders as ce
4. # Create a sample dataframe with 3 columns: name, gender and color
5. df = pd.DataFrame({
6.     'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane', 'Bo'],
7.     'gender': ['F', 'M', 'M', 'M', 'F', 'F', 'M'],

```



```

8.     'color': ['red', 'blue', 'green', 'yellow', 'pink', 'red', 'blue
    ']
9. })
10. # Print the original dataframe
11. print("Original dataframe")
12. print(df)
13. # Create a binary encoder object
14. encoder = ce.BinaryEncoder(cols=['gender', 'color'])
15. # Fit and transform the dataframe using the encoder
16. df_encoded = encoder.fit_transform(df)
17. # Print the encoded dataframe
18. print("Binary encoded dataframe")
19. print(df_encoded)

```

Output:

```

1. Original dataframe
2.    name gender  color
3. 0  Alice    F    red
4. 1   Bob    M   blue
5. 2 Charlie    M  green
6. 3  David    M yellow
7. 4   Eve    F   pink
8. 5   Ane    F    red
9. 6    Bo    M   blue
10. Binary encoded dataframe
11.    name  gender_0  gender_1  color_0  color_1  color_2
12. 0  Alice        0         1         0         0         1
13. 1   Bob         1         0         0         1         0
14. 2 Charlie        1         0         0         1         1
15. 3  David         1         0         1         0         0
16. 4   Eve         0         1         1         0         1
17. 5   Ane         0         1         0         0         1

```

18. 6 Bo 1 0 0 1 0

The difference between binary encoders and one-hot encoders is in how they encode categorical variables. One-hot encoding, which creates a new column for each categorical value and marks their existence with either 1 or 0. However, binary encoding converts each categorical variable value into a binary code and separates them into distinct columns. For example, data frame's color column can be one-hot encoded, as shown below.

The same color column of the data frame as can be binary encoded, where each unique combination of bits represents a specific color, as follows:

Tutorial 2.30: An example to illustrate difference of one-hot encoding and binary encoding, is as follows:

```
1. # Import the display function to show the data frames
2. from IPython.display import display
3. # Import pandas library to work with data frames
4. import pandas as pd
5. # Import category_encoders library to apply different encoding techniques
6. import category_encoders as ce
7.
8. # Class to compare the difference between one-hot encoding and binary encoding
9. class Encoders_Difference:
10.     # Constructor method to initialize the object's attribute
11.     def __init__(self, df):
12.         self.df = df
13.
14.     # Method to apply one-hot encoding to the color column
```

```
15. def one_hot_encoding(self):
16.     # Use the get_dummies function to create binary vectors for each color category
17.     df_encoded1 = pd.get_dummies(df, columns=['color'], dtype=int)
18.     # Display the encoded data frame
19.     print("One-hot encoded dataframe")
20.     print(df_encoded1)
21.
22.     # Method to apply binary encoding to the color column
23.     def binary_encoder(self):
24.         # Create a binary encoder object with the color column as the target
25.         encoder = ce.BinaryEncoder(cols=['color'])
26.         # Fit and transform the data frame with the encoder object
27.         df_encoded2 = encoder.fit_transform(df)
28.         # Display the encoded data frame
29.         print("Binary encoded dataframe")
30.         print(df_encoded2)
31.
32.     # Create a sample data frame with 3 columns: name, gender and color
33.     df = pd.DataFrame({
34.         'name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Ane'],
35.         'gender': ['F', 'M', 'M', 'M', 'F', 'F'],
36.         'color': ['red', 'blue', 'green', 'blue', 'green', 'red']
37.     })
38.
39.     # Create an object of Encoders_Difference class with the sample data frame as an argument
```

```

40. encoderDifference_obj = Encoders_Difference(df)
41. # Call the one_hot_encoding method to show the result of one-hot encoding
42. encoderDifference_obj.one_hot_encoding()
43. # Call the binary_encoder method to show the result of binary encoding
44. encoderDifference_obj.binary_encoder()

```

Output:

1. One-hot encoded dataframe

	name	gender	color_blue	color_green	color_red
3. 0	Alice	F	0	0	1
4. 1	Bob	M	1	0	0
5. 2	Charlie	M	0	1	0
6. 3	David	M	1	0	0
7. 4	Eve	F	0	1	0
8. 5	Ane	F	0	0	1

9. Binary encoded dataframe

	name	gender	color_0	color_1
11. 0	Alice	F	0	1
12. 1	Bob	M	1	0
13. 2	Charlie	M	1	1
14. 3	David	M	1	0
15. 4	Eve	F	1	1
16. 5	Ane	F	0	1

- **Hash coding:** This technique applies a hash function to each category of a categorical variable and maps it to a numeric value within a predefined range. The hash function is typically a one-way function that produces a unique output for each input.
- **Feature scaling:** This technique transforms numerical variables into a common scale or range, usually between 0 and 1 or -1 and 1. Different methods of feature scaling,

such as min-max scaling, standardization, and normalization, are discussed above.

Missing data, detecting and treating outliers

Data values that are not stored or captured for some variables or observations in a dataset are referred to as missing data. It may happen for a number of reasons, including human mistakes, equipment malfunctions, data entry challenges, privacy concerns, or flaws with survey design. The accuracy and reliability of the analysis and inference can be impacted by missing data. In structured data, identifying missing values is pretty easy whereas in semi and unstructured it may not always be the case.

Tutorial 2.31: An example to illustrate how to count sum of all the null and missing values in large data frame, is as follows:

```
1. import pandas as pd
2. # Create a dataframe with some null values
3. df = pd.DataFrame({"Name": ["Alice", "Bob", "Charlie",
    None, "Eve"],
4.                    "Age": [25, 30, 35, None, 40],
5.                    "Gender": ["F", "M", None, None, "F"]})
6. # Display the dataframe
7. print("Original dataframe")
8. print(df)
9. # Use isna().sum() to view the sum of null values for each column
10. print("Null value count in dataframe")
11. print(df.isna().sum())
```

Output:

```
1. Original dataframe
2.    Name  Age Gender
```

```

3. 0 Alice 25.0 F
4. 1 Bob 30.0 M
5. 2 Charlie 35.0 None
6. 3 None NaN None
7. 4 Eve 40.0 F
8. Null value count in dataframe
9. Name 1
10. Age 1
11. Gender 2

```

Some of the most common techniques to handle missing data are deletion of missing data row or column, imputation of missing value and prediction of missing value.

Tutorial 2.32: An example to show all columns in data frame and remaining columns after applying drop, is as follows:

```

1. import pandas as pd
2. # Read the json file from the direcotory
3. diabetes_df = pd.read_csv(
4.     "/workspaces/ImplementingStatisticsWithPython/data
   /chapter2/Adult_UCI/adult.data")
5. # View all columns in dataframe
6. print("Columns before drop")
7. print(diabetes_df.columns)
8. # Drop the 'Age' and 'Work' columns
9. diabetes_df = diabetes_df.drop(columns=
   [' Work', ' person_id', ' education', ' education_number',
10.     ' marital_status'], axis=1)
11. # Verify the updated DataFrame
12. print("Columns after drop")
13. print(diabetes_df.columns)

```

Output:

1. Columns before drop
2. `Index(['Age', ' Work', ' person_id', ' education', ' education_number',`
3. `' marital_status', ' occupation', ' relationship', ' race',`
4. `' capital_gain', ' capital_loss', ' hours_per_week', ' na`
5. `' income'],`
6. `dtype='object')`
7. Columns after drop
8. `Index(['Age', ' occupation', ' relationship', ' race', ' gende`
9. `' capital_gain', ' capital_loss', ' hours_per_week', ' na`
10. `' income'],`
11. `dtype='object')`

Data imputation replaces missing or invalid data values with reasonable estimates, improving the quality and usability of data for analysis and modeling. For example, let us examine a data set that includes student grades in four subjects that is, Mathematics, English, Science, and History. However, some grades are either invalid or missing, as demonstrated in the following table:

Name	Math	English	Science	History
Ram	90	85	95	?
Deep	80	?	75	70
John	?	65	80	60
David	70	75	?	65

Table 2.5: *Grades of students in different subjects*

One easiest method for data imputation is by calculating the mean (average) of available values for each column. For

example, the mean of math is $(90 + 80 + 70) / 3 = 80$, the mean of English is $(85 + 65 + 75) / 3 = 75$, and so on. These means can be used to replace missing or invalid values with the corresponding mean values as shown in [Table 2.4](#):

Name	Math	English	Science	History
Ram	90	85	95	73.3
Deep	80	75	75	70
John	80	65	80	60
David	70	75	78.3	65

Table 2.6: *Imputing missing scores based on mean*

Tutorial 2.33: An example to illustrate imputation of missing value in data frame with **mean()**, is as follows:

```

1. import pandas as pd
2. # Create a DataFrame with student data using a dictionary
3. data = {'Name': ['John', 'Anna', 'Peter', 'Hari', 'Suresh', 'Ram'],
4.         'Age': [15, 16, np.nan, 16, 30, 31],
5.         'Score': [85, 92, 78, 80, np.nan, 76]}
6. student_DF = pd.DataFrame(data)
7. # Print a message before showing the dataframe with missing values
8. print(f'Before Mean Imputation DataFrame')
9. # Display the dataframe with missing values using the display function
10. print(student_DF)
11. # Calculate the mean of the Age column and store it in a variable
12. mean_age = student_DF['Age'].mean()
13. # Calculate the mean of the Score column and store it in

```


a variable

```
14. mean_score = student_DF['Score'].mean()
15. # Print a message before showing the dataframe with imputed values
16. print(f'DataFrame after mean imputation')
17. # Replace the missing values in the dataframe with the mean values using the fillna method and a dictionary
18. student_DF = student_DF.fillna(value={'Age': mean_age, 'Score': mean_score})
19. # Display the dataframe with imputed values using the display function
20. print(student_DF)
```

Output:

1. Before Mean Inputation DataFrame

2. Name Age Score

3. 0 John 15.0 85.0

4. 1 Anna 16.0 92.0

5. 2 Peter NaN 78.0

6. 3 Hari 16.0 80.0

7. 4 Suresh 30.0 NaN

8. 5 Ram 31.0 76.0

9. DataFrame after mean imputation

10. Name Age Score

11. 0 John 15.0 85.0

12. 1 Anna 16.0 92.0

13. 2 Peter 21.6 78.0

14. 3 Hari 16.0 80.0

15. 4 Suresh 30.0 82.2

16. 5 Ram 31.0 76.0

In some cases, missing value prediction can be estimated and predicted based on other information available in the

data set. If the estimation is not done properly, it can introduce noise and uncertainty into the data. Missingness can also be used as a variable to indicate whether a value was missing or not. However, this can increase dimensionality. More about this is discussed in later chapters. Some general guidelines to handle missing values are as follows:

- If the missing data are randomly distributed in the data set and are not too many (less than 5% of the total observations), then a simple method such as replacing the missing values with the mean, median, or mode of the corresponding variable may be sufficient.
- If the missing data are not randomly distributed or are too many (more than 5% of the total observations), a simple method may introduce bias and reduce the variability of the data. In this case, a more sophisticated method that takes into account the relationship between variables may be preferable. For example, you can use a regression model to predict the missing values based on other variables, or a nearest neighbor approach to find the most similar observation and use its value as an imputation.
- If the missing data are longitudinal, that is, they occur in repeated measurements over time, then a method that takes into account the temporal structure of the data may be more appropriate. For example, one can use a time series model to predict the missing values based on past and future observations, or a mixed effects model to account for both fixed and random effects over time.

Visualization and plotting of data

Data visualization and plotting entail creating graphical representations of information, including charts, graphs, maps, and other visual aids. Using visual tools is imperative for comprehending intricate data and presenting

information captivantly and efficiently. This is essential in recognizing patterns, trends, and anomalies in a dataset and conveying our discoveries effectively. For data visualization and plotting, there are various libraries available such as **Matplotlib**, **Seaborn**, **Plotly**, **Bokeh**, and **Vega-altair**, among others. When presenting information in a chart, the first step is to determine what type of chart is appropriate for the data. There are many factors to consider when choosing a chart type, such as the number of variables, the type of data the purpose of the analysis, and the preferences of the audience. To compare values within or between groups, utilize a bar graph, column graph, or bullet graph. These charts are effective for displaying distinctions, rankings, or proportions of categories. Pie charts, donut charts, and tree maps are effective for illustrating how data is composed of various components. These charts are useful for depicting percentages or fractions of a total.

A line, area or column chart is ideal for displaying temporal changes. These graphs are efficient in presenting trends, patterns, or fluctuations within a specific time frame. Use a scatter plot, bubble chart, or connected scatter plot to display the relationship between multiple variables. These charts effectively portray how variables are interconnected. To effectively display a data distribution across a range of values, consider utilizing a histogram, box plot, or scatter plot. These plots are ideal for illustrating the data's shape, spread, and outliers. The various types of plots are discussed as follows:

Line plot

Line plots are ideal for displaying trends and changes in continuous or ordered data points, especially for time series data that depicts how a variable evolves over time. For instance, one could use a line plot to monitor a patient's blood pressure readings taken at regular intervals

throughout the year, to monitor their health.

Tutorial 2.34: An example to plot patient blood pressure reading taken at different months of year using line plot, is as follows:

```
1. # Import matplotlib.pyplot module
2. import matplotlib.pyplot as plt
3. # Create a list of dates for the x-axis
4.
   dates = ["01/08/2023", "01/09/2023", "01/10/2023", "01/11/2023", "01/12/2023"]
5. # Create a list of blood pressure readings for the y-axis
6. bp_readings = [120, 155, 160, 170, 175]
7. # Plot the line plot with dates and bp_readings
8. plt.plot(dates, bp_readings)
9. # Add a title for the plot
10.
    plt.title("Patient's Blood Pressure Readings Throughout the Year")
11. # Add labels for the x-axis and y-axis
12. plt.xlabel("Date")
13. plt.ylabel("Blood Pressure (mmHg)")
14. # Show the plot
15.
    plt.savefig("lineplot.jpg", dpi=600, bbox_inches='tight')
16. plt.show()
```

Output:

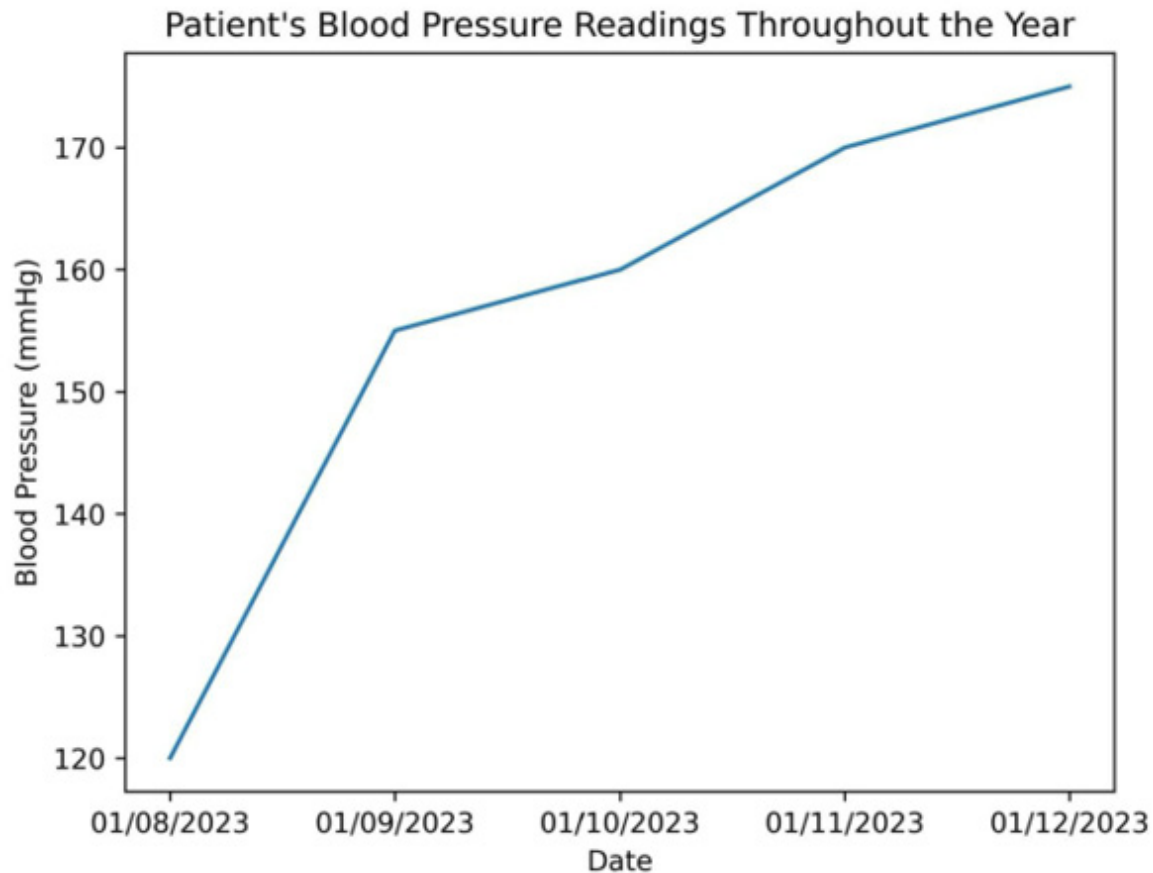


Figure 2.5: Patient's blood pressure over the month in a line graph.

Pie chart

Pie chart is useful when showing the parts of a whole and the relative proportions of different categories. Pie charts are best suited for categorical data with only a few different categories. Use pie charts to display the percentages of daily calories consumed from carbohydrates, fats, and proteins in a diet plan.

Tutorial 2.35: An example to display the percentages of daily calories consumed from carbohydrates, fats, and proteins in a pie chart, is as follows:

1. `# Import matplotlib.pyplot module`
2. `import matplotlib.pyplot as plt`
3. `# Create a list of dates for the x-axis`

```
4.
   dates = ["01/08/2023", "01/09/2023", "01/10/2023", "01/11/2023", "01/12/2023"]
5. # Create a list of blood pressure readings for the y-axis
6. bp_readings = [120, 155, 160, 170, 175]
7. # Plot the line plot with dates and bp_readings
8. plt.plot(dates, bp_readings)
9. # Add a title for the plot
10.
    plt.title("Patient's Blood Pressure Readings Throughout the Year")
11. # Add labels for the x-axis and y-axis
12. plt.xlabel("Date")
13. plt.ylabel("Blood Pressure (mmHg)")
14. # Show the plot
15.
    plt.savefig("lineplot.jpg", dpi=600, bbox_inches='tight')
16. plt.show()
```

Output:

Percentages of Daily Calories Consumed from Carbohydrates, Fats, and Proteins

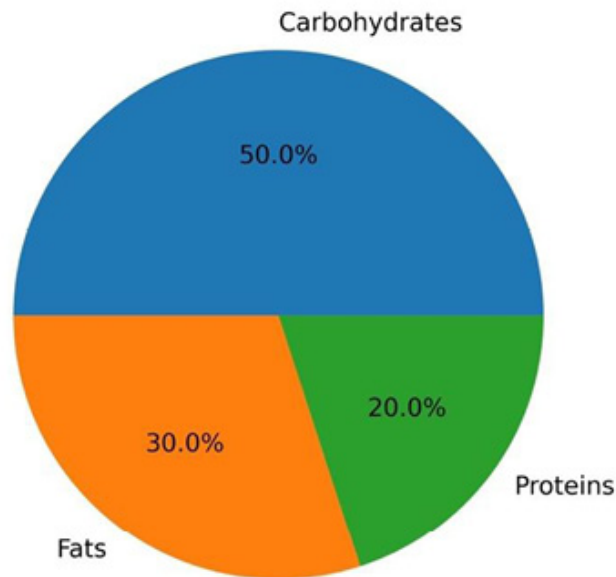


Figure 2.6: Daily calories consumed from carbohydrates, fats, and proteins in a pie chart

Bar chart

Bar charts are suitable for comparing values of different categories or showing the distribution of categorical data. Mostly useful for categorical data with distinct categories data type. For example: comparing the average daily step counts of people in their 20s, 30s, 40s, and so on, to assess the relationship between age and physical activity.

Tutorial 2.36: An example to plot average daily step counts of people in their 20s, 30s, 40s, and so on using bar chart, is as follows:

1. `# Import matplotlib.pyplot module`
2. `import matplotlib.pyplot as plt`
3. `# Create a list of percentages of daily calories consumed from carbohydrates, fats, and proteins`
4. `calories = [50, 30, 20]`
5. `# Create a list of labels for the pie chart`

```

6. labels = ["Carbohydrates", "Fats", "Proteins"]
7. # Plot the pie chart with calories and labels
8. plt.pie(calories, labels=labels, autopct="%1.1f%%")
9. # Add a title for the pie chart
10. plt.title("Percentages of Daily Calories Consumed from
    Carbohydrates, Fats, and Proteins")
11. # Show the pie chart
12. plt.savefig("piechart1.jpg", dpi=600, bbox_inches='tight')
plt.show()

```

Output:

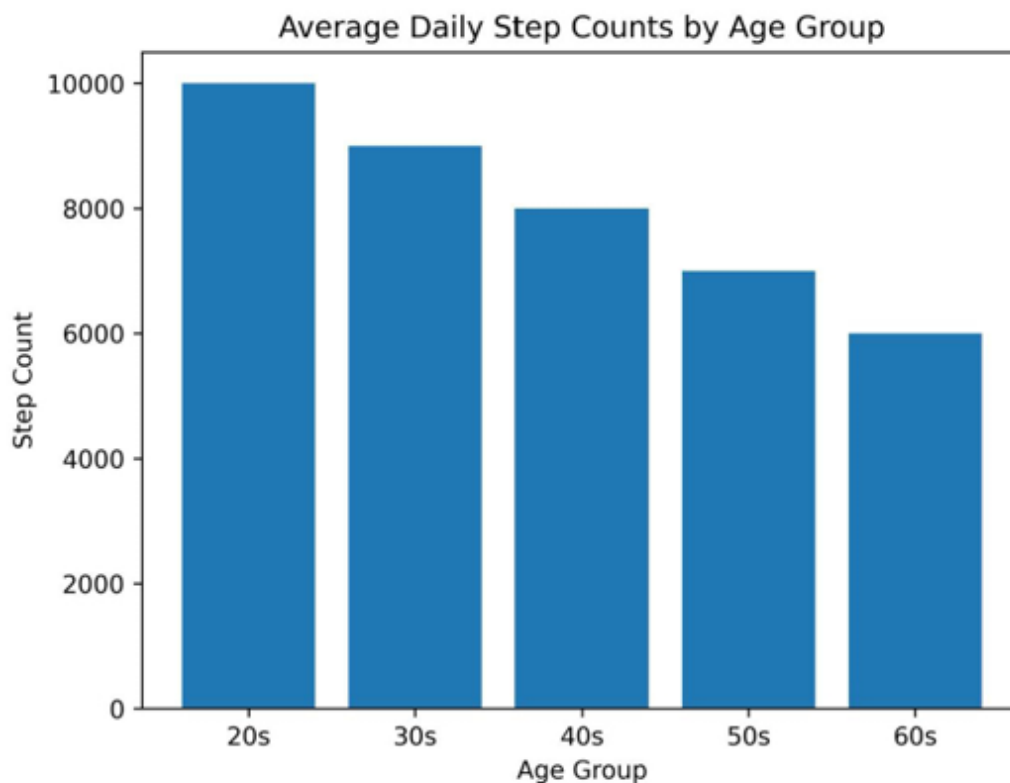


Figure 2.7: Daily step counts of people in different age category using bar chart

Histogram

Histograms are used to visualize the distribution of continuous data or to understand the frequency of values within a range. Mostly used for continuous data. For example, to show **Body Mass Indexes (BMIs)** in a large sample of individuals to see how the population's BMIs are distributed.

Tutorial 2.37: An example to plot distribution of individual BMIs in a histogram plot, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Generate a large sample of BMIs using numpy.random
   .normal function
4. # The mean BMI is 25 and the standard deviation is 5
5. bmis = np.random.normal(25, 5, 1000)
6. # Plot the histogram with bmis and 20 bins
7. plt.hist(bmis, bins=20)
8. # Add a title for the histogram
9. plt.title("Histogram of BMIs in a Large Sample of Individuals")
10. # Add labels for the x-axis and y-axis
11. plt.xlabel("BMI")
12. plt.ylabel("Frequency")
13. # Show the histogram
14. plt.savefig('histogram.jpg', dpi=600, bbox_inches='tight')
15. plt.show()
```

Output:

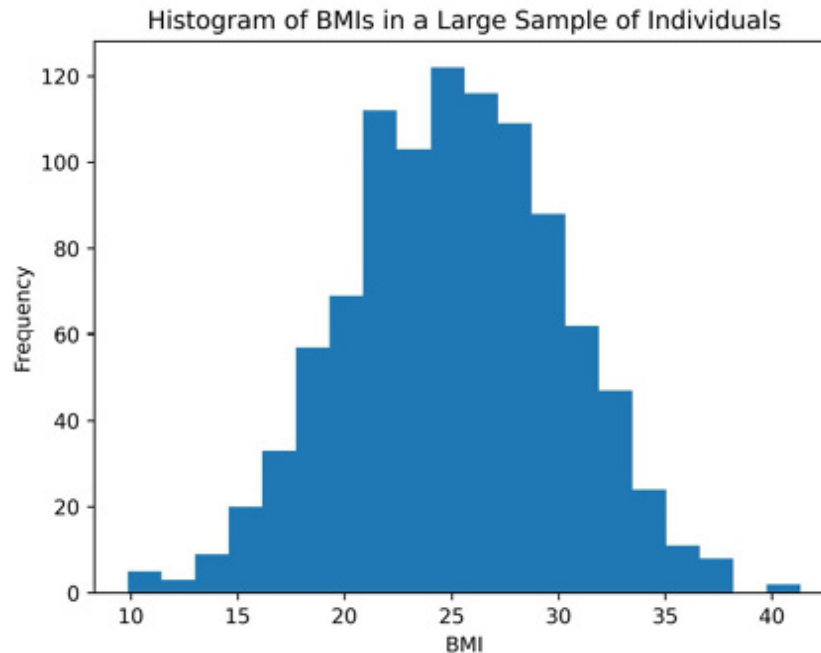


Figure 2.8: Distribution of Body Mass Index of individuals in histogram

Scatter plot

Scatter plots are ideal for visualizing relationships between two continuous variables. It is mostly used for two continuous variables that you want to analyze for correlation or patterns. For example, plotting the number of hours of sleep on the x-axis and the self-reported stress levels on the y-axis to see if there is a correlation between the two variables.

Tutorial 2.38: An example to plot number of hours of sleep and stress levels to show their correlation in a scatter plot, is as follows:

1. `import matplotlib.pyplot as plt`
2. `import numpy as np`
3. `# Generate a sample of hours of sleep using numpy.random.uniform function`
4. `# The hours of sleep range from 4 to 10`
5. `sleep = np.random.uniform(4, 10, 100)`
6. `# Generate a sample of stress levels using numpy.rando`

m.normal function

```
7. # The stress levels range from 1 to 10, with a negative c  
   orrelation with sleep  
8. stress = np.random.normal(10 - sleep, 1)  
9. # Plot the scatter plot with sleep and stress  
10. plt.scatter(sleep, stress)  
11. # Add a title for the scatter plot  
12. plt.title("Scatter Plot of Hours of Sleep and Stress Levels  
   ")  
13. # Add labels for the x-axis and y-axis  
14. plt.xlabel("Hours of Sleep")  
15. plt.ylabel("Stress Level")  
16. # Show the scatter plot  
17. plt.savefig("scatterplot.jpg", dpi=600, bbox_inches='tight')  
18. plt.show()
```

Output:

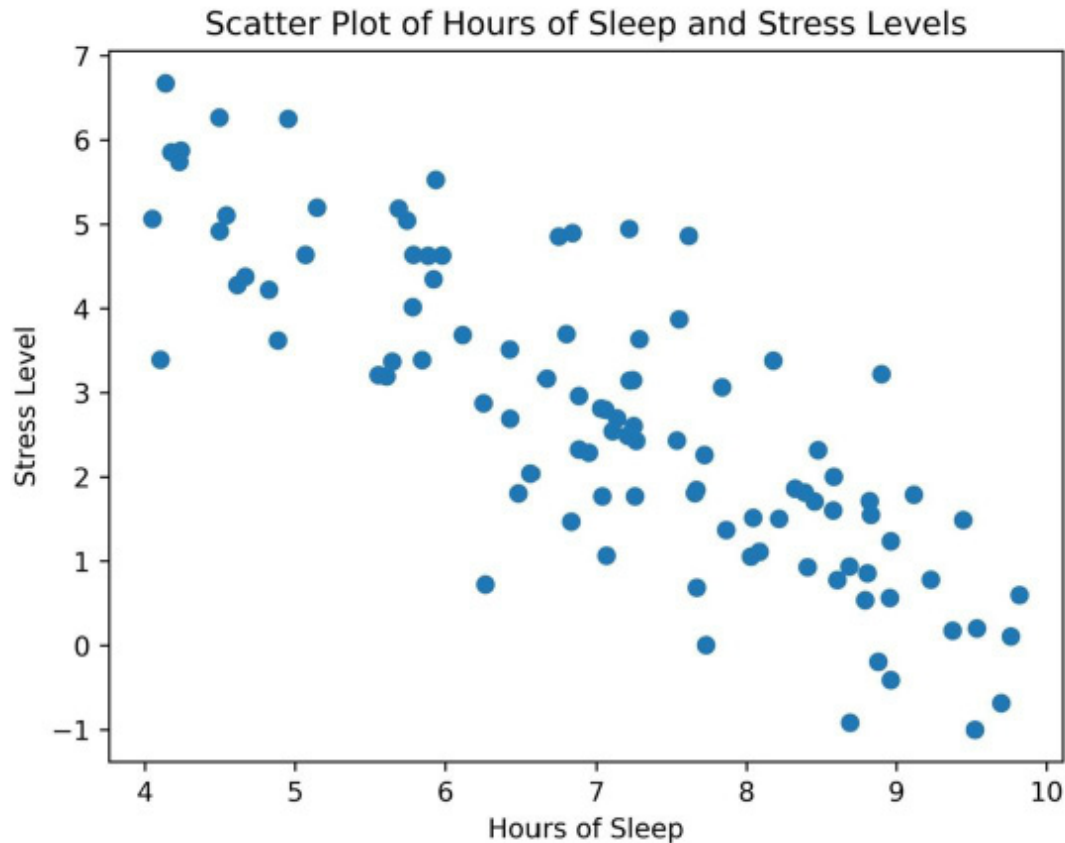


Figure 2.9: Number of hours of sleep and stress levels in a scatter plot

Stacked area plot

Stacked area chart illustrates the relationship between multiple variables throughout a continuous time frame. It is a useful tool for comparing the percentages or proportions of various components that comprise the entirety.

Tutorial 2.39: An example to plot patient count based on age categories (child, teen, adult, old) over the years using stacked area plot, is as follows:

1. `import matplotlib.pyplot as plt`
2. `import numpy as np`
3. `# Create a sample data set with four variables`
4. `x = np.arange(2020, 2025)`
5. `y1 = np.random.randint(1, 10, 5)`

```
6. y2 = np.random.randint(1, 10, 5)
7. y3 = np.random.randint(1, 10, 5)
8. y4 = np.random.randint(1, 10, 5)
9. # Plot the stacked area plot with x and y1, y2, y3, y4
10. plt.stackplot(x, y1, y2, y3, y4, labels=
    ["y1", "y2", "y3", "y4"])
11. # Add a title for the stacked area plot
12. plt.title("Stacked Area Plot of Sample Data Set")
13. # Add labels for the x-axis and y-axis
14. plt.xlabel("Year")
15. plt.ylabel("y")
16. # Add a legend for the plot
17. plt.legend()
18. # Show the stacked area plot
19. plt.savefig('stackedareaplot.jpg', dpi=600, bbox_inches=
    'tight')
20. plt.show()
```

Output:

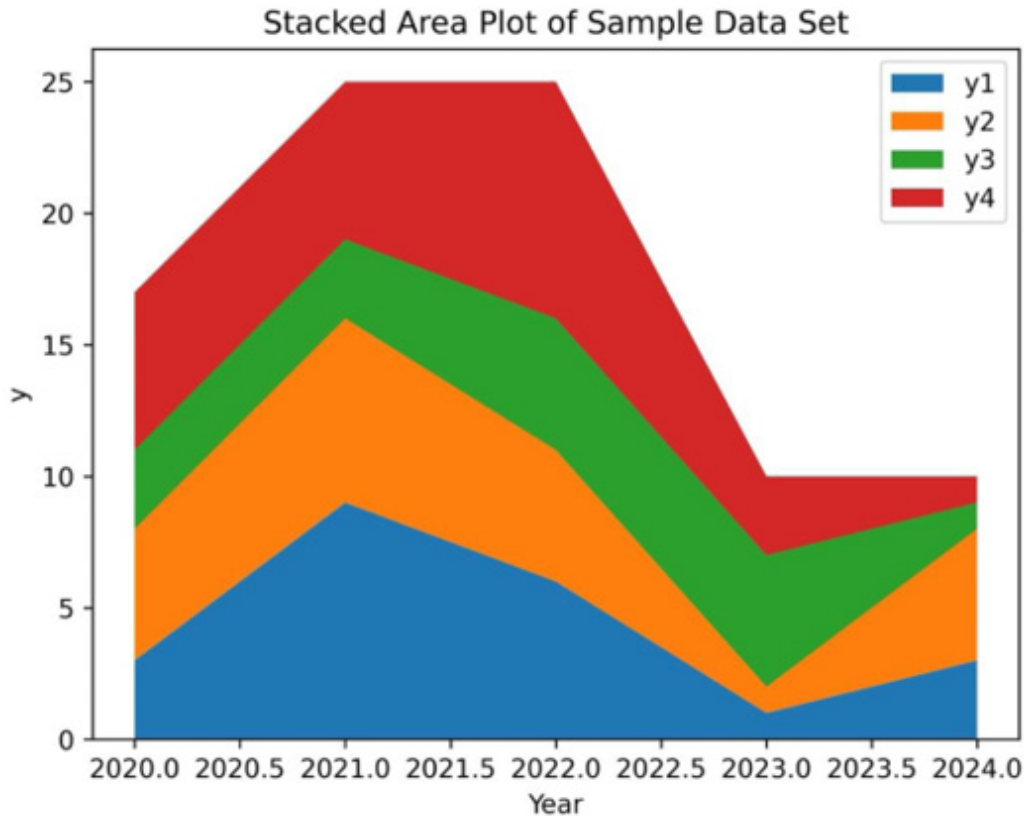


Figure 2.10: Number of patients based on age categories in stacked area plot

Dendrograms

Dendrogram illustrates the hierarchy of clustered data points based on their similarity or distance. It allows for exploration of data patterns and structure, as well as identification of clusters or groups of data points that are similar.

Violin plot

Violin plot shows how numerical data is distributed across different categories, allowing for comparisons of shape, spread, and outliers. This can reveal similarities or differences between categories.

Word cloud

Word cloud is a type of visualization that shows the

frequency of words in a text or a collection of texts. It is useful when you want to explore the main themes or topics of the text, or to see which words are most prominent or relevant.

Graph

Graph visually displays the relationship between two or more variables using points, lines, bars, or other shapes. It offers valuable insights into data patterns, trends, and correlations, as well as allows for the comparison of values or categories. It is suggested to use graphs for data analysis.

Conclusion

Exploratory data analysis involves several critical steps to prepare and analyze data effectively. Data is first aggregated, normalized, standardized, transformed, binned, and grouped. Missing data and outliers are detected and treated appropriately before visualization and plotting. Data encoding is also used to handle categorical variables. These preprocessing steps are essential for EDA because they improve the quality and reliability of the data and help uncover useful insights and patterns. EDA includes many steps beyond these and depends on the data, problem statement, objective, and others. To summarize the main steps, it includes. Data aggregation combines data from different sources or groups to form a summary or a new data set. Data aggregation reduces the complexity and size of the data, and to reveal patterns or trends across different categories or dimensions. Data normalization scales the numerical values of the data to a common range, such as 0 to 1 or -1 to 1. Data normalization reduces the effect of different units or scales on the data, making the data comparable and consistent. Data standardization contributes to remove the effect of outliers or extreme

values on the data, and to make the data follow a normal distribution. The data transformation helps to change the shape or distribution of the data, and to make the data more suitable for certain analyses or models. Data binning is dividing the numerical values of the data into discrete intervals or bins, such as low, medium, high, etc. Data binning can help to reduce the noise or variability of the data, and to create categorical variables from numerical variables. The data grouping groups the data based on certain criteria or attributes, such as age, gender, location, etc. Data grouping helps to segment or classify the data into meaningful categories or clusters, and to analyze the differences or similarities between groups. Data encoding techniques, such as one-hot encoding, label encoding, and ordinal encoding, convert categorical variables into numerical variables, making the data compatible with analyses or models that require numerical inputs. Data cleaning detects and treats missing data and outliers. Similarly when performing EDA of data, data visualization assists to understand the data, display the summary, view the relationship among the variables through charts, graphs and other graphical representations. As you begin your work in data science and statistics, these steps cover the things you need to consider. So, this is the initial step while working with data, and everything starts with this.

In [*Chapter 3: Frequency Distribution, Central Tendency, Variability*](#), we will start with descriptive statistics, which will delve into ways to describe and understand the pre-processed data based on frequency distribution, central tendency, variability.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Frequency Distribution, Central Tendency, Variability

Introduction

Descriptive statistics is a way of better describing and summarizing the data and its characteristics, in a meaningful way. The part of descriptive statistics includes the measure of frequency distribution, the measure of central tendency, which includes mean, median, mode, measure of variability, measure of association, and shapes. Descriptive statistics simply show what the data shows. Frequency distribution is primarily used to show the distribution of categorical or numerical observations, counting in different categories and ranges. Central tendency calculates the mode, which is the most frequent data set, median which is the middle value in an ordered set and mean which is the average value. The measures of variability estimate how much the values of a variable are spread, or it calculates the variations in the value of the variable. They allow us to understand how far the data deviate from the typical or average value. Range, variance,

and standard deviation are commonly used measures of variability. Measures of association estimate the relationship between two or more variables, through scatterplots, correlation, regression. Shapes describe the pattern and distribution of data by measuring skewness, symmetry of shape, bimodal, unimodal, and uniform modality, kurtosis, counting and grouping.

Structure

In this chapter, we will discuss the following topics:

- Measures of frequency
- Measures of central tendency
- Measures of variability or dispersion
- Measures of association
- Measures of shape

Objectives

By the end of this chapter, readers will learn about descriptive statistics and how to use them to gain meaningful insights. You will gain the skills necessary to calculate measures of frequency distribution, central tendency, variability, association, shape, and how to apply them using Python.

Measure of frequency

A measure of frequency counts the number of times a specific value or category appears within a dataset. For example, to find out how many children in a class like each animal, you can apply the measure of frequency on a data set that contains the five most popular animals. [Table 3.1](#) displays how many times each animal was chosen by the 10 children. Out of the 10 children, 4 like dogs, 3 like cats, 2 like cow, and 1 like rabbit.

Animal	Frequency
Dog	4
Cat	3
Cow	2
Rabbit	1

Table 3.1: *Frequency of animal chosen by children*

Another option is to visualize the frequency using plots, graphs, and charts. For example, we can use pie chart, bar chart, and other charts.

Tutorial 3.1: To visualize the measure of frequency using pie chart, bar chart, by showing both plots in subplots, is as follows:

```

1. import pandas as pd
2. import matplotlib.pyplot as plt
3. # Create a data frame with the new data
4. data = {"Animal": ["Dog", "Cat", "Cow", "Rabbit"],
5.         "Frequency": [4, 3, 2, 1]}
6. df = pd.DataFrame(data)
7. # Create a figure with three subplots
8. fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 6))
9. # Plot a pie chart of the frequency of each animal on the
   first subplot
10. ax1.pie(df["Frequency"], labels=df["Animal"], autopct="
    %1.1f%%")
11. ax1.set_title("Pie chart of favorite animals")
12. # Plot a bar chart of the frequency of each animal on the
    second subplot
13. ax2.bar(df["Animal"], df["Frequency"], color=
    ["brown", "orange", "black", "gray"])
14. ax2.set_title("Bar chart of favorite animals")

```

```
15. ax2.set_xlabel("Animal")
16. ax2.set_ylabel("Frequency")
17. # Save and show the figure
18. plt.savefig('measure_frequency.jpg',dpi=600,bbox_inches='tight')
19. plt.show()
```

Output:

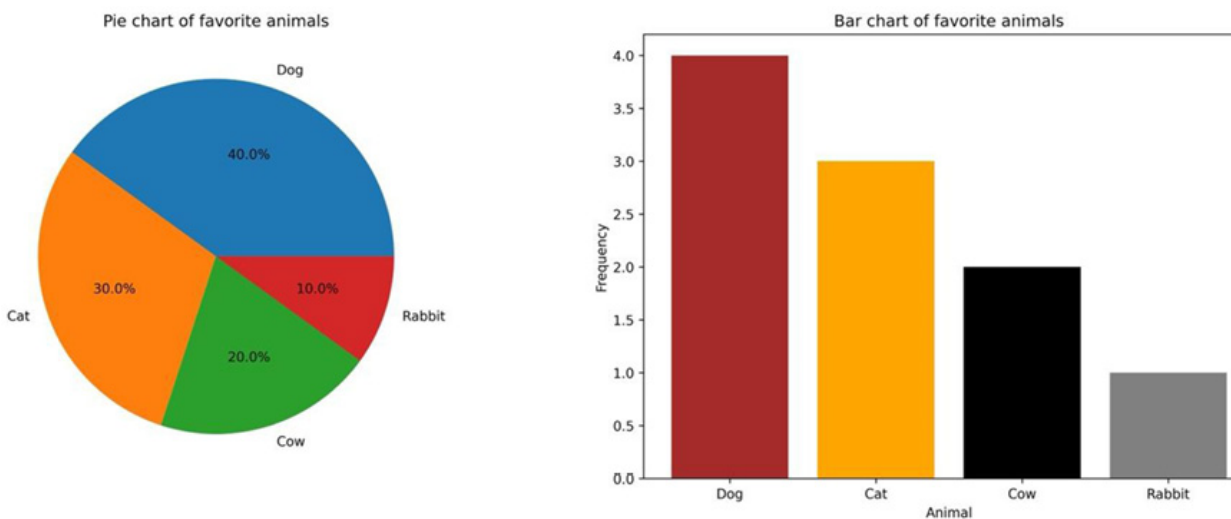


Figure 3.1: Frequency distribution in pie and bar charts

Frequency tables and distribution

Frequency tables and distribution are methods of sorting and summarizing data in descriptive statistics. **Frequency tables** display how often each value or category of a variable appears in a dataset. **Frequency distribution** exhibits the frequency pattern of a variable, which can be illustrated using graphs or tables. Distribution is a way of summarizing and displaying the number or proportion of observations for each possible value or category of a variable.

For example, on the data about favorite animals of ten school children, you can create a table that displays how many children like each animal and a distribution chart that

reveals the data's shape as discussed above in the measure of frequency and in the examples of relative and cumulative frequency, as explained in the next section.

Relative and cumulative frequency

Relative frequency is the ratio of the number of times a value or category appears in the data set to the total number of data values. Its relative and calculated by dividing the frequency of a category by the total number of observations. On the other hand, **cumulative frequency** is the total number of observations that fit into a specific range of categories, along with all of the categories that came before it. To calculate it, add the frequency of the current category to the cumulative frequency of the previous category.

For example, suppose we have a data set of the favorite animals of 10 children, as shown in the [Table 3.1](#) above. To determine the relative frequency of each animal, divide the frequency by the total number of children, which is 10. Doing so for dogs the relative frequency is $4/10 = 0.4$, meaning that 40% of the children like dogs. For cats, it is $3/10 = 0.3$, meaning that 30% of the children like cats. Further relative frequencies of each animal are shown in the following table:

Animal	Frequency	Relative frequency
Dog	4	0.4
Cat	3	0.3
Cow	2	0.2
Rabbit	1	0.1

Table 3.2: Relative frequency of each animal

Now, to calculate the cumulative frequency for each animal, add up the relative frequencies of all animals that are less

than or equal to the current animal in the table. For example, dog's cumulative frequency is 0.4, identical to their relative frequency. The cumulative frequency of cats is $0.4 + 0.3 = 0.7$, indicating that 70% of the children prefer dogs or cats. Similarly, relative frequency of cow is $0.4 + 0.3 + 0.2 = 0.9$, which means 90% of the children like dogs, cats and cow, as in shown in [Table 3.3](#):

Animal	Frequency	Relative frequency	Cumulative relative frequency
Dog	4	0.4	0.4
Cat	3	0.3	0.7
Cow	2	0.2	0.9
Rabbit	1	0.1	1

Table 3.3: Comparison of relative and cumulative relative frequency

Tutorial 3.2: An example to view the relative frequency in pie chart and cumulative frequency in a line plot, is as follows:

```

1. import pandas as pd
2. import matplotlib.pyplot as plt
3. # Create a data frame with the given data
4. data = {"Animal": ["Dog", "Cat", "Cow", "Rabbit"],
5.         "Frequency": [4, 3, 2, 1]}
6. df = pd.DataFrame(data)
7. # Calculate the relative frequency by dividing the frequency by the sum of all frequencies
8. df["Relative Frequency"] = df["Frequency"] / df["Frequency"].sum()
9. # Calculate the cumulative frequency by adding the relative frequencies of all the values that are less than or equal to the current value

```

```
10. df["Cumulative Frequency"] = df["Relative Frequency"].  
    cumsum()  
11. # Print the data frame with the relative and cumulative f  
    reQUENCY columns  
12. print(df)  
13. # Create a figure with two subplots  
14. fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))  
15. # Plot a pie chart of the relative frequency of each anim  
    al on the first subplot  
16. ax1.pie(df["Relative Frequency"], labels=df["Animal"], a  
    utopct="%1.1f%%")  
17. ax1.set_title("Pie chart of relative frequency of favorite a  
    nimals")  
18. # Plot a line chart of the cumulative frequency of each an  
    imal on the second subplot  
19. ax2.plot(df["Animal"], df["Cumulative Frequency"], mark  
    er="o", color="red")  
20. ax2.set_title("Line chart of cumulative frequency of favor  
    ite animals")  
21. ax2.set_xlabel("Animal")  
22. ax2.set_ylabel("Cumulative Frequency")  
23. # Show the figure  
24. plt.savefig('relative_cummalative.jpg',dpi=600,bbox_inch  
    es='tight')  
25. plt.show()
```

Output:

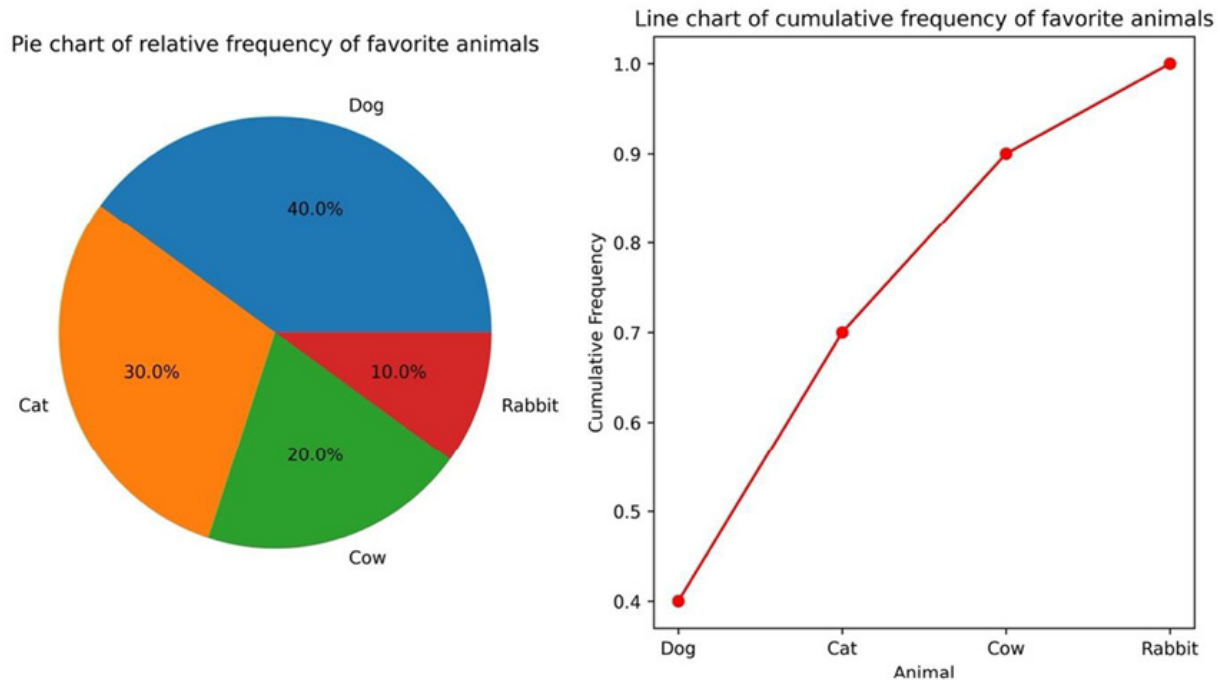


Figure 3.2: Relative frequency in pie chart and cumulative frequency in a line plot

Measure of central tendency

Measure of central tendency is a method to summarize a data set using a single value that represents its center or typical value. This helps us understand the basic features of the data and compare different sets. There are three common measures of central tendency: the mean, the median, and the mode. The average, or mean, is found by adding up all the numbers and then dividing by the total length of numbers. For example, let us say we have five test scores: 80, 85, 90, 95, and 100. To find the mean, we add up all the scores and divide by 5. This gives us the following:

$$(80 + 85 + 90 + 95 + 100) / 5 = 90.$$

The median is the middle number when all the numbers are arranged in order, either from smallest to largest or largest to smallest. To calculate the median, we start by organizing the data and selecting the value in the middle. If the data

set has an even number of values, we average the two middle values. For instance, if there are five test scores, 80, 85, 90, 95, and 100, the median is 90, since it is the third value in the sorted list. If we have six test scores, 80, 85, 90, 90, 95, and 100, the median is the average of 90 and 90, which is also 90. The center number in a set of numbers is called the median. To find the number that appears most often in a set, we count how many times each number appears. In a set of five scores: 80, 85, 80, 95, and 100, the mode is 80 since it appears more than once. However, in a set of six scores: 80, 85, 90, 90, 95, and 100, the mode is 90 since it appears twice, which is more frequent. If all numbers appear the same number of times, there is no mode. We also discussed mean, median, and mode measures in [Chapter 2, Exploratory Data Analysis](#).

Let us recall the measure of central tendency with an example to compute the salary in different regions of Norway, based on the average income by region.

The following table shows the data:

Region	Oslo	South	Mid-Norway	North
Salary (NOK)	57,000	54,000	53,000	50,000

Table 3.4: Average income by region in Norway

To find the middle value, average, and the most frequent value in this set of salaries, we can use the median, mean, and mode, respectively. The mean is the sum of all the salaries divided by 4, which equals $(57,000 + 54,000 + 53,000 + 50,000) / 4 = 53,500$. The two middle numbers are 54,000 and 53,000. We can calculate the median by adding up the four numbers and dividing the sum by 2. As a result, the middle value of the 4 numbers is 53,500. In this case, none of the salaries have the same frequency, hence there is no mode.

Tutorial 3.3: Let us look at an example to compute the

measure of central tendency with a python function. Refer to the following table:

Country	Salary (NOK)
USA	57,000
Norway	54,000
Nepal	50,000
India	50,000
China	50,000
Canada	53,000
Sweden	53,000

Table 3.5: Salary in different countries

Code:

```
1. import pandas as pd
2. import statistics as st
3. # Define a function that takes a data frame as an argument and returns the mean, median, and mode of the salary column
4. def central_tendency(df):
5.     # Calculate the mean, median, and mode of the salary column
6.     mean = df["Salary (NOK)"].mean()
7.     median = df["Salary (NOK)"].median()
8.     mod = st.mode(df["Salary (NOK)"])
9.     # Return the mean, median, and mode as a tuple
10.    return (mean, median, mod)
11. # Create a data frame with the new data
12. data = {"Country": ["USA", "Norway", "Nepal", "India", "China", "Canada", "Sweden"],
13.         "Salary (NOK)": [57000, 54000, 50000, 50000, 50000, 53000, 53000]}
```

```

    00, 53000, 53000]]}
14. df = pd.DataFrame(data)
15. # Call the function and print the results
16. mean, median, mod = central_tendency(df)
17. print(f"The mean of the salary is {mean} NOK.")
18. print(f"The median of the salary is {median} NOK.")
19. print(f"The mode of the salary is {mod} NOK.")

```

Output:

1. The mean of the salary is 52428.57142857143 NOK.
2. The median of the salary is 53000.0 NOK.
3. The mode of the salary is 50000 NOK.

Measures of variability or dispersion

Measures of variability is a measure that show how spread-out data is from the center or scattered a set of data points are. They help to summarize and understand the data better. Simply, measures of variability help you figure out if your data points are tightly packed around the average or spread out over a wider range. Measuring variability or dispersion is important for several reasons as follows:

- It is simpler to compare various data sets thanks to their ability to quantify variability. We can determine that one group of data is more variable or distributed than the other if, for example, the two sets have the same average but different ranges.
- They help determine the form and features of the distribution. For example, a high degree of variation in the data could indicate skewness or outliers. A low degree of variability in the data may indicate that it is normal or symmetric.
- They help in testing hypothesis and using data to guide decisions. For example, when there is little variability in the data, the sample better represents the whole group,

resulting in more comprehensive and reliable conclusions. On the other hand, when there is a high degree of variability, the sample is not as representative of the population, leading to less trustworthy conclusions.

Some common measures of variability or dispersion are, range, variance, standard deviation, interquartile range. Range is the difference between the highest and lowest values in a data. For example, if you have a dataset with numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, the range would be 9 (the difference of the highest and the lowest score).

Tutorial 3.3: An example to compute the range in the data, is as follows:

```
1. # Define a data set as a list of numbers
2. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3. # Find the maximum and minimum values in the data set
4. max_value = max(data)
5. min_value = min(data)
6. # Calculate the range by subtracting the minimum from the maximum
7. range = max_value - min_value
8. # Print the range
9. print("Range:", range)
```

Output:

```
1. Range: 9
```

Interquartile range (IQR) is difference between third and first quartile of a data, which measures spread of the middle 50% of the data. For example, let's compute the IQR of the data set 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

First quartile ($Q1$) = 3.25

Third quartile ($Q3$) = 7.75

Then $IQR = Q3 - Q1 = 7.75 - 3.25 = 4.5$

Tutorial 3.4: An example to compute the interquartile range in data, is as follows:

```
1. import numpy as np
2. # Dataset
3. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4. # Calculate the first quartile (Q1)
5. q1 = np.percentile(data, 25)
6. # Calculate the third quartile (Q3)
7. q3 = np.percentile(data, 75)
8. # Calculate the interquartile range (IQR)
9. iqr = q3 - q1
10. print(f"Interquartile range:: {iqr}")
```

Output:

```
1. Interquartile range: 4.5
```

Variance equals the mean of the squared distances between data points. For example, in a set of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, where the mean is 5.5, the variance would be 8.25.

Tutorial 3.5: An example to compute the interquartile range in data, is as follows:

```
1. import statistics
2. # Define a data set as a list of numbers
3. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4. # Find the mean of the data set
5. mean = statistics.mean(data)
6. # Find the sum of squared deviations from the mean
7. ssd = 0
8. for x in data:
9.     ssd += (x - mean) ** 2
10. # Calculate the variance by dividing the sum of squared
    deviations by the number of values
```

```
11. variance = ssd / len(data)
12. print("Variance:", variance)
```

Output:

```
1. Variance: 8.25
```

Standard deviation is square root of variance which measures how much data points deviate from mean. For example, in a data of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 standard deviation is 2.87.

Tutorial 3.6: An example to compute the standard deviation in data, is as follows:

```
1. # Import math library
2. import math
3. # Define a data set
4. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5. # Find the mean of the data set
6. mean = sum(data) / len(data)
7. # Find the sum of squared deviations from the mean
8. ssd = 0
9. for x in data:
10.     ssd += (x - mean) ** 2
11. # Calculate the variance by dividing the sum of squared
    deviations by the number of values
12. variance = ssd / len(data)
13. # Calculate the standard deviation by taking the square
    root of the variance
14. std = math.sqrt(variance)
15. print("Standard deviation:", std)
```

Output:

```
1. Standard deviation: 2.87
```

Mean deviation is the average of the absolute distances of each value from the mean, median or mode. For example, in

a data of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 the mean deviation is 2.5.

Tutorial 3.7: An example to compute the mean deviation in data, is as follows:

```
1. # Define a data set as a list of numbers
2. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3. # Calculate the mean of the data set
4. mean = sum(data) / len(data)
5. # Calculate the mean deviation by summing the absolute
   differences between each data point and the mean
6. mean_deviation = sum(abs(x - mean) for x in data) / len(
    data)
7. # Print the mean deviation
8. print("Mean Deviation:", mean_deviation)
```

Output:

```
1. Mean Deviation: 2.5
```

Measure of association

Measure of association is used to describe how multiple variables are related to each other. The measure of association varies and depends on the nature and level of measurement of variables. We can measure the relationship between variables by evaluating their strength and direction of association while also determining their independence or dependence through hypothesis testing. Before we go any further, let us understand what hypothesis testing is

Hypothesis testing is used in statistics to investigate ideas about the world. It's often used by scientists to test certain predictions (called hypotheses) that arise from theories. There are two types of hypotheses: null hypotheses and alternative hypotheses. Let us understand them with an example where a researcher want to see, if there is a relationship between gender and height. Then the hypotheses are as follows.

- **Null hypothesis (H_0):** States the prediction that there is no relationship between the variables of interest. So, for the example above, the null hypothesis will be that men are not, on average, taller than women.
- **Alternative hypothesis (H_a or H_1):** Predicts a particular relationship between the variables. So, for the example above, the alternative hypothesis to null hypothesis will be that men are, on average, taller than women.

Continuing measures of association, it can help identify potential causal factors, confounding variables, or moderation effects that impact the outcome in question. Covariance, correlation, chi-squared, Cramer's V, and contingency coefficients, discussed below, are used in statistical analyses to understand the relationships between variables.

To demonstrate the importance of a measure of association, let us take a simple example. Suppose we wish to investigate the correlation between smoking habits and lung cancer. We collect data from a sample of individuals, recording whether or not they smoke and whether or not they have lung cancer. Then, we can employ a measure of association, like the chi-square test (described further below), to ascertain if there is a link between smoking and lung cancer. The chi-square test assesses the extent to which smoking, and lung cancer frequencies observed differ from expected frequencies, assuming their independence. A high chi-square value demonstrates a notable correlation between the variables, while a low chi-square value suggests that they are independent.

For example, suppose we have the following data, and we want to see the effect of smoking in lung cancer:

Smoking	Lung Cancer	No Lung Cancer	Total
Yes	80	20	100

No	20	80	100
Total	100	100	200

Table 3.6: Frequency of patients with and without cancer of the lung and their smoking habits

Based on [Table 3.6](#), we can calculate the observed and expected frequencies for each patient. Using the formula of expected frequency as follows:

$$(E) = (\text{row total} * \text{column total}) / \text{grand total}$$

In [Table 3.6](#), data the expected frequency of the patient where smoking is yes and lung cancer is yes, is given as follows:

$$E = (100 * 100) / 200 = 50$$

Refer to the following [Table 3.7](#), 50 is expected frequency:

Smoking	Lung Cancer	No Lung Cancer	Total
Yes	80 (50)	20 (50)	100
No	20 (50)	80 (50)	100
Total	100	100	200

Table 3.7: Expected frequency of patients

Further to calculate the test statistic, which is the chi-square value. The formula for the chi-square value, is as follows:

$$X^2 = \sum (O-E)^2 / E$$

Here, O is the observed frequency and E is the expected frequency. The sum is taken over all patient in the [Table 3.6](#). For example, the contribution of the [Table 3.6](#) patient where smoking is yes and lung cancer is yes to the chi-square value is as follows:

$$(80-50)^2 / 50 = 18$$

The following table shows the contribution of each patient to the chi-square value:

Smoking	Lung Cancer	No Lung Cancer	Total
Yes	18	18	36
No	18	18	36
Total	36	36	72

Table 3.8: *Chi-square value of patients*

Using an alpha value of 0.05 and a degree of freedom of 1, because we have two categories of smoking (yes or no) so $2-1=1$. the critical value from the chi-square distribution table is 3.841. In this case, the test statistic is 72, which is greater than the critical value of 3.841. Therefore, we reject the null hypothesis and conclude that there is a significant association between smoking and lung cancer. This indicates that smoking is a risk factor for lung cancer, making individuals who smoke more susceptible to developing lung cancer compared to non-smokers. This is a straightforward example of how a measure of association can aid in comprehending the relationship between two variables and drawing conclusions about their causal effects.

Covariance and correlation

Covariance is a method for assessing the link between two things. It displays if those two things change in the same or opposite direction. For example, we can use covariance to explore if taller people weigh more or less than shorter people when we investigate whether height and weight are correlated. Let us look at a simple demonstration of covariance. Consider a group of students who take math and English exams. Calculating the relationship between math scores and English scores can tell us if there is a connection between the two subjects. If the covariance is positive, it means that students who excel in math generally perform well in English, and vice versa. If the covariance is

negative, it suggests that students who excel in math usually struggle in English, and vice versa. If the correlation is zero, there is no direct link between math and English scores.

Let us have a look at the following table:

Student	Math score	English score
A	80	90
B	70	80
C	60	70
D	50	60
E	40	50

Table 3.9: Group of students and their respective grades in Math and English

Use the formula to compute covariance,

$$\text{Covariance} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{n}$$

Where , x_i , and y_i are the individual scores for math and English, $\bar{x}$ and $\bar{y}$ are the mean scores for math and English, and n is the number of students.

Using the data from Table 3.9, the mean ($\bar{x}$) is 60 and the mean ($\bar{y}$) is 70. The sum of the products of paired deviations $\sum (x_i - \bar{x})(y_i - \bar{y})$ is 1000. Finally, the covariance between column maths and English score is calculated to be 250. Which means, there is positive linear relation between a student's math and English scores. Their meaning that as one variable increases, the other variable also tends to increase.

Tutorial 3.8: An example to compute the covariance in data, is as follows:

```

1. import pandas as pd
2. # Define the dataframe as a dictionary
3. df = {"Student": ["A", "B", "C", "D", "E"], "Math Score": [
4.     80, 70, 60, 50, 40], "English Score": [90, 80, 70, 60, 50]}
5. # Convert the dictionary to a pandas dataframe
6. df = pd.DataFrame(df)
7. # Calculate the covariance between math and english scores using the cov method
8. covariance = df["Math Score"].cov(df["English Score"])
9. # Print the result
10. print(f"The covariance between math and english score is {covariance}")

```

Output:

1. The covariance between math and english score is 250.0

Covariance and correlation are similar, but not the same. They both measure the relationship between two variables, but they differ in how they scale and interpret the results.

Following are some key differences between covariance and correlation:

- Covariance can take any value from negative infinity to positive infinity, while correlation ranges from -1 to 1. This means that correlation is a normalized and standardized measure of covariance, which makes it easier to compare and interpret the strength of the relationship.
- Covariance has units, which depend on the units of the two variables. Correlation is dimensionless, which means it has no units. This makes correlation independent of the scale and units of the variables, while covariance is sensitive to them.
- Covariance only indicates the direction of the linear relationship between two variables, such as positive,

negative, or zero. Correlation also indicates the direction, but also the degree of how closely the two variables are related. A correlation of -1 or 1 means a perfect linear relationship, while a correlation of 0 means no linear relationship.

Tutorial 3.9: An example to compute the correlation in the Math and English score data, is as follows:

```
1. import pandas as pd
2. # Create a dictionary with the data
3. data = {"Student": ["A", "B", "C", "D", "E"],
4.         "Math Score": [80, 70, 60, 50, 40],
5.         "English Score": [90, 80, 70, 60, 50]}
6. df = pd.DataFrame(data)
7. # Compute the correlation between the two columns
8. correlation = df["Math Score"].corr(df["English Score"])
9. print("Correlation between math and english score:", correlation)
```

Output:

```
1. Correlation between math and english score: 1.0
```

Chi-square

Chi-square tests if there is a significant connection between two categories. For example, to determine if there is a connection between the music individuals listen to and their emotional state, chi-squared association tests can be used to compare observed frequencies of different moods with different types of music to expected frequencies if there is no relationship between music and mood. The test finds the chi-squared value by adding the squared differences between the observed and expected frequencies and then dividing that sum by the expected frequencies. If the chi-squared value is higher, it suggests a stronger likelihood of a significant connection between the variables.

The next step confirms the significance of the chi-squared value by comparing it to a critical value from a table that considers the degree of freedom and level of significance. If the chi-squared value is higher than the critical value, we will discard the assumption of no relationship.

Tutorial 3.10: An example to show the use of chi-square test to find association between different types of music and mood of a person, is as follows:

```
1. import pandas as pd
2. # Import chi-
   squared test function from scipy.stats module
3. from scipy.stats import chi2_contingency
4. # Create a sample data frame with music and mood cate
   gories
5. data = pd.DataFrame({"Music": ["Rock", "Pop", "Jazz", "
   Classical", "Rap"],
6.                       "Happy": [25, 30, 15, 10, 20],
7.                       "Sad": [15, 10, 20, 25, 30],
8.                       "Angry": [10, 15, 25, 30, 15],
9.                       "Calm": [20, 15, 10, 5, 10]})
10. # Print the original data frame
11. print(data)
12. # Perform chi-square test of association
13. chi2, p, dof, expected = chi2_contingency(data.iloc[:, 1:]
   )
14. # Print the chi-square test statistic, p-
   value, and degrees of freedom
15. print("Chi-square test statistic:", chi2)
16. print("P-value:", p)
17. print("Degrees of freedom:", dof)
18. # Print the expected frequencies
19. print("Expected frequencies:")
```

20. print(expected)

Output:

```
1.      Music Happy Sad Angry Calm
2. 0      Rock    25  15   10   20
3. 1      Pop     30  10   15   15
4. 2      Jazz    15  20   25   10
5. 3 Classical   10  25   30    5
6. 4      Rap     20  30   15   10
7. Chi-square test statistic: 50.070718462823734
8. P-value: 1.3577089704505725e-06
9. Degrees of freedom: 12
10. Expected frequencies:
11. [[19.71830986 19.71830986 18.73239437 11.83098592]
12.  [19.71830986 19.71830986 18.73239437 11.83098592]
13.  [19.71830986 19.71830986 18.73239437 11.83098592]
14.  [19.71830986 19.71830986 18.73239437 11.83098592]
15.  [21.12676056 21.12676056 20.07042254 12.67605634]
    ]
```

The chi-square test results indicate a significant connection between the type of music and the mood of listeners. This suggests that the observed frequencies of different music-mood combinations are not random occurrences but rather signify an underlying relationship between the two variables. A higher chi-square value signifies a greater disparity between observed and expected frequencies. In this instance, the chi-square value is 50.07, a notably large figure. Given that the p-value is less than 0.05, we can reject the null hypothesis and conclude that there is indeed a significant association between music and mood. The degrees of freedom, indicating the number of independent categories in the data, is calculated as $(\text{number of rows} - 1) \times (\text{number of columns} - 1)$, resulting in 12 degrees of freedom in this case. Expected frequencies represent what

would be anticipated under the null hypothesis of no association, calculated by multiplying row and column totals and dividing by the grand total. Comparing observed and expected frequencies reveals the expected distribution if music and mood were independent. Notably, rap and sadness are more frequent than expected (30 vs 21.13), suggesting that rap music is more likely to induce sadness. Conversely, classical and calm are less frequent than expected (5 vs 11.83), indicating that classical music is less likely to induce calmness.

Cramer's V

Cramer's V is a measure of the strength of the association between two categorical variables. It ranges from 0 to 1, where 0 indicates no association and 1 indicates perfect association. Cramer's V and chi-square is related but are different concepts. Cramer's V is an effect size that describes how strongly two variables are related, while chi-square is a test statistic that evaluates whether the observed frequencies are different from the expected frequencies. Cramer's V is based on chi-square, but also takes into account the sample size and the number of categories. Cramer's V is useful for comparing the strength of association between different tables with different numbers of categories. Chi-square can be used to test whether there is a significant association between two nominal variables, but it does not tell us how strong or weak that association is. Cramer's V can be calculated from the chi-squared value and the degrees of freedom of the contingency table.

$$\text{Cramer's } V = \sqrt{(X^2/n) / \min(c-1, r-1)}$$

Where:

- **X²**: The Chi-square statistic
- **n**: Total sample size
- **r**: Number of rows

- **c**: Number of columns

For example, Cramer's V is to compare the association between gender and eye color in two different populations. Suppose we have the following data:

Population	Gender	Eye color	Frequency
A	Male	Blue	10
A	Male	Brown	20
A	Female	Blue	15
A	Female	Brown	25
B	Male	Blue	5
B	Male	Brown	25
B	Female	Blue	25
B	Female	Brown	5

Table 3.10: Gender and eye color in two different populations

Tutorial 3.11: An example to illustrate the use of Cramer's V to measure the strength of the association between gender and eye color in each population, is as follows:

1. `import pandas as pd`
2. `# Importing necessary functions from the scipy.stats module`
3. `from scipy.stats import chi2_contingency, chi2`
4. `# Create a dataframe from the given data`
5. `df = pd.DataFrame({"Population": ["A", "A", "A", "A", "B", "B", "B", "B"],`
`"Gender": ["Male", "Male", "Female", "Female", "Male", "Male", "Female", "Female"],`
`"Eye Color": ["Blue", "Brown", "Blue", "Brown", "Blue", "Brown", "Blue", "Brown"]},`

```
8.         "Frequency": [10, 20, 15, 25, 5, 25, 25, 5])
9. # Pivot the dataframe to get a contingency table
10. table = pd.pivot_table(
11.     df, index=
12.     ["Population", "Gender"], columns="Eye Color", values=
13.     "Frequency")
14. # Print the table
15. print(table)
16. # Perform chi-square test for each population
17. for pop in ["A", "B"]:
18.     # Subset the table by population
19.     subtable = table.loc[pop]
20.     # Calculate the chi-square statistic, p-
21.     # value, degrees of freedom, and expected frequencies
22.     chi2_stat, p_value, dof, expected = chi2_contingency(s
23.     ubtable)
24.     # Print the results
25.     print(f"\nChi-square test for population {pop}:")
26.     print(f"Chi-square statistic = {chi2_stat:.2f}")
27.     print(f"P-value = {p_value:.4f}")
28.     print(f"Degrees of freedom = {dof}")
29.     print(f"Expected frequencies:")
30.     print(expected)
31. # Calculate Cramer's V for population B and population
32. A
33. # Cramer's V is the square root of the chi-
34. # square statistic divided by the sample size and the mini
35. # mum of the row or column dimensions minus one
36. n = df["Frequency"].sum() # Sample size
37. k = min(table.shape) - 1 # Minimum of row or column d
38. # imensions minus one
39. # Chi-square statistic for population B
```

```

32. chi2_stat_B = chi2_contingency(table.loc["B"])[0]
33. # Chi-square statistic for population A
34. chi2_stat_A = chi2_contingency(table.loc["A"])[0]
35. cramers_V_B = (chi2_stat_B / (n * k)) ** 0.5 # Cramer's
    V for population B
36. cramers_V_A = (chi2_stat_A / (n * k)) ** 0.5 # Cramer's
    V for population A
37. # Print the results
38. print(f"\nCramer's V for population B and population A:"
    )
39. print(f"Cramer's V for population B = {cramers_V_B:.2f}"
    )
40. print(f"Cramer's V for population A = {cramers_V_A:.2f}"
    )

```

Output:

```

1. Eye Color      Blue  Brown
2. Population Gender
3. A      Female   15    25
4.      Male     10    20
5. B      Female   25     5
6.      Male      5    25
7.
8. Chi-square test for population A:
9. Chi-square statistic = 0.01
10. P-value = 0.9140
11. Degrees of freedom = 1
12. Expected frequencies:
13. [[14.28571429 25.71428571]
14.  [10.71428571 19.28571429]]
15.
16. Chi-square test for population B:
17. Chi-square statistic = 24.07

```

18. P-value = 0.0000
19. Degrees of freedom = 1
20. Expected frequencies:
21. [[15. 15.]
22. [15. 15.]]
- 23.
24. Cramer's V for population B and population A:
25. Cramer's V for population B = 0.43
26. Cramer's V for population A = 0.01

Above data shows the frequencies of eye color by gender and population for two populations, A and B. Here, the chi-square test is used to test whether there is a significant association between gender and eye color in each population. The null hypothesis is that there is no association, and the alternative hypothesis is that there is an association. The p-value is the probability of obtaining the observed or more extreme results under the null hypothesis. A small p-value (usually less than 0.05) indicates strong evidence against the null hypothesis, and a large p-value (usually greater than 0.05) indicates weak evidence against the null hypothesis. The results show that for population A, the p-value is 0.9140, which is very large. This means that we fail to reject the null hypothesis and conclude that there is no significant association between gender and eye color in population A. The chi-square statistic is 0.01, which is very small and indicates that the observed frequencies are very close to the expected frequencies under the null hypothesis. The expected frequencies are 14.29 and 25.71 for blue and brown eyes respectively for females, and 10.71 and 19.29 for blue and brown eyes respectively for males. The results show that for population B, the p-value is 0.0000, which is very small. This means that we reject the null hypothesis and conclude that there is a significant association between gender and eye

color in population B. The chi-square statistic is 24.07, which is very large and indicates that the observed frequencies are very different from the expected frequencies under the null hypothesis. The expected frequencies are 15 and 15 for both blue and brown eyes for both females and males.

Since Cramer's V is a measure of the strength of the association between two categorical variables based on the chi-squared statistic and sample size.

The results show that Cramer's V for population B is 0.43, which indicates a moderate association between gender and eye color. Cramer's V for population A is 0.01, which indicates a very weak association between gender and eye color. This confirms the results of the chi-square test.

Contingency coefficient

The contingency coefficient is a measure of association in statistics that indicates whether two variables or data sets are independent or dependent on each other. It is also known as Pearson's coefficient.

The contingency coefficient is based on the chi-square statistic and is defined by the following formula:

$$C = \sqrt{\frac{\chi^2}{\chi^2 + N}}$$

Where:

- χ^2 is the chi-square statistic
- N is the total number of cases or observations in our analysis/study.
- C is the contingency coefficient

The contingency coefficient can range from 0 (no association) to 1 (perfect association). If C is close to zero (or equal to zero), you can conclude that your variables are independent of each other; there is no association between them. If C is away from zero, there is some association. Contingency coefficient is important because it can help us

summarize the relationship between two categorical variables in a single number. It can also help us compare the degree of association between different tables or groups.

Tutorial 3.12: An example to measure the association between two categorical variables gender and product using contingency coefficient, is as follows:

```
1. import pandas as pd
2. from scipy.stats import chi2_contingency
3. # Create a simple dataframe
4. data = {'Gender': ['Male', 'Female', 'Female', 'Male', 'Male', 'Female'],
5.         'Product': ['Product A', 'Product B', 'Product A', 'Product A', 'Product B', 'Product B']}
6. df = pd.DataFrame(data)
7. # Create a contingency table
8. contingency_table = pd.crosstab(df['Gender'], df['Product'])
9. # Perform Chi-Square test
10. chi2, p, dof, expected = chi2_contingency(contingency_table)
11. # Calculate the contingency coefficient
12. contingency_coefficient = (chi2 / (chi2 + df.shape[0])) ** 0.5
13. print('Contingency Coefficient is:', contingency_coefficient)
```

Output:

```
1. Contingency Coefficient is: 0.0
```

In this case, the contingency coefficient is 0 which shows there is no association at all between gender and product.

Tutorial 3.13: Similarly, as shown in [Table 3.9](#), if we want to know whether gender and eye color are related in two

different populations, we can calculate the contingency coefficient for each population and see which one has a higher value. A higher value indicates a stronger association between the variables.

Code:

```
1. import pandas as pd
2. from scipy.stats import chi2_contingency
3. import numpy as np
4. df = pd.DataFrame({"Population": ["A", "A", "A", "A", "B",
    "B", "B", "B"],
5.                    "Gender": ["Male", "Male", "Female", "Female", "Male", "Male", "Female", "Female"],
6.                    "Eye Color": ["Blue", "Brown", "Blue", "Brown",
    "Blue", "Brown", "Blue", "Brown"],
7.                    "Frequency": [10, 20, 15, 25, 5, 25, 25, 5]})
8. # Create a pivot table
9. pivot_table = pd.pivot_table(df, values='Frequency', index=[
10.    'Population', 'Gender'], columns=
    ['Eye Color'], aggfunc=np.sum)
11. # Calculate chi-square statistic
12. chi2, _, _, _ = chi2_contingency(pivot_table)
13. # Calculate the total number of observations
14. N = df['Frequency'].sum()
15. # Calculate the Contingency Coefficient
16. C = np.sqrt(chi2 / (chi2 + N))
17. print(f"Contingency Coefficient: {C}")
```

Output:

```
1. Contingency Coefficient: 0.43
```

This gives contingency coefficient 0.4338. Which indicates that there is a moderate association between the variables

in the above data (population, gender, and eye color). This means that knowing the category of one variable gives some information about the category of the other variables. However, the association is not very strong because the coefficient is closer to 0 than to 1. Furthermore, the contingency coefficient has some limitations, such as being affected by the size of the table and not reaching 1 for perfect association. Therefore, some alternative measures of association, such as Cramer's V or the phi coefficient, may be preferred in some situations.

Measures of shape

Measures of shape are used to describe the general shape of a distribution, including its symmetry, skewness, and kurtosis. These measures help to give a sense of how the data is spread out, and can be useful for identifying potentially outlier observations or data points. For example, imagine you are a teacher, and you want to evaluate your students' performance on a recent math test. Here the skewness tells you distribution of the scores. If the scores are more spread out on one side of the mean than the other, and kurtosis tells you how peaked or flattened the distribution of scores is.

Skewness

Skewness measures the degree of asymmetry in a distribution. A distribution is symmetrical if the two halves on either side of the mean are mirror images of each other. Positive skewness indicates that the right tail of the distribution is longer or thicker than the left tail, while negative skewness indicates the opposite.

Tutorial 3.14: Let us consider a class of 10 students who recently took a math test. Their scores (out of 100) are as follows, and based on these scores we can see the skewness of the students' scores, whether they are positively skewed

(toward high scores) or negatively skewed (toward low scores).

Refer to the following table:

Student ID	1	2	3	4	5	6	7	8	9	10
Score	85	90	92	95	96	96	97	98	99	100

Table 3.11: Students and their respective scores

Code:

```
1. import matplotlib.pyplot as plt
2. import seaborn as sns
3. from scipy.stats import skew
4. data = [85, 90, 92, 95, 96, 96, 97, 98, 99, 100]
5. # Calculate skewness
6. data_skewness = skew(data)
7. # Create a combined histogram and kernel density plot
8. plt.figure(figsize=(8, 6))
9. sns.histplot(data, bins=10, kde=True, color='skyblue', edgecolor='black')
10. # Add skewness information
11. plt.xlabel('Score')
12. plt.ylabel('Count')
13. plt.title(f'Skewness: {data_skewness:.2f}')
14. # Show the figure
15. plt.savefig('skew_negative.jpg', dpi=600, bbox_inches='tight')
16. plt.show()
```

Output: *Figure 3.3* shows negative skew:

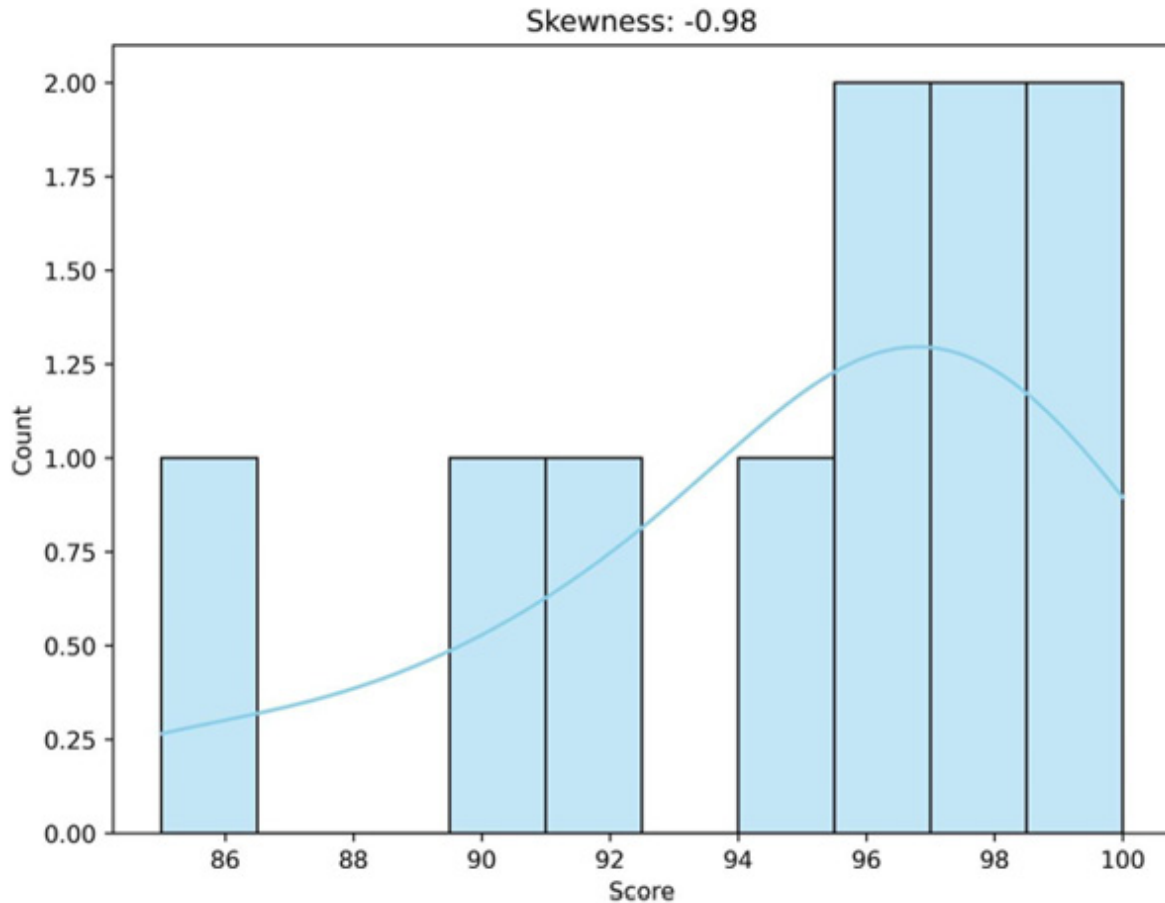


Figure 3.3: Negative skewness

The given data, exhibiting a skewness of -0.98, is negatively skewed. The graphical representation indicates that the distribution of students' scores is not symmetrical. The majority of scores are concentrated to the left, while fewer scores are concentrated to the right. This is an example of **negative skewness**, also known as left skew. In a negatively skewed distribution, the mean is smaller than the median, and the left tail (smaller numbers) is longer or thicker than the right tail. In this scenario, the teacher can deduce that most students scored below the average on the test, with very few scorings above the average. This could suggest that the test was challenging or that the class faces difficulties in the subject matter. Remember that skewness is only one aspect of understanding the distribution of data.

It is also important to consider other factors, such as kurtosis, standard deviation, etc., for a more complete understanding.

Tutorial 3.15: An example to view the positive skewness of data, is as follows:

```
1. import matplotlib.pyplot as plt
2. import seaborn as sns
3. from scipy.stats import skew
4. data = [115, 120, 85, 90, 92, 95, 96, 96, 97, 98]
5. # Calculate skewness
6. data_skewness = skew(data)
7. # Create a combined histogram and kernel density plot
8. plt.figure(figsize=(8, 6))
9. sns.histplot(data, bins=10, kde=True, color='skyblue', edgecolor='black')
10. # Add skewness information
11. plt.xlabel('Score')
12. plt.ylabel('Count')
13. plt.title(f'Skewness: {data_skewness:.2f}')
14. # Display the plot
15. plt.savefig('skew_positive.jpg', dpi=600, bbox_inches='tight')
16. plt.show()
```

Output: *Figure 3.4* shows positive skew:

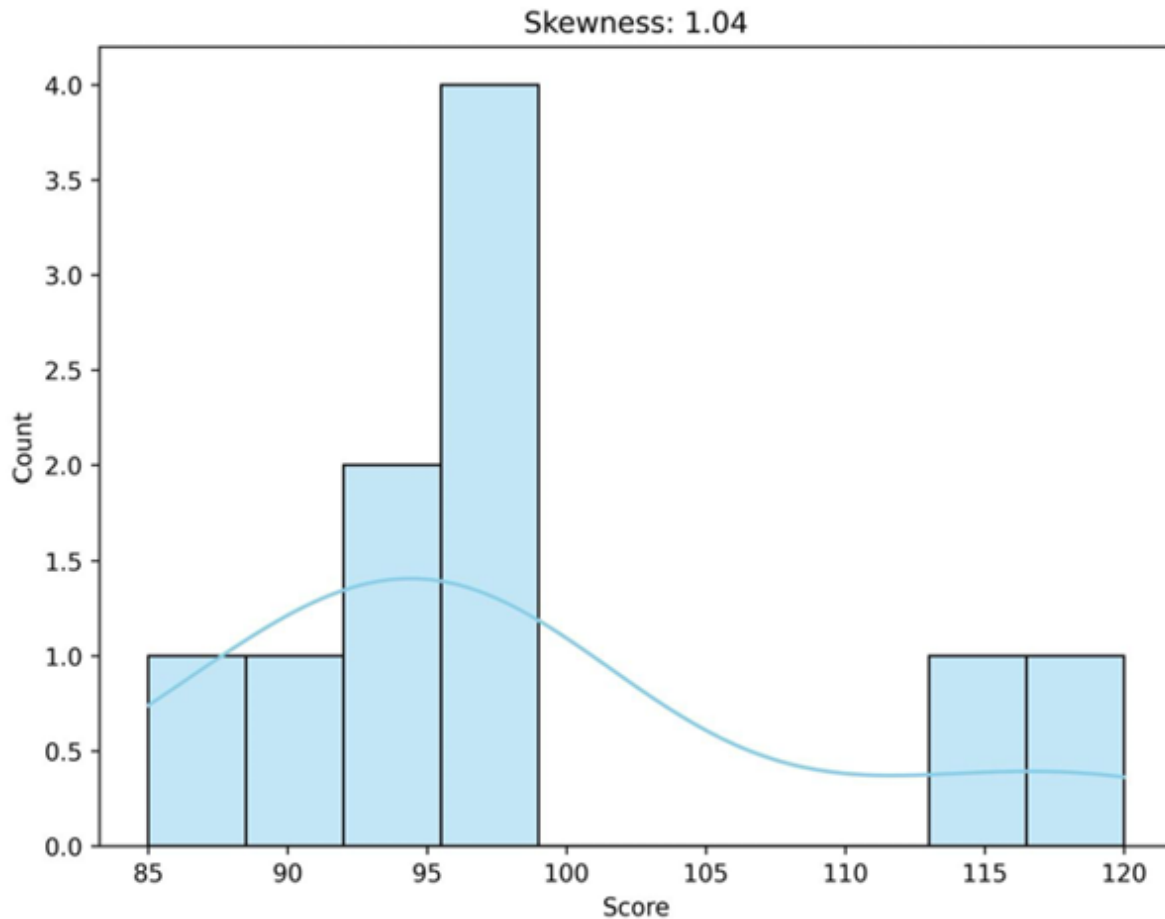


Figure 3.4: Positive skewness

Tutorial 3.16: An example to show the symmetrical distribution, positive and negative skewness of data respectively in a subplot, is as follows:

1. `import numpy as np`
2. `import matplotlib.pyplot as plt`
3. `from scipy.stats import skew`
4. *# Define the three datasets*
5. `data1 = np.array([1, 2, 3, 4, 5, 5, 4, 3, 2, 1])`
6. `data2 = np.array([2, 3, 4, 5, 6, 7, 8, 9, 10, 20])`
7. `data3 = np.array([20, 15, 10, 9, 8, 7, 6, 5, 4, 2])`
8. *# Calculate skewness for each dataset*
9. `skewness1 = skew(data1)`

```
10. skewness2 = skew(data2)
11. skewness3 = skew(data3)
12. # Plot the data and skewness in subplots
13. fig, axes = plt.subplots(1, 3, figsize=(12, 8))
14. # Subplot 1
15. axes[0].plot(data1, marker='o', linestyle='-')
16. axes[0].set_title(f'Data 1\nSkewness: {skewness1:.2f}')
17. # Subplot 2
18. axes[1].plot(data2, marker='o', linestyle='-')
19. axes[1].set_title(f'Data 2\nSkewness: {skewness2:.2f}')
20. # Subplot 3
21. axes[2].plot(data3, marker='o', linestyle='-')
22. axes[2].set_title(f'Data 3\nSkewness: {skewness3:.2f}')
23. # Adjust layout
24. plt.tight_layout()
25. # Display the plot
26. plt.savefig('skew_all.jpg', dpi=600, bbox_inches='tight')
27. plt.show()
```

Output:

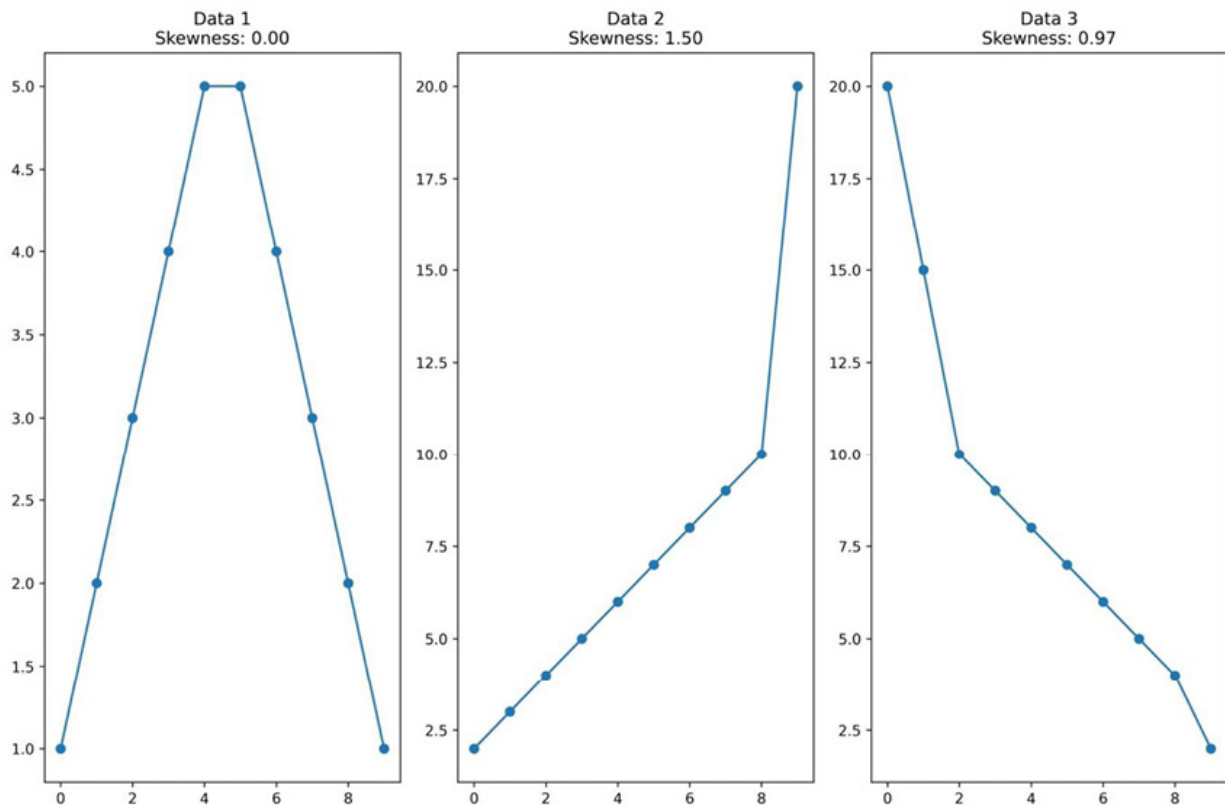


Figure 3.5: Symmetrical distribution, positive and negative skewness of data

Tutorial 3.17: An example to measure skewness in diabetes dataset data frame **Age** column using plot, is as follows:

```
1. import pandas as pd
2. import matplotlib.pyplot as plt
3. import seaborn as sns
4. from scipy.stats import skew
5. diabeties_df = pd.read_csv(
6.     '/workspaces/ImplementingStatisticsWithPython/data
7.     /chapter1/diabetes.csv')
8. data = diabeties_df['Age']
9. # Calculate skewness
10. data_skewness = skew(data)
11. # Create a combined histogram and kernel density plot
12. plt.figure(figsize=(8, 6))
13. sns.histplot(data, bins=10, kde=True, color='skyblue', e
```

```
dgecolor='black')
13. # Add skewness information
14. plt.title(f'Skewness: {data_skewness:.2f}')
15. # Display the plot
16. plt.savefig('skew_age.jpg', dpi=600, bbox_inches='tight')
17. plt.show()
```

Output:

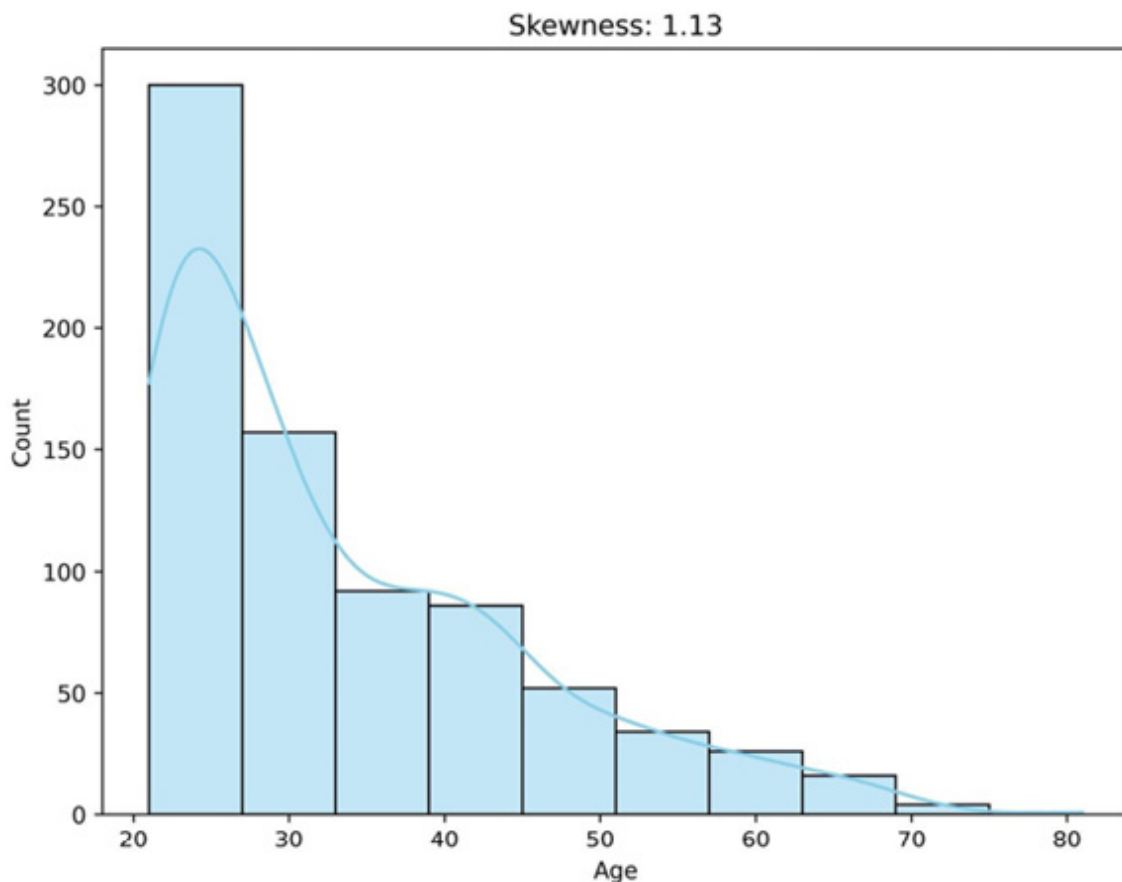


Figure 3.6: Positive skewness in diabetes dataset Age column

Kurtosis

Kurtosis measures the tilt of a distribution (that is, the concentration of values at the tails). It indicates whether the tails of a given distribution contain extreme values. If you think of a data distribution as a mountain, the kurtosis

would tell you about the shape of the peak and the tails. A high kurtosis means that the data has heavy tails or outliers. In other words, the data has a high peak (more data in the middle) and fat tails (more extreme values). This is called a **leptokurtic distribution**. Low kurtosis in a data set is an indicator that the data has light tails or lacks outliers. The data points are moderately spread out (less in the middle and less extreme values), which means it has a flat peak. This is called a **platykurtic distribution**. A normal distribution has zero kurtosis. Understanding the kurtosis of a data set helps to identify volatility, risk, or outlier detection in various fields such as finance, quality control, and other statistical modeling where data distribution plays a key role.

Tutorial 3.15: An example to understand how viewing the Kurtosis of a dataset helps in identifying the presence of outliers.

Let us look at three different data sets, as follows:

- **Dataset A:** [1, 1, 2, 2, 3, 3, 4, 4, 9, 9] - This dataset has a few extreme values (9).
- **Dataset B:** [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] - This dataset has no extreme values and is evenly distributed.
- **Dataset C:** [1, 2, 2, 3, 3, 3, 4, 4, 4, 4] - This data set has more values around the mean (3 and 4).

Let us calculate the kurtosis for these data sets.

Code:

```
1. import scipy.stats as stats
2. # Datasets
3. dataset_A = [1, 1, 2, 2, 3, 3, 4, 4, 4, 30]
4. dataset_B = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5. dataset_C = [1, 2, 3, 3, 3, 3, 3, 3, 4, 5]
6. # Calculate kurtosis
7. kurtosis_A = stats.kurtosis(dataset_A)
```

```
8. kurtosis_B = stats.kurtosis(dataset_B)
9. kurtosis_C = stats.kurtosis(dataset_C)
10. print(f"Kurtosis of Dataset A: {kurtosis_A}")
11. print(f"Kurtosis of Dataset B: {kurtosis_B}")
12. print(f"Kurtosis of Dataset C: {kurtosis_C}")
```

Output:

```
1. Kurtosis of Dataset A: 4.841818043320611
2. Kurtosis of Dataset B: -1.2242424242424244
3. Kurtosis of Dataset C: 0.3999999999999999
```

Here we see, in data set A: $[1, 1, 2, 2, 3, 3, 4, 4, 4, 30]$ has a kurtosis of 4.84. This is a high positive value, indicating that the data set has heavy tails and a sharp peak. This means that there are more extreme values in the data set, as indicated by the value 30. This is an example of a leptokurtic distribution. In the data set B: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ has a kurtosis of -1.22. This is a negative value, indicating that the data set has light tails and a flat peak. This means that there are fewer extreme values in the data set and the values are evenly distributed. This is an example of a platykurtic distribution. The data set C: $[1, 2, 3, 3, 3, 3, 3, 3, 4, 5]$ has a kurtosis of 0.4, which is close to zero. This indicates that the data set has a distribution shape similar to a normal distribution (mesokurtic). The values are somewhat evenly distributed around the mean, with a balance between extreme values and values close to the mean.

Conclusion

Descriptive statistics is a branch of statistics that organizes, summarizes, and presents data in a meaningful way. It uses different types of measures to describe various aspects of the data. For example, measures of frequency, such as relative and cumulative frequency, frequency tables and

distribution, help to understand how many times each value of a variable occurs and what proportion it represents in the data. Measures of central tendency, such as mean, median, and mode, help to find the average or typical value of the data. Measures of variability or dispersion, such as range, variance, standard deviation, and interquartile range, help to measure how much the data varies or deviates from the center. Measures of association, such as correlation and covariance, help to examine how two or more variables are related to each other. Finally, measures of shape, such as skewness and kurtosis, help to describe the symmetry and the heaviness of the tails of a probability distribution. These methods are vital in descriptive statistics because they give a detailed summary of the data. This helps us understand how the data behaves, find patterns, and make knowledgeable choices. They are fundamental for additional statistical analysis and hypothesis testing.

In *Chapter 4: Unravelling Statistical Relationships* we will see more about the statistical relationship and understand the meaning and implementation of covariance, correlation and probability distribution.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Unravelling Statistical Relationships

Introduction

Understanding the connection between different variables is part of unravelling statistical relationships. Covariance and correlation, outliers and probability distributions are critical to the unravelling of statistical relationships and make accurate interpretations based on data. Covariance and correlation essentially measure the same concept, the change in two variables with respect to each other. They aid in comprehending the relationship between two variables in a dataset and describe the extent to which two random variables or random variable sets are prone to deviate from their expected values in the same manner. Covariance illustrates the degree to which two random variables vary together. And correlation is a mathematical method for determining the degree of statistical dependence between two variables. Ranging from -1 (perfect negative correlation) to $+1$ (perfect positive correlation). Statistical relationships are based on data and most data contains outliers. Outliers are observations that are significantly different from other data points, such as data variability or

experimental errors. Such outliers can significantly skew data analysis and statistical modeling, potentially leading to erroneous conclusions. Therefore, it is essential to identify and manage outliers to ensure accurate results. To facilitate comprehension and prediction of data patterns measuring likelihood and distribution of likelihood is required. For these statisticians use probability and probability distribution. The probability measures the likelihood of a specific event occurring and is denoted by a value between 0 and 1, where 0 implies impossibility and 1 signifies certainty.

A probability distribution which is a mathematical function describes how probabilities are spread out over the values of a random variable. For instance, in a fair roll of a six-sided dice, the probability distribution would indicate that each outcome (1, 2, 3, 4, 5, 6) has a probability of $1/6$. While probability measures the likelihood of a single event, a probability distribution considers all potential events and their respective probabilities. It offers a comprehensive view of the randomness or variability of a particular data set. Sometimes there can be many data point or large data that need to be represented as one. In such case the data points in the form of arrays and matrices allow us to explore statistical relationships, distinguish true correlations from spurious ones, and visualize complex dependencies in data. All of these concepts in the structure below are basic, but very important steps in unraveling and understanding the statistical relationship.

Structure

In this chapter, we will discuss the following topics:

- Covariance and correlation
- Outliers and anomalies
- Probability

- Array and matrices

Objectives

By the end of this chapter, readers will see what covariance, correlation, outliers, anomalies are, how they affect data analysis, statistical modeling, and learning, how they can lead to misleading conclusions, and how to detect and deal with them. We will also look at probability concepts and the use of probability distributions to understand data, its distribution, and its properties, how they can help in making predictions, decisions, and estimating uncertainty.

Covariance

Covariance in statistics measures how much two variables change together. In other words, it is a statistical tool that shows us how much two numbers vary together. A positive covariance indicates that the two variables tend to increase or decrease together. Conversely, a negative covariance indicates that as one variable increases, the other tends to decrease and vice versa. Covariance and correlation are important in measuring association, as discussed in [Chapter 3, Frequency Distribution, Central Tendency, Variability](#). While correlation is limited to -1 to +1, covariance can be practically any number. Now, let us consider a simple example.

Suppose you are a teacher with a class of students. And you observed when the temperature is high in the summer, the students' test scores generally decrease, while in the winter when it is low, the scores tend to rise. This is a negative covariance because as one variable, temperature, goes up, the other variable, test scores, goes down. Similarly, if students who study more hours tend to have higher test scores, this is a positive covariance. As study hours increase, test scores also increase. Covariance helps

identify the relationship between different variables.

Tutorial 4.1: An example to calculate the covariance between temperature and test scores, and between study hours and test scores, is as follows:

```
1. import numpy as np
2. # Let's assume these are the temperatures in Celsius
3. temperatures = np.array([30, 32, 28, 31, 33, 29, 34, 35,
                           36, 37])
4. # And these are the corresponding test scores
5. test_scores = np.array([70, 68, 72, 71, 67, 73, 66, 65, 64,
                          63])
6. # And these are the corresponding study hours
7. study_hours = np.array([5, 6, 7, 6, 5, 7, 4, 3, 2, 1])
8. # Calculate the covariance between temperature and test scores
9. cov_temp_scores = np.cov(temperatures, test_scores)
   [0, 1]
10. print(f"Covariance between temperature and test scores
        : {cov_temp_scores}")
11. # Calculate the covariance between study hours and test scores
12. cov_study_scores = np.cov(study_hours, test_scores)
   [0, 1]
13. print(f"Covariance between study hours and test scores:
        {cov_study_scores}")
```

Output:

1. Covariance between temperature and test scores: -10.277777777777777
2. Covariance between study hours and test scores: 6.733333333333334

As output shows, covariance between temperature and test score is negative (indicating that as temperature increases,

test scores decrease), and the covariance between study hours and test scores is positive (indicating that as study hours increase, test scores also increase).

Tutorial 4.2: Following is an example to calculate the covariance in a data frame, here we only compute covariance of selected three columns from the diabetes dataset:

```
1. # Import the pandas library and the display function
2. import pandas as pd
3. from IPython.display import display
4. # Load the diabetes dataset csv file
5. diabetics_df = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
6. diabetics_df[['Glucose','Insulin','Outcome']].cov()
```

Output:

```
1.      Glucose   Insulin   Outcome
2. Glucose 1022.248314 1220.935799  7.115079
3. Insulin 1220.935799 13281.180078  7.175671
4. Outcome  7.115079   7.175671   0.227483
```

The diagonal elements (1022.24 for glucose, 13281.18 for insulin, and 0.22 for outcome) represent the variance of each variable. Looking at glucose its variance is 1022.24, which means that glucose levels vary quite a bit and insulin varies even more. Covariance between glucose and insulin is a positive number, which means that high glucose levels tend to be associated with high insulin levels and vice versa, and the covariance between insulin and outcome is 7.17. Since, these are positive numbers, this means that high glucose and insulin levels tend to be associated with high outcome and vice versa.

While covariance is a powerful tool for understanding relationships in numerical data, other techniques are typically more appropriate for text and image data. For

example, **term frequency-inverse document frequency (TF-IDF)**, cosine similarity, or word embeddings (such as **Word2Vec**) are often used to understand relationships and variations in text data. For image data, **convolutional neural networks (CNNs)**, image histograms, or feature extraction methods are used.

Correlation

Correlation in statistics measures the magnitude and direction of the connection between two or more variables. It is important to note that correlation does not imply causality between the variables. The correlation coefficient assigns a value to the relationship on a -1 to 1 scale. A **positive** correlation, closer to 1, indicates that as one variable increases, so does the other. Conversely, a **negative** correlation, closer to -1 means that as one variable increases, the other decreases. A correlation of zero suggests no association between two variables. More about correlation is also discussed in [Chapter 1, Introduction to Statistics and Data](#), and [Chapter 3, Frequency Distribution, Central Tendency, Variability](#). Remember that while covariance and correlation are related correlation provides a more interpretable measure of association, especially when comparing variables with different units of measurement.

Let us understand correlation with an example, consider relationship between study duration and exam grade. If students who spend more time studying tend to achieve higher grades, we can conclude that there is a positive correlation between study time and exam grades, as an increase in study time corresponds to an increase in exam grades. On the other hand, an analysis of the correlation between the amount of time devoted to watching television and test scores reveals a negative correlation. Specifically,

as the duration of television viewing (one variable) increases, the score on the exam (the other variable) drops. Bear in mind that correlation does not necessarily suggest causation. Mere correlation between two variables does not reveal a cause-and-effect relationship.

Tutorial 4.3: An example to calculate the correlation between study time and test scores, and between TV watching time and test scores, is as follows:

```
1. import numpy as np
2. # Let's assume these are the study hours
3. study_hours = np.array([5, 6, 7, 6, 5, 7, 4, 3, 2, 1])
4. # And these are the corresponding test scores
5. test_scores = np.array([70, 72, 75, 72, 70, 75, 68, 66, 64, 62])
6. # And these are the corresponding TV watching hours
7. tv_hours = np.array([1, 2, 1, 2, 3, 1, 4, 5, 6, 7])
8. # Calculate the correlation between study hours and test scores
9. corr_study_scores = np.corrcoef(study_hours, test_scores)[0, 1]
10. print(f"Correlation between study hours and test scores: {corr_study_scores}")
11. # Calculate the correlation between TV watching hours and test scores
12. corr_tv_scores = np.corrcoef(tv_hours, test_scores)[0, 1]
13. print(
14.     f"Correlation between TV watching hours and test scores: {corr_tv_scores}")
```

Output:

1. Correlation between study hours and test scores: 0.9971289059323629
2. Correlation between TV watching hours and test scores: -0.9495412844036697

Output shows an increase in study hours correspond to a higher test score, indicating a positive correlation. A negative correlation is between the number of hours spent watching television and test scores. This suggests that an increase in TV viewing time is linked to a decline in test scores.

Outliers and anomalies

Outlier is a data point that significantly differs from other observations. It is a value that lies at an abnormal distance from other values in a random sample from a population.

Anomalies, are similar to outliers as they are values in a data set that do not fit the expected behavior or pattern of the data. The terms outliers and anomalies are often used interchangeably in statistics, they can have slightly different connotations depending on the context. For example, let us say you are a teacher and you are looking at the test scores of your students. Most of the scores are between 70 and 90, but there is one score that is 150. This score would be considered an outlier because it is significantly higher than the rest of the scores. It is also an anomaly because it does not fit the expected pattern (since test scores usually range from 0 to 100). Another example is, in a dataset of human ages, a value of 150 would be an outlier because it is significantly higher than expected. However, if you have a sequence of credit card transactions and you suddenly see a series of very high-value transactions from a card that usually only has small transactions, that would be an anomaly. The individual transaction amounts might not be outliers by themselves, but the sequence or pattern of transactions is unusual given the past behavior of the card.

So, while all outliers could be considered anomalies (because they are different from the norm), not all anomalies are outliers (because they might not be extreme values, but rather unexpected pattern or behavior).

Tutorial 4.4: An example to calculates the concept of outliers and anomalies, is as follows:

```
1. import numpy as np
2. from scipy import stats
3. import matplotlib.pyplot as plt
4. # Let's assume these are the ages of a group of people
5. ages = np.array([20, 25, 30, 35, 40, 45, 50, 55, 60, 150]
6. )
7. # Now let's consider a sequence of credit card transactions
8. transactions = np.
9. array([100, 120, 150, 110, 105, 102, 108, 2000, 2100, 220
10. 0])
11.
12. # Define a function to detect outliers using the Z-score
13. def detect_outliers(data):
14.     outliers = []
15.     threshold = 1
16.     mean = np.mean(data)
17.     std = np.std(data)
18.     for i in data:
19.         z_score = (i - mean) / std
20.         if np.abs(z_score) > threshold:
21.             outliers.append(i)
22.     return outliers
23. Unravelling Statistical Relationships v 141
24.
25. # Define a function to detect anomalies based on sudden increase in transaction amounts
26. def detect_anomalies(data):
27.     anomalies = []
```

```

24. threshold = 1.5 # this could be any value based on your understanding of the data
25. mean = np.mean(data)
26. for i in range(len(data)):
27.     if i == 0:
28.         continue # skip the first transaction
29.     # if the current transaction is more than twice the previous one
30.     if data[i] > threshold * data[i-1]:
31.         anomalies.append(data[i])
32.     return anomalies
33.
34. anomalies = detect_anomalies(transactions)
35. print(f"Anomalies in transactions: {anomalies}")
36. outliers = detect_outliers(ages)
37. print(f"Outliers in ages: {outliers}")
38. # Plot ages with outliers in red
39. fig, (axs1, axs2) = plt.subplots(2, figsize=(15, 8))
40. axs1.plot(ages, 'bo')
41. axs1.plot([i for i, x in enumerate(ages) if x in outliers],
42.            [x for x in ages if x in outliers], 'ro')
43. axs1.set_title('Ages with Outliers')
44. axs1.set_ylabel('Age')
45. # Plot transactions with anomalies in red
46. axs2.plot(transactions, 'bo')
47. axs2.plot([i for i, x in enumerate(transactions) if x in anomalies],
48.            [x for x in transactions if x in anomalies], 'ro')
49. axs2.set_title('Transactions with Anomalies')

```

```

50. axs2.set_ylabel('Transaction Amount')
51. plt.savefig('outliers_anomalies.jpg', dpi=600, bbox_inches='tight')
52. plt.show()

```

In this program, we define two numpy arrays: **ages** and **transactions**, which represent the collected data. Two functions, **detect_outliers** and **detect_anomalies**, are then defined. The **detect_outliers** function uses the z-score method to identify outliers in the ages data. Likewise, the **detect_anomalies** function identifies anomalies in the transaction data based on a sudden increase in transaction amounts.

Output:

1. Anomalies in transactions: [2000]
2. Outliers in ages: [150]

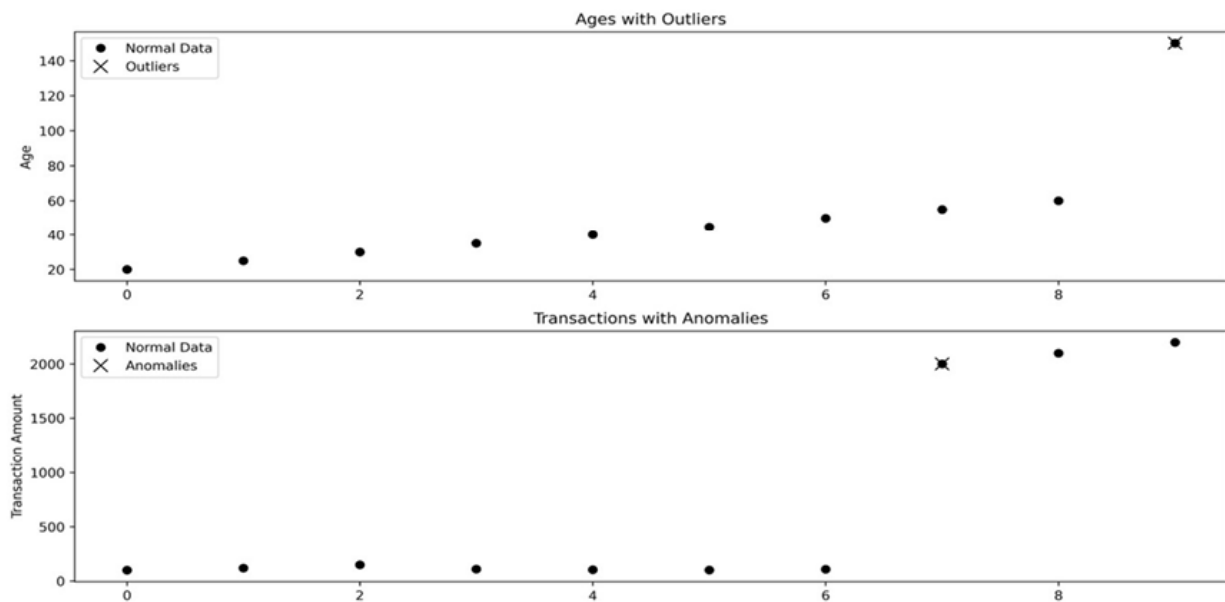


Figure 4.1: Subplots showing outliers in age and anomalies in transaction

The **detect_outliers** function identifies the age of 150 as an outlier, while the **detect_anomalies** function recognizes the transactions of 2000 as anomalies. marking a change in pattern with cross(x).

For textual data, an outlier could be a document or text entry that is considerably lengthier or shorter compared to the other entries in the dataset. An anomaly could occur when there is a sudden shift in the topic or sentiment of texts in a particular time series, or the use of uncommon words or phrases. For image data, an outlier could be an image that differs significantly in terms of its size, color distribution, or other measurable characteristics, contrasted with other images in the dataset. An anomaly is an image that includes objects or scenes that are not frequently found within the dataset. Detecting outliers and anomalies in image and text data often requires more intricate techniques compared to numerical data. These methods could involve **Natural Language Processing (NLP)** for text data and computer vision algorithms for image data. It is crucial to address outliers and anomalies correctly as they can greatly affect the efficiency of data analysis and machine learning models.

Tutorial 4.5: An example to demonstrate the concept of outliers in text data, is as follows:

```
1. import numpy as np
2. # Create a CountVectorizer instance to convert text data
   into a bag-of-words representation
3. from sklearn.feature_extraction.text import CountVector
   izer
4. # Let's assume these are the text entries in our dataset
5. texts = [
6.     "I love to play football",
7.     "The weather is nice today",
8.     "Python is a powerful programming language",
9.     "Machine learning is a fascinating field",
10.    "I enjoy reading books",
11.    "The Eiffel Tower is in Paris",
12.    "Outliers are unusual data points that differ significan
```

tly from other observations",

```
13. "Anomaly detection is the identification of rare items,  
    events or observations which raise suspicions by differin  
    g significantly from the majority of the data"  
14. ]  
15. # Convert the texts to word count vectors  
16. vectorizer = CountVectorizer()  
17. X = vectorizer.fit_transform(texts)  
18. # Calculate the length of each text entry  
19. lengths = np.array([len(text.split()) for text in texts])  
20.  
21. # Define a function to detect outliers based on text lengt  
    h  
22. def detect_outliers(data):  
23.     outliers = []  
24.     threshold = 1 # this could be any value based on your  
        understanding of the data  
25.     mean = np.mean(data)  
26.     std = np.std(data)  
27.     for i in data:  
28.         z_score = (i - mean) / std  
29.         if np.abs(z_score) > threshold:  
30.             outliers.append(i)  
31.     return outliers  
32.  
33. outliers = detect_outliers(lengths)  
34. print(  
35.     f"Outlier text entries based on length: {[texts[i] for i, x  
        in enumerate(lengths) if x in outliers]}")
```

Here, we first define a list of text entries. We then convert these texts to word count vectors using the **CountVectorizer** class from

sklearn.feature_extraction.text. This allows us to calculate the length of each text entry. We then define a function **detect_outliers** to detect outliers based on text length. This function uses the **z-score** method to detect outliers, similar to the method used for numerical data. The **detect_outliers** function should detect the last text entry as an outlier because it is significantly longer than the other text entries.

Output:

1. Outlier text entries based on length: ['Anomaly detection is the identification of rare items, events or observations which raise suspicions by differing significantly from the majority of the data']

In the output, the function **detect_outliers** is designed to identify texts that are significantly longer or shorter than the average length of texts in the dataset. The output text is considered an outlier because it contains more words than most of the other texts in the dataset.

For anomaly detection in text data, more advanced techniques are typically required, such as topic modeling or sentiment analysis. These techniques are beyond the scope of this simple example. Detecting anomalies in text data could involve identifying texts that are off-topic or have unusual sentiment compared to the rest of the dataset. This would require NLP techniques and is a large and complex field of study in itself.

Tutorial 4.6: An example to demonstrate detection of anomalies in text data, based on the **Z-score** method. Considering the length of words in a text, anomalies in this context would be words that are significantly longer than the average, is as follows:

1. `import numpy as np`
- 2.
3. `# Define a function to detect anomalies`

```

4. def find_anomalies(text):
5.     # Split the text into words
6.     words = text.split()
7.     # Calculate the length of each word
8.     word_lengths = [len(word) for word in words]
9.     # Calculate the mean and standard deviation of the word lengths
10.    mean_length = np.mean(word_lengths)
11.    std_dev_length = np.std(word_lengths)
12.    # Define a list to hold anomalies
13.    anomalies = []
14.    # Find anomalies: words whose length is more than 1
    standard deviations away from the mean
15.    for word in words:
16.        z_score = (len(word) - mean_length) / std_dev_length
17.        if np.abs(z_score) > 1:
18.            anomalies.append(word)
19.    return anomalies
20.
21. text = "Despite having osteosarchaematosplanchnochon
    droneuromuelous and osseocarnisanguineoviscericartilaginonervomedullary conditions, he is fit."
22. print(find_anomalies(text))

```

Output:

```

1. ['osteosarchaematosplanchnochondroneuromuelous', 'osseocarnisanguineoviscericartilaginonervomedullary']

```

Since, the words highlighted in the output have a z-score greater than one, they have been identified as anomalies. However, the definition of an outlier can change based on the context and the specific statistical methods you are using.

Probability

Probability is the likelihood of an event occurring, it is between 0 and 1, where 0 means the event is impossible and 1 means it is certain. For example, when you flip a coin, you can get either heads or tails. The chance of getting heads is $1/2$ or 50%. That is because each outcome has an equal chance of occurring, and one of them is heads. Probability can also be used to determine the likelihood of more complicated events, such as the chance of getting two heads in a row is one in four, or 25%. For example, flipping a coin twice has four possible outcomes: heads-heads, heads-tails, tails-heads, tails-tails.

Probability consists of outcomes, events, sample space. Let us look at them in detail as follows:

- **Outcomes** are results of an experiment, like in coin toss head and tail are outcomes.
- **Events** are set of one or more outcomes and sample space is set of all possible outcomes. In the coin flip experiment, the event getting heads consists of the single outcome heads. In a dice roll, the event rolling a number less than 5 includes the outcomes 1, 2, 3, and 4.
- **Sample space** is set of all possible outcomes. For the coin flip experiment, the sample space is {heads, tails}. For the dice experiment, the sample space is {1, 2, 3, 4, 5, 6}.

Tutorial 4.7: An example to illustrate probability, outcomes, events, and sample space using the example of rolling dice, is as follows:

1. `import random`
2. `# Define the sample space`
3. `sample_space = [1, 2, 3, 4, 5, 6]`
4. `print(f"Sample space: {sample_space}")`
5. `# Define an event`

```

6. event = [2, 4, 6]
7. print(f"Event of rolling an even number: {sample_space}"
  )
8. # Conduct the experiment (roll the die)
9. outcome = random.choice(sample_space)
10. # Check if the outcome is in the event
11. if outcome in event:
12.     print(f"Outcome {outcome} is in the event.")
13. else:
14.     print(f"Outcome {outcome} is not in the event.")
15. # Calculate the probability of the event
16. probability = len(event) / len(sample_space)
17. print(f"Probability of the event: {probability}.")

```

Output:

1. Sample space: [1, 2, 3, 4, 5, 6]
2. Event of rolling an even number: [1, 2, 3, 4, 5, 6]
3. Outcome 1 is not in the event.
4. Probability of the event: 0.5.

Probability distribution

Probability distribution is a mathematical function that provides the probabilities of occurrence of different possible outcomes in an experiment. Let us consider flipping a fair coin. The experiment has two possible outcomes, **Heads (H)** and **Tails (T)**. Since the coin is fair, the likelihood of both outcomes is equal.

This experiment can be represented using a probability distribution, as follows:

- Probability of getting heads $P(H) = 0.5$
- Probability of getting tails $P(T) = 0.5$

In probability theory, the sum of all probabilities within a distribution must always equal 1, representing every

possible outcome of an experiment. For instance, in our coin flip example, $P(H) + P(T) = 0.5 + 0.5 = 1$. This is a fundamental rule in probability theory.

Probability distributions can be discrete and continuous as follows:

- **Discrete probability** distributions are used for scenarios with finite or countable outcomes. For example, you have a bag of 10 marbles, 5 of which are red and 5 of which are blue. If you randomly draw a marble from the bag, the possible outcomes are a red marble or a blue marble. Since there are only two possible outcomes, this is a discrete probability distribution. The probability of getting a red marble is $1/2$, and the probability of getting a blue marble is $1/2$.

Tutorial 4.8: To illustrate discrete probability distributions based on example of 10 marbles, 5 of which are red and 5 of which are blue, is as follows:

```
1. import random
2. # Define the sample space
3. sample_space = ['red', 'red', 'red', 'red', 'red', 'blue', 'blue', 'blue', 'blue', 'blue']
4. # Conduct the experiment (draw a marble from the bag)
5. outcome = random.choice(sample_space)
6. # Check if the outcome is red or blue
7. if outcome == 'red':
8.     print(f"Outcome is a: {outcome}")
9. elif outcome == 'blue':
10.    print(f"Outcome is a: {outcome}")
11. # Calculate the probability of the events
12. probability_red = sample_space.count('red') / len(sample_space)
13. probability_blue = sample_space.count('blue') / len(sample_space)
```

```
14. print(f"Overall probablity of drawing a red marble: {probability_red}")
15. print(f"Overall probablity of drawing a blue marble: {probability_blue}")
```

Output:

1. Outcome is a: red
2. Overall probablity of drawing a red marble: 0.5
3. Overall probablity of drawing a blue marble: 0.5

- **Continuous probability** distributions are used for scenarios with an infinite number of possible outcomes. For example, you have a scale that measures the weight of objects to the nearest gram. When you weigh an apple, the possible outcomes are any weight between 0 and 1000 grams. This is a continuous probability distribution because there are an infinite number of possible outcomes in the range of 0 to 1000 grams. The probability of getting any particular weight, such as 150 grams, is zero. However, we can calculate the probability of getting a weight within a certain range, such as between 100 and 200 grams.

Tutorial 4.9: To illustrate continuous probability distributions, is as follows:

```
1. import numpy as np
2. # Define the range of possible weights
3. min_weight = 0
4. max_weight = 1000
5. # Generate a random weight for the apple
6. apple_weight = np.random.uniform(min_weight, max_weight)
7. print(f"Weight of the apple is {apple_weight} grams")
8. # Define a weight range
9. min_range = 100
```

```

10. max_range = 200
11. # Check if the weight is within the range
12. if min_range <= apple_weight <= max_range:
13.     print(f"Weight of the apple is within the range of {min_range}-{max_range} grams")
14. else:
15.     print(f"Weight of the apple is not within the range of {min_range}-{max_range} grams")
16. # Calculate the probability of the weight being within the range
17. probability_range = (max_range - min_range) / (max_weight - min_weight)
18. print(f"Probability of the weight of the apple being within the range of {min_range}-{max_range} grams is {probability_range}")

```

Output:

1. Weight of the apple is 348.2428034693577 grams
2. Weight of the apple is not within the range of 100-200 grams
3. Probability of the weight of the apple being within the range of 100-200 grams is 0.1

Uniform distribution

In uniform distribution, all possible outcomes are equally likely. The flipping a fair coin, is a uniform distribution. There are two possible outcomes: **Heads (H)** and **Tails (T)**. Here, every outcome is equally likely.

Tutorial 4.10: An example to illustrate uniform probability distributions, is as follows:

1. `import random`
2. *# Define the sample space*
3. `sample_space = ['H', 'T']`
4. *# Conduct the experiment (flip the coin)*

```

5. outcome = random.choice(sample_space)
6. # Print the outcome
7. print(f"Outcome of the coin flip: {outcome}")
8. # Calculate the probability of the events
9. probability_H = sample_space.count('H') / len(sample_space)
10. probability_T = sample_space.count('T') / len(sample_space)
11. print(f"Probability of getting heads (P(H)): {probability_H}")
12. print(f"Probability of getting tails (P(T)): {probability_T}")

```

Output:

1. Outcome of the coin flip: T
2. Probability of getting heads (P(H)): 0.5
3. Probability of getting tails (P(T)): 0.5

Normal distribution

Normal distribution is symmetric about the mean, meaning that data near the mean is more likely to occur than data far from the mean. It is also known as the **Gaussian distribution** and describes data with bell-shaped curves. For example, measuring the test scores of 100 students. The resulting data would likely follow a normal distribution, with most students' scores falling around the mean and fewer students having very high or low scores.

Tutorial 4.11: An example to illustrate normal probability distributions, is as follows:

```

1. import numpy as np
2. import matplotlib.pyplot as plt
3. from scipy.stats import norm
4. # Define the parameters for the normal distribution,
5. # where loc is the mean and scale is the standard deviation

```


on.

```
6. # Let's assume the average test score is 70 and the standard deviation is 10.
7. loc, scale = 70, 10
8. # Generate a sample of test scores
9. test_scores = np.random.normal(loc, scale, 100)
10. # Create a histogram of the test scores
11. plt.hist(test_scores, bins=20, density=True, alpha=0.6, color='g')
12. # Plot the probability distribution function
13. xmin, xmax = plt.xlim()
14. x = np.linspace(xmin, xmax, 100)
15. p = norm.pdf(x, loc, scale)
16. plt.plot(x, p, 'k', linewidth=2)
17. title = "Fit results: mean = %.2f, std = %.2f" % (loc, scale)
18. plt.title(title)
19. plt.savefig('normal_distribution.jpg', dpi=600, bbox_inches='tight')
20. plt.show()
```

Output:

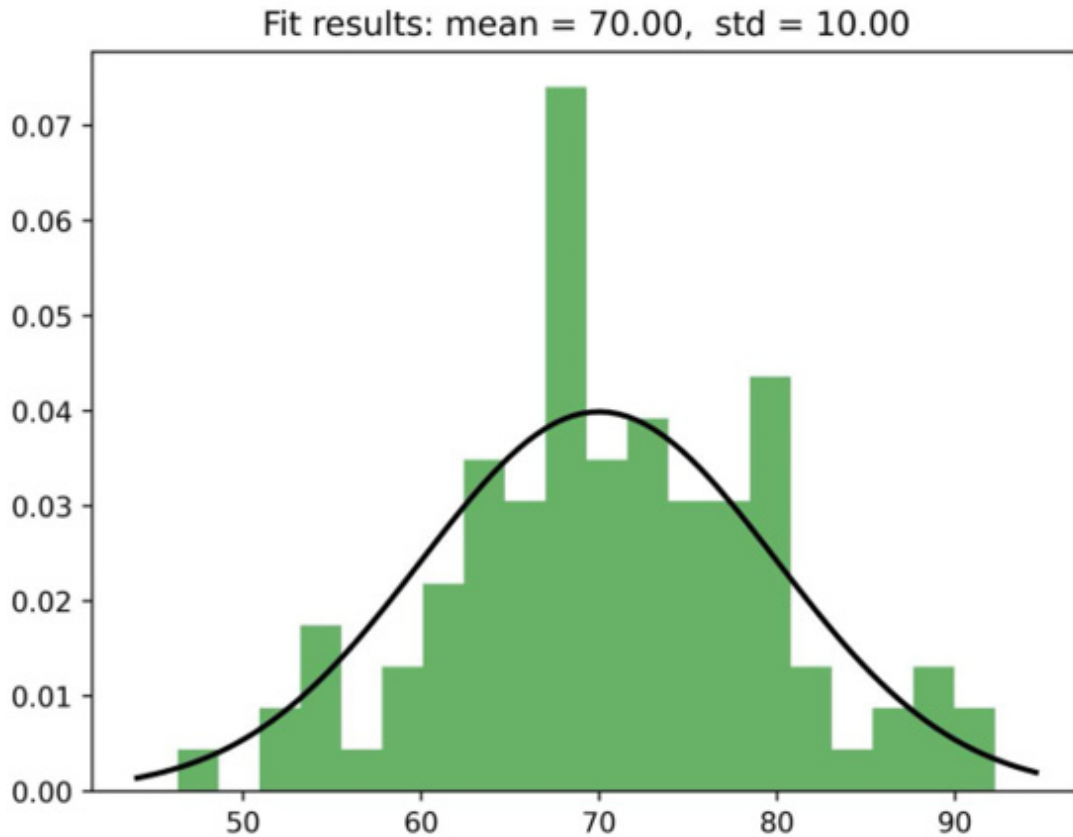


Figure 4.2: Plot showing the normal distribution

Binomial distribution

Binomial distribution describes the number of successes in a series of independent trials that only have two possible outcomes: success or failure. It is determined by two parameters, n , which is the number of trials, and p , which is the likelihood of success in each trial. For example, suppose you flip a coin ten times. There is a 50-50 chance of getting either heads or tails. For instance, the likelihood of getting strictly three heads is, we can use the binomial distribution to figure out how likely it is to get a specific number of heads in those ten flips.

For instance, the likelihood of getting strictly three heads, is as follows:

$$P(X = 3) = nCr * p^x * (1-p)^{(n-x)}$$

Where:

- nCr is the binomial coefficient, which is the number of ways to choose x successes out of n trials
- p is the probability of success on each trial (0.5 in this case)
- $(1-p)$ is the probability of failure on each trial (0.5 in this case)
- x is the number of successes (3 in this case)
- n is the number of trials (10 in this case)

Substituting the values provided, we can calculate that there is a 12.16% chance of getting exactly 3 heads out of ten-coin tosses.

Tutorial 4.12: An example to illustrate binomial probability distributions, using coin toss example, is as follows:

```
1. from scipy.stats import binom
2. import matplotlib.pyplot as plt
3. import numpy as np
4. # number of trials, probability of each trial
5. n, p = 10, 0.5
6. # generate a range of numbers from 0 to n (number of trials)
7. x = np.arange(0, n+1)
8. # calculate binomial distribution
9. binom_dist = binom.pmf(x, n, p)
10. # display probability distribution of each
11. for i in x:
12.     print(
13.         f"Probability of getting exactly {i} heads in {n} flips is: {binom_dist[i]:.5f}")
14. # plot the binomial distribution
15. plt.bar(x, binom_dist)
16. plt.title(
```

```
17. 'Binomial Distribution PMF: 10 coin Flips, Odds of Success for Heads is p=0.5')
18. plt.xlabel('Number of Heads')
19. plt.ylabel('Probability')
20. plt.savefig('binomial_distribution.jpg', dpi=600, bbox_inches='tight')
21. plt.show()
```

Output:

1. Probability of getting exactly 0 heads in 10 flips is: 0.00098
2. Probability of getting exactly 1 heads in 10 flips is: 0.00977
3. Probability of getting exactly 2 heads in 10 flips is: 0.04395
4. Probability of getting exactly 3 heads in 10 flips is: 0.11719
5. Probability of getting exactly 4 heads in 10 flips is: 0.20508
6. Probability of getting exactly 5 heads in 10 flips is: 0.24609
7. Probability of getting exactly 6 heads in 10 flips is: 0.20508
8. Probability of getting exactly 7 heads in 10 flips is: 0.11719
9. Probability of getting exactly 8 heads in 10 flips is: 0.04395
10. Probability of getting exactly 9 heads in 10 flips is: 0.00977
11. Probability of getting exactly 10 heads in 10 flips is: 0.00098

Binomial Distribution PMF: 10 coin Flips, Odds of Success for Heads is $p=0.5$

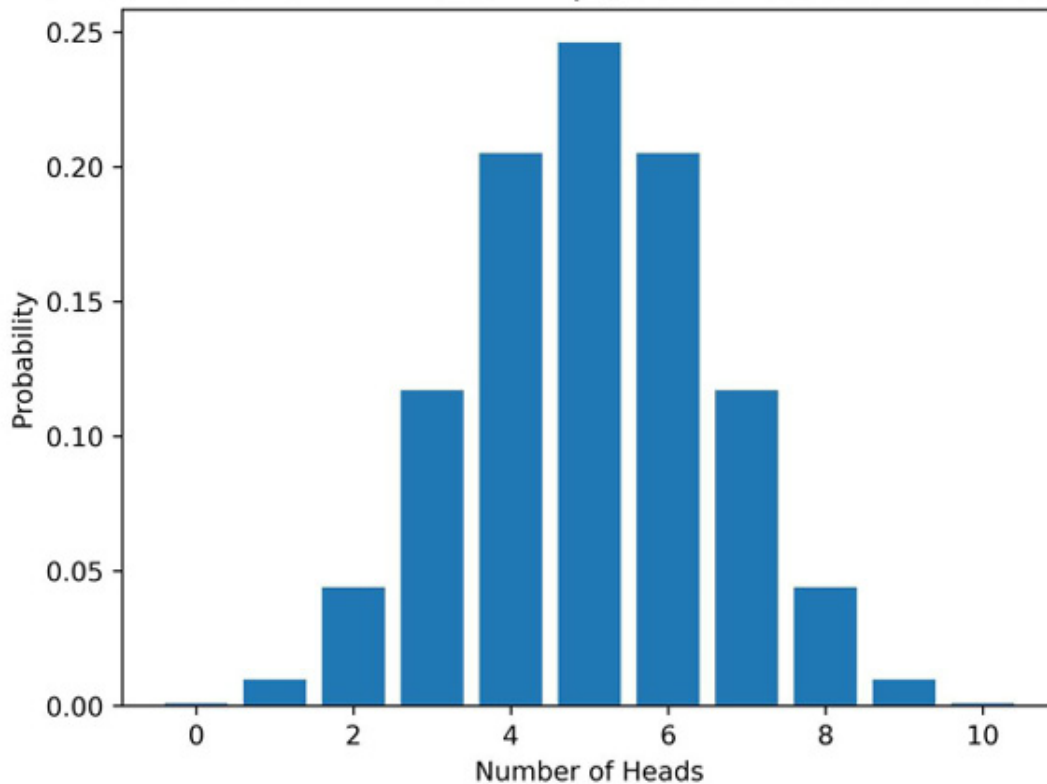


Figure 4.3: Plot showing the normal distribution

Poisson distribution

Poisson distribution is a discrete probability distribution that describes the number of events occurring in a fixed interval of time or space if these events occur independently and with a constant rate. The Poisson distribution has only one parameter, λ (lambda), which is the mean number of events. For example, assume you run a website that gets an average of 500 visitors per day. This is your λ (lambda). Now you want to find the probability of getting exactly 550 visitors in a day. This is a Poisson distribution problem because the number of visitors can be any non-negative integer, the visitors arrive independently, and you know the average number of visitors per day. Using the Poisson distribution formula, you can calculate the probability.

Tutorial 4.13: An example to illustrate Poisson probability

distributions, is as follows:

```
1. from scipy.stats import poisson
2. import matplotlib.pyplot as plt
3. import numpy as np
4. # average number of visitors per day
5. lambda_ = 500
6. # generate a range of numbers from 0 to 600
7. x = np.arange(0, 600)
8. # calculate Poisson distribution
9. poisson_dist = poisson.pmf(x, lambda_)
10. # number of visitors we are interested in
11. k = 550
12. prob_k = poisson.pmf(k, lambda_)
13. print(f"Probability of getting exactly {k} visitors in a day
    is: {prob_k:.5f}")
14. # plot the Poisson distribution
15. plt.bar(x, poisson_dist)
16. plt.title('Poisson Distribution PMF:  $\lambda=500$ ')
17. plt.xlabel('Number of Visitors')
18. plt.ylabel('Probability')
19. plt.savefig('poisson_distribution.jpg', dpi=600, bbox_inches='tight')
20. plt.show()
```

We set **lambda_** to 500 in the program, representing the average daily visitors. The average number of visitors per day is 500. We generate numbers between 0 and 600 for x to cover your desired number of visitors, specifically 550. The program calculates and displays a bar chart of the Poisson distribution once executed. This chart represents the probability of receiving a specific number of visitors per day. The horizontal axis indicates the number of visitors, and the vertical axis displays the probability. The chart

displays the likelihood of having a certain number of visitors in a day. Each bar on the chart represents the probability of obtaining that exact number of visitors in one day.

Output:

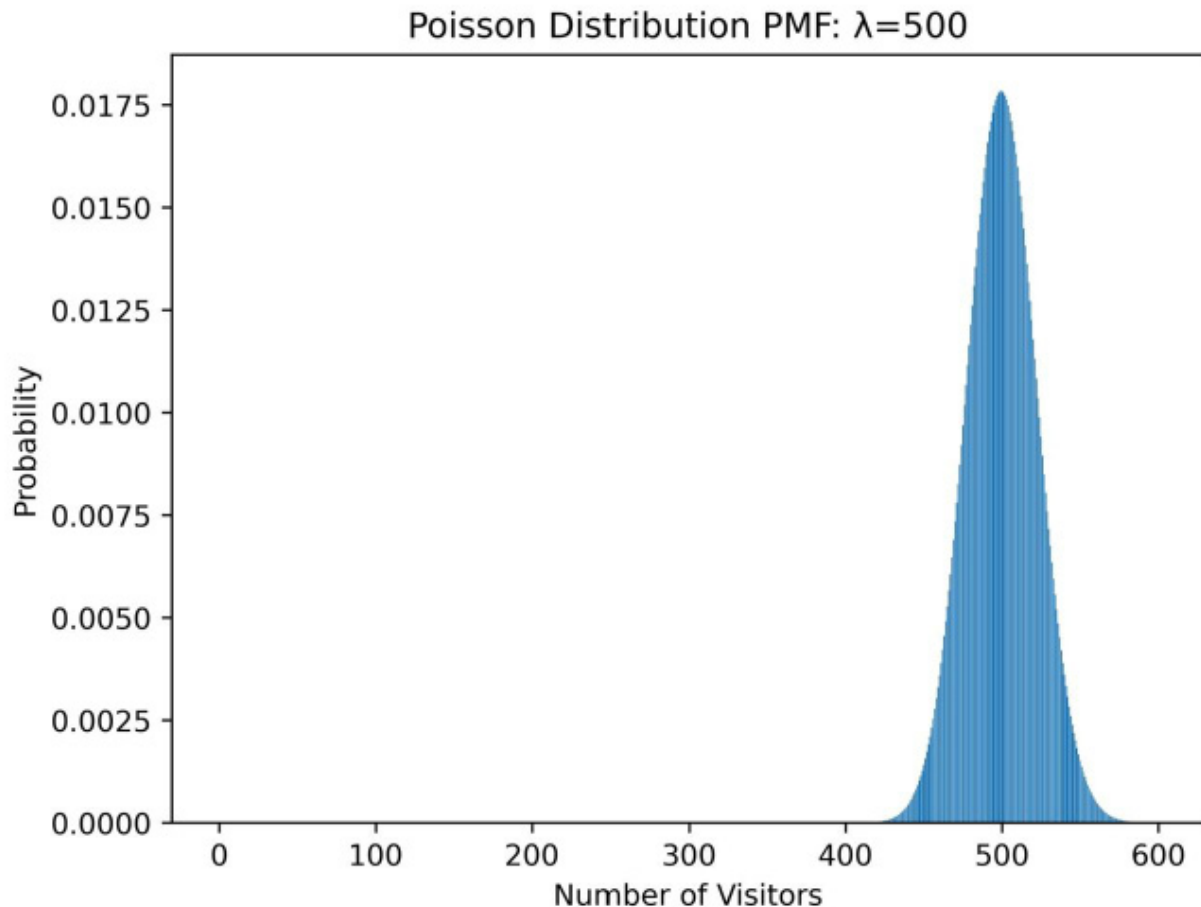


Figure 4.4: Plot showing the Poisson distribution

Array and matrices

Arrays are collections of elements of the same data type, arranged in a linear fashion. They are used to hold a collection of numerical data points, representing a variety of things, such as measurements taken over time, scores on a test, or other information. **Matrices** are 2-Dimensional arrays of numbers or symbols arranged in rows and

columns, used to organize and manipulate data in a structured way.

Arrays and matrices are fundamental structures to store and manipulate numerical data, crucial for statistical analysis and modeling. They provide a powerful and efficient way to store, manipulate, compute and analyze large datasets. Both array and matrices are used for the following:

- Storing and manipulating data
- Convenient and efficient mathematical calculation
- Static modelling to analyze data and make prediction

Tutorial 4.14: An example to illustrate array or 1-Dimensional array, is as follows:

```
1. import statistics as stats
2. # Creating an array of data
3. data = [2, 8, 3, 6, 2, 4, 8, 9, 2, 5]
4. # Calculating the mean
5. mean = stats.mean(data)
6. print("Mean: ", mean)
7. # Calculating the median
8. median = stats.median(data)
9. print("Median: ", median)
10. # Calculating the mode
11. mode = stats.mode(data)
12. print("Mode: ", mode)
```

Output:

```
1. Mean: 4.9
2. Median: 4.5
3. Mode: 2
```

Tutorial 4.15: An example to illustrate 2-Dimensional array (which are matrix), is as follows:

```
1. import numpy as np
```



```

2. # Creating a 2D array (matrix) of data
3. data = np.array([[2, 8, 3], [6, 2, 4], [8, 9, 2], [5, 7, 1]])
4. # Calculating the mean of each row
5. mean = np.mean(data, axis=1)
6. print("Mean of each row: ", mean)
7. # Calculating the median of each row
8. median = np.median(data, axis=1)
9. print("Median of each row: ", median)
10. # Calculating the standard deviation of each row
11. std_dev = np.std(data, axis=1)
12. print("Standard deviation of each row: ", std_dev)

```

Output:

```

1. Mean of each row: [4.33333333 4.        6.33333333 4.33333333]
2. Median of each row: [3. 4. 8. 5.]
3. Standard deviation of each row:
   [2.62466929 1.63299316 3.09120617 2.49443826]

```

Use of array and matrix

Use of array and matrices includes, using them to store large and wide data points and also be useful for analysis of those data. For example, use of matrix to storing data from surveys which consist of number of respondents in each age group or the average income for each education level. It can also be used in modeling of data. Let us look *Tutorial 4.16* and *Tutorial 4.17* to illustrate use of a matrix to store data from surveys which shows the number of respondents in each age group and the average income for each education level.

Tutorial 4.16: An example to illustrate use of a matrix to store data from surveys which shows the number of respondents in each age group, is as follows:

We first create a 2D array (matrix) to store the survey data.

With each row in the matrix stands for a survey taker, and every column corresponds to an attribute (like age range, education level, or earnings).

```
1. import numpy as np
2. # Creating a matrix to store survey data
3. data = np.array([
4.     ['18-24', 'High School', 30000],
5.     ['25-34', 'Bachelor', 50000],
6.     ['35-44', 'Master', 70000],
7.     ['18-24', 'Bachelor', 35000],
8.     ['25-34', 'High School', 45000],
9.     ['35-44', 'Master', 65000]
10. ])
11. print("Data Matrix:")
12. print(data)
```

Output:

```
1. Data Matrix:
2. [['18-24' 'High School' '30000']
3.  ['25-34' 'Bachelor' '50000']
4.  ['35-44' 'Master' '70000']
5.  ['18-24' 'Bachelor' '35000']
6.  ['25-34' 'High School' '45000']
7.  ['35-44' 'Master' '65000']]
```

Tutorial 4.17: To extend above *Tutorial 4.16* for basic analysis of data matrix to compute the average income for each education level, is as follows:

```
1. import numpy as np
2. # Creating a matrix to store survey data
3. data = np.array([
4.     ['18-24', 'High School', 30000],
5.     ['25-34', 'Bachelor', 50000],
```

```

6.     ['35-44', 'Master', 70000],
7.     ['18-24', 'Bachelor', 35000],
8.     ['25-34', 'High School', 45000],
9.     ['35-44', 'Master', 65000]
10. ])
11. # Calculating the number of respondents in each age group
12. age_groups = np.unique(data[:, 0], return_counts=True)
13. print("Number of respondents in each age group:")
14. for age_group, count in zip(age_groups[0], age_groups[1]):
15.     print(f"{age_group}: {count}")
16. # Calculating the average income for each education level
17. education_levels = np.unique(data[:, 1])
18. print("\nAverage income for each education level:")
19. for education_level in education_levels:
20.     income = data[data[:, 1] == education_level]
21.            [:, 2].astype(np.float64)
22.     average_income = np.mean(income)
23.     print(f"{education_level}: {average_income}")

```

In this program, we first create a matrix to store the survey data. We then calculate the number of respondents in each age group by finding the unique age groups in the first column of the matrix and counting the occurrences of each. Next, we calculate the average income for each education level by iterating over the unique education levels in the second column of the matrix, filtering the matrix for each education level, and calculating the average of the income values in the third column.

Output:

1. Number of respondents in each age group:
2. 18-24: 2

3. 25-34: 2
4. 35-44: 2
- 5.
6. Average income for each education level:
7. Bachelor: 42500.0
8. High School: 37500.0
9. Master: 67500.0

Conclusion

Understanding covariance and correlation is critical to determining relationships between variables, while understanding outliers and anomalies is essential to ensuring the accuracy of data analysis. The concept of probability and its distributions is the backbone of statistical prediction and inference. Finally, understanding arrays and matrices is fundamental to performing complex computations and manipulations in data analysis. These concepts are not only essential in statistics, but also have broad applications in fields as diverse as data science, machine learning, and artificial intelligence. Using covariance, correlation, observing outliers, anomalies, understanding of how data and probability concepts are used to predict outcomes and analyze the likelihood of events. All of these descriptive statistics concepts help to untangles statistical relationships. Finally, this covers descriptive statistics,

In *Chapter 5, Estimation and Confidence Intervals* we will start with the important concept of inferential statistics and how estimation is done, confidence interval is measured.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Estimation and Confidence Intervals

Introduction

Estimation involves making an inference on the true value, while the confidence interval provides a range of values that we can be confident contains the true value. For example, suppose you are a teacher and you want to estimate the average height of the students in your school. It is not possible to measure the height of every student, so you take a sample of 30 students and measure their heights. Let us say the average height of your sample is 160 cm and the standard deviation is 10 cm. This average of 160 cm is your point estimate of the average height of all students in your school. However, it should be noted that the 30 students sampled may not be a perfect representation of the entire class, as there may be taller or shorter students who were not included. Therefore, it cannot be definitively concluded that the average height of all students in the class is exactly 160 cm. To address this uncertainty, a confidence interval can be calculated. A confidence interval is an estimate of the range in which the true population mean, the average height of all students in

the class, is likely to lie. It is based on the sample mean and standard deviation and provides a measure of the uncertainty in the estimate. In this example, a 95% confidence interval was calculated, indicating that there is a 95% probability that the true average height of all students in the class falls between 155 cm and 165 cm.

These concepts from descriptive statistics aid in making informed decisions based on the available data by quantifying uncertainty, understanding variations around an estimate, comparing different estimates, and testing hypotheses.

Structure

In this chapter, we will discuss the following topics:

- Points and interval estimation
- Standard error and margin of error
- Confidence intervals

Objectives

By the end of this chapter, readers will be introduced to the concept of estimation in data analysis and explain how to perform it using different methods. Estimation is the process of inferring unknown population parameters from sample data. There are two types of estimation: point estimation and interval estimation. This chapter will also discuss the types of errors in estimation, and how to measure them. Moreover, this chapter will demonstrate how to construct and interpret various confidence intervals for different scenarios, such as comparing means, proportions, or correlations. Finally, this chapter will show how to use t-tests and p-values to test hypotheses about population parameters based on confidence intervals. Examples and exercises will be provided throughout the

chapter to help the reader understand and apply the concepts and methods of estimation.

Point and interval estimate

Point estimate is a single value that represents our best approximate value for an unknown population parameter. It is like taking a snapshot of a population based on a limited sample. This snapshot is not the perfect representation of the entire population, but it serves as a best guess or estimate. Some common point estimates used in statistics are mean, median, mode, variance standard deviation, proportion of the sample. For example, a manufacturing company might want to estimate the average life span of a product. They sample a few products from a production batch and measure their durability. The average lifespan of these samples is a point estimate of the expected lifespan of the product in general.

Tutorial 5.1: An illustration of point estimate based on life span of ten products, is as follows:

```
1. import numpy as np
2. # Simulate product lifespans for a sample of 10 products
3. product_lifespans = [539.84,458.10,474.71,560.67,
465.95,474.46,545.27,419.74,447.93,471.52]
4. # Print the lifespan of the product
5. print("Lifespan of the product:", product_lifespans)
6. # Calculate the average lifespan of the sample
7. average_lifespan = np.mean(product_lifespans)
8. # Print the point estimate for the average lifespan of the product
9. print(f"Point estimate for the average lifespan of the product:{average_lifespan:.2f}")
```

Output:

1. Lifespan of the product: [539.84, 458.1, 474.71, 560.67, 465.95, 474.46, 545.27, 419.74, 447.93, 471.52]
2. Point estimate for the average lifespan of the product: 485.82

Another example is, you are a salesperson for a grocery store chain and you want to estimate the average household spending on groceries in Oslo. It is impossible to collect data from every household, so you randomly select 500 households and record their food expenditures. The average expenditure of this sample represents the point estimate of the total expenditure of all households in Oslo.

Tutorial 5.2: An illustration of the point estimate based on household spending on groceries, is as follows:

1. `import numpy as np`
2. `# Set the seed for reproducibility`
3. `np.random.seed(0)`
4. `# Assume the average household spending on groceries is between $100 and $500`
5. `expenditures = np.random.uniform(low=100, high=500, size=500)`
6. `# Calculate the point estimate (average expenditure of the sample)`
7. `point_estimate = np.mean(expenditures)`
8. `print(f"Point estimate of the total expenditure of all households in the Oslo: NOK {point_estimate:.2f}")`

Output:

1. Point estimate of the total expenditure of all households in the Oslo: NOK 298.64

Tutorial 5.3: An illustration of the point estimate based on mean, median, mode, variance standard deviation, proportion of the sample, is as follows:

1. `import numpy as np`

```
2. # Sample data for household spending on groceries
3. household_spending = np.array([250.32, 195.87, 228.24,
    212.81,
    233.99, 241.45, 253.34, 208.53, 231.23, 221.28])
4. # Calculate point estimate for household spending using mean
5. mean_household_spending = np.mean(household_spending)
6. print(f"Point estimate of household spending using mean:{mean_household_spending}")
7. # Calculate point estimate for household spending using median
8. median_household_spending = np.median(household_spending)
9. print(f"Point estimate of household spending using median:{median_household_spending}")
10. # Calculate point estimate for household spending using mode
11. mode_household_spending = np.argmax(np.histogram(household_spending)[0])
12. print(f"Point estimate of household spending using mode:{household_spending[mode_household_spending]}")
13. # Calculate point estimate for household spending using variance
14. variance_household_spending = np.var(household_spending)
15. print(f"Point estimate of household spending using variance:{variance_household_spending:.2f}")
16. # Calculate point estimate for household spending using standard deviation
17. std_dev_household_spending = np.std(household_spending)
18. print(f"Point estimate of household spending using standard deviation:{std_dev_household_spending:.2f}")
```

```
19. # Calculate point estimate for proportion of households
    spending over $213
20. proportion_household_spending_over_213 = len(househ
    old_spending[household_spending > 213]) / len(househ
    old_spending)
21. print("Proportion of households spending over NOK 213
    :", proportion_household_spending_over_213)
```

Output:

1. Point estimate of household spending using mean:227.706
2. Point estimate of household spending using median:229.735
3. Point estimate of household spending using mode:228.24
4. Point estimate of household spending using variance:305.40
5. Point estimate of household spending using standard deviation:17.48
6. Proportion of households spending over NOK 213: 0.7

An **interval estimate** is a range of values that is likely to contain the true value of a population parameter. It is calculated from sample data and provides more information about the uncertainty of the estimate than a point estimate. For example, suppose you want to estimate the average height of all adult males in Norway. You take a random sample of 100 adult males and find that their average height is 5'10". This is a point estimate of the average height of all adult males in Norway.

However, you know that the average height of a small sample of men is likely to be different from the average height of the entire population. This is due to sampling error. Sampling error is the difference between the sample mean and the population mean. To account for sampling error, you can calculate an interval estimate. An interval

estimate is a range of values that is likely to contain the true average height of all adult males in Norway.

The formula for calculating an interval estimate is: *point estimate* $\pm$ *margin of error*

The margin of error is the amount of sampling error you are willing to accept. A common margin of error is ± 1.96 standard deviations from the sample mean. Using this formula, you can calculate that the 95% confidence interval for the average height of all adult males in the Norway, assuming margin of error of 0.68 inches as: 5'10" $\pm$ 0.68 inches. This means that you are 95% confident that the true average height of all adult males in the Norway is between 5'9" and 5'11".

Tutorial 5.4: To estimate an interval of average lifespan of the product, is as follows:

```
1. import numpy as np
2. # Simulate product lifespans for a sample of 20 products
3. product_lifespans = np.random.normal(500, 50, 20)
4. # Print the lifespan of the product
5. print("Lifespan of the product:", product_lifespans)
6. # Calculate the sample mean and standard deviation
7. sample_mean = np.mean(product_lifespans)
8. sample_std = np.std(product_lifespans)
9. # Calculate the 95% confidence interval
10. confidence_level = 0.95
11. margin_of_error = 1.96 * sample_std / np.sqrt(20)
12. lower_bound = sample_mean - margin_of_error
13. upper_bound = sample_mean + margin_of_error
14. # Print the 95% confidence interval
15. print(
16.     "95% confidence interval for the average lifespan of the product:", (lower_bound, upper_bound)
```

17.)

Tutorial 5.4 simulates 20 product lifetimes from a normal distribution with a mean of 500 and a standard deviation of 50. It calculates the sample mean and standard deviation of the simulated data, and then determines 95% confidence interval using the sample mean, standard deviation, and confidence level. The confidence interval is a range of values that is likely to contain the true mean lifetime of the product.

Output:

1. Lifespan of the product: [546.83712318 570.6163853 381.52065474 543.20261502 388.01979707
2. 520.07495275 561.24352821 503.24280532 436.01554134 470.72843979
3. 486.91772771 490.88776081 489.85515796 494.50586103 510.67400245
4. 439.57131731 487.89900851 575.91305852 480.76772884 477.80819534]
5. 95% confidence interval for the average lifespan of the product: (469.90930134271343, 515.7208647778637)

Tutorial 5.5: To estimate an interval of average household spending on groceries based on ten sample data, is as follows:

1. `import numpy as np`
2. `# Sample data for household spending on groceries`
3. `household_spending = np.array([250.32, 195.87, 228.24, 212.81, 233.99, 241.45, 253.34, 208.53, 231.23, 221.28])`
4. `# Calculate the sample mean and standard deviation`
5. `sample_mean = np.mean(household_spending)`
6. `sample_std = np.std(household_spending)`
7. `# Calculate the 95% confidence interval`

```

8. confidence_level = 0.95
9. margin_of_error = 1.96 * sample_std / np.sqrt
   (len(household_spending))
10. lower_bound = sample_mean - margin_of_error
11. upper_bound = sample_mean + margin_of_error
12. # Print the 95% confidence interval
13. print(
14.     "95% confidence interval for the average
       household spending:", (lower_bound, upper_bound)
15. )

```

Tutorial 5.5 initially calculates the sample mean and standard deviation of the household expenditure data. It then uses these values, along with the confidence level, to calculate the 95% confidence interval. This interval represents a range of values that is likely to contain the true average household spending in the population.

Output:

```

1. 95% confidence interval for the average household spending:
   (216.87441676204998, 238.53758323795)

```

Standard error and margin of error

Standard error measures the precision of an estimate of a population mean. The smaller the standard error, the more accurate the estimate. The standard error is calculated as the square root of the variance of the sample. It measures the variability in a sample. It is calculated as follows:

$$\text{Standard Error} = \text{Standard Deviation} / \sqrt{(\text{Sample Size})}$$

For example, a researcher wants to estimate the average weight of all adults in Oslo. She randomly selects 100 adults and finds that their average weight is 160 pounds. The sample standard deviation is 15 pounds. Then, the standard error is as follows:

Standard error = 15 pounds / $\sqrt{(100)}$ pounds = 1.5 pounds

Tutorial 5.6: An implementation of standard error, is as follows:

```
1. import math
2. # Sample size
3. n = 100
4. # Sample mean
5. mean = 160
6. # Sample standard deviation
7. sd = 15
8. # Standard error
9. se = sd / math.sqrt(n)
10. # Print standard error
11. print("Standard error:", se)
```

Output:

```
1. Standard error: 1.5
```

Margin of error on the other side measures the uncertainty in a sample statistic, such as the mean or proportion. It is an estimate of the range within which the true population mean is likely to fall with a specified level of confidence. It is calculated by multiplying the standard error by a z-score, which is a value from a standard normal distribution. The z-score is chosen based on the desired confidence level, t-score is used instead of the z-score when the sample size is small (less than 30), is as follows:

$$\text{Margin of Error} = \text{z-score} * \text{Standard Error}$$

For example, a researcher wants to estimate the average weight of all adults in Oslo with 95% confidence. The z-score for the 95% confidence level is 1.96. Then, the margin of error is as follows:

$$\text{Margin of error} = 1.96 * 1.5 \text{ pounds} = 2.94 \text{ pounds}$$

This means that the researcher is 95% confident that the

average weight of all adults in Oslo is between 157.06 pounds and 162.94 pounds.

Tutorial 5.7: An implementation of margin of error, is as follows:

```
1. import math
2. # Sample size
3. n = 100
4. # Sample mean
5. mean = 160
6. # Sample standard deviation
7. sd = 15
8. # Z-score for 95% confidence
9. z_score = 1.96
10. # Margin of error
11. moe = z_score * sd / math.sqrt(n)
12. # Print margin of error
13. print("Margin of error:", moe)
14. # Calculate confidence interval
15. confidence_interval = (mean - moe, mean + moe)
16. # Print confidence interval
17. print("Confidence interval:", confidence_interval)
```

Output:

```
1. Margin of error: 2.94
2. Confidence interval: (157.06, 162.94)
```

Tutorial 5.8: Calculating the standard error and margin of error for a survey. For example, a political pollster conducted a survey to estimate the proportion of registered voters in a particular district who support a specific candidate. The survey included 100 randomly selected registered voters in the district, and the results showed that 60% of them support the candidate as follows:


```

1. import numpy as np
2. # Example data representing survey responses (1 for support, 0 for not support)
3. data = [1, 0, 1, 1, 0, 1, 0, 0, 1, 1]
4. sample_mean = 0.6 # Calculate sample mean
5. # Calculate sample standard deviation
6. sample_std = np.std([1 if voter == 1 else 0 for voter in data])
7. # Calculate standard error
8. standard_error = sample_std / np.sqrt(len(data))
9. print(f"Standard Error:{standard_error:.2f}")
10. # Find z-score for 95% confidence level
11. z_score = 1.96
12. # Calculate margin of error
13. margin_of_error = z_score * standard_error
14. print(f"Margin of Error:{margin_of_error:.2f}")

```

Output:

1. Standard Error:0.09
2. Margin of Error:0.19

A standard error of 0.09 is an indication that your sample mean is relatively close to the population mean. 0.6 is the sample proportion because 60% support the candidate. This means that the pollster can be 95% confident that the true proportion of registered voters in the district who support the candidate is between 41% and 79%.

Confidence intervals

All confidence intervals are interval estimates, but not all interval estimates are confidence intervals. Interval estimate is a broader term that refers to any range of values that is likely to contain the true value of a population parameter. For instance, if you have a population of

students and want to estimate their average height, you might reason that it is likely to fall between 5 feet 2 inches and 6 feet 2 inches. This is an interval estimate, but it does not have a specific probability associated with it.

Confidence interval, on the other hand, is a specific type of interval estimate that is accompanied by a probability statement. For example, a 95% confidence interval means that if you repeatedly draw different samples from the same population, 95% of the time, the true population parameter will fall within the calculated interval.

As discussed, confidence interval is also used to make inferences about the population based on the sample data.

Tutorial 5.9: Suppose you want to estimate the average height of all adult women in your city. You take a sample of 100 women and find that their average height is 5 feet 5 inches. You want to estimate the true average height of all adult women in the city with 95% confidence. This means that you are 95% confident that the true average height is between 5 feet 3 inches and 5 feet 7 inches. Based on this example a Python program illustrating confidence intervals, is as follows:

```
1. import numpy as np
2. from scipy import stats
3. # Sample data
4. data = np.array([5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9,
                    6])
5. # Calculate sample mean and standard deviation
6. mean = np.mean(data)
7. std = np.std(data)
8. # Calculate confidence interval with 95% confidence level
9. margin_of_error = stats.norm.ppf(0.975) * std / np.sqrt(
    len(data))
```

```
10. confidence_interval = (mean - margin_of_error, mean +  
    margin_of_error)  
11. print("Sample mean:", mean)  
12. print("Standard deviation:", std)  
13. print("95% confidence interval:", confidence_interval)
```

Output:

```
1. Sample mean: 5.55  
2. Standard deviation: 0.2872281323269015  
3. 95% confidence interval: (5.371977430445669, 5.72802  
    2569554331)
```

The sample mean is 5.55, indicating that the average height in the sample is 5.55 feet. The standard deviation is 0.287, indicating that the heights in the sample vary by about 0.287 feet. The 95% confidence interval is (5.371, 5.72), which suggests that we can be 95% confident that the true average height of all adult women in the city falls within this range. To put it simply, if we were to take multiple samples of 10 women from the city and calculate the average height of each sample, the true average height would fall within the range of 5.37 feet to 5.72 feet 95% of the time.

Tutorial 5.10: A Python program to illustrate confidence interval for the age column in the diabetes dataset, is as follows:

```
1. import pandas as pd  
2. # Load the diabetes data from a csv file  
3. diabeties_df = pd.read_csv("/workspaces/Implementing  
    StatisticsWithPython/data/chapter1/diabetes.csv")  
4. # Calculate the mean and standard deviation of the 'Age'  
    ' column  
5. mean = diabeties_df['Age'].mean()  
6. std_dev = diabeties_df['Age'].std()  
7. # Calculate the standard error
```

```
8. std_err = std_dev / (len(diabities_df['Age']) ** 0.5)
9. # Calculate the 95% Confidence Interval
10. ci = stats.norm.interval(0.95, loc=mean, scale=std_err)
11. print(f"95% confidence interval for the 'Age' column is {
    ci}")
```

Output:

```
1. 95% confidence interval for the 'Age' column is
    (32.40915352661263, 34.0726173067207)
```

Types and interpretation

The importance of confidence intervals lies in their ability to measure the uncertainty or variability around a sample estimate. Confidence intervals are especially useful when studying an entire population is not feasible, so researchers select a sample or subgroup of the population.

Following are some common types of confidence intervals:

- A confidence interval for a mean estimates the population mean. It is used especially when the data follows a normal distribution. It is discussed in *Point Interval Estimate*, and *Confidence Interval* above.
- When data does not follow a normal distribution, various methods may be used to calculate the confidence interval. For example, suppose you are researching the duration of website loading times. You have collected data from 20 users and discovered that the load times are not normally distributed, possibly due to a few users having slow internet connections that skew the data. In this scenario, one way to calculate the confidence interval is to use the bootstrap method. To estimate the confidence interval, the data is resampled with replacement multiple times, and the mean is calculated each time. The distribution of these means is then used.

Tutorial 5.11: A Python program that uses the bootstrap

method to calculate the confidence interval for non-normally distributed data, is as follows:

```
1. import numpy as np
2. def bootstrap(data, num_samples, confidence_level):
3.     # Create an array to hold the bootstrap samples
4.     bootstrap_samples = np.zeros(num_samples)
5.     # Generate the samples
6.     for i in range(num_samples):
7.         sample = np.random.choice(data, len(data), replace=True)
8.         bootstrap_samples[i] = np.mean(sample)
9.     # Calculate the confidence interval
10.    lower_percentile = (1 - confidence_level) / 2 * 100
11.    upper_percentile = (1 + confidence_level) / 2 * 100
12.    confidence_interval = np.percentile(
13.        bootstrap_samples, [lower_percentile, upper_percentile])
14.    return confidence_interval
15. # Suppose these are your load times
16. load_times = [1.2, 0.9, 1.3, 2.1, 1.8, 2.4, 1.9, 2.2, 1.7,
17.               2.3, 1.5, 2.0, 1.6, 2.5, 1.4, 2.6, 1.1, 2.7, 1.0, 2.8]
18. # Calculate the confidence interval
19. confidence_interval = bootstrap(load_times, 1000, 0.95)
20. print(f"95% confidence interval : {confidence_interval}")
```

Output:

```
1. 95% confidence interval : [1.614875 2.085]
```

- A confidence interval for proportions estimates the population proportion. It is used when dealing with categorical data. More about this is illustrated in *Confidence Interval For Proportion*.
- Another type of confidence interval estimates the difference between two population means or proportions.

It is used when you want to compare the means or proportions of two populations.

Confidence interval and t-test relation

The t-test is used to compare the means of two independent samples or the mean of a sample to a population mean. It is a type of hypothesis test used to determine whether there is a statistically significant difference between the two means. The t-test assumes that the two samples are drawn from normally distributed populations with equal variances. As the confidence interval is calculated using the sample mean, the standard error of the mean, and the desired confidence level.

Tutorial 5.12: To illustrate the use of the t-test for confidence intervals, consider the following example: Suppose we want to estimate the average height of adult male basketball players in the United States. We randomly sample 50 male basketball players and measure their heights. We then calculate the sample mean and standard deviation of the heights as follows:

```
1. import numpy as np
2. # Sample heights of 50 male basketball players
3. heights = np.array([75, 78, 76, 79, 80, 81, 82, 83, 84,
4.                    85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
5.                    96, 97, 98, 99, 100, 101, 102, 103,
6.                    104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114,
7.                    115, 116, 117, 118, 119, 120])
8. # Calculate sample mean and standard deviation
9. mean = heights.mean()
10. std = heights.std()
```

Then, we will now calculate a 95% confidence interval for the mean height of male basketball players in the population. To determine the appropriate standard error of the mean, we will use the t-test since we are working with a

sample of the population as follows:

```
1. from scipy import stats
2. # Calculate degrees of freedom
3. df = len(heights) - 1
4. # Calculate t-statistic for 95% confidence level
5. t = stats.t.ppf(0.95, df)
6. # Calculate standard error of the mean
7. sem = std / np.sqrt(len(heights))
8. # Calculate confidence interval
9. ci = (mean - t * sem, mean + t * sem)
10. print("95% confidence interval for population mean:", ci
    )
```

The *Tutorial 5.12* output shows, we can be 95% confident that the true population mean height of male basketball players in the United States is between 77.6 and 84 inches. Here the t-test is used to calculate the **Standard Error of the Mean (SEM)**, which is a crucial component of the confidence interval. The SEM represents the average difference between the sample mean and the true population mean. To account for the variability within the sample, the t-test considers the sample size and the standard deviation of the heights. This ensures that the confidence interval is not overly narrow or excessively wide, providing a reliable range for the true population mean.

Confidence interval and p-value

A confidence interval is a range of values that likely contains the true value of a parameter with a certain level of confidence. A p-value is a probability that measures the compatibility of the observed data with a null hypothesis. The relationship between confidence intervals and p-values is based on the same underlying theory and calculations, but they convey different information. A p-value indicates

whether the observed data are statistically significant or not, that is, whether they provide enough evidence to reject the null hypothesis or not. A confidence interval provides information on the precision and uncertainty of an estimate, indicating how close it is to the true value and the degree to which it may vary due to random error. One way to understand the relationship is to think of the confidence interval as arms that embrace values consistent with the data. If the null value, usually zero or one, falls within the confidence interval, it is not rejected by the data, and the p-value must be greater than the significance level, usually 0.05. If the null value falls outside the confidence interval, it is rejected by the data, and the p-value must be less than the significance level.

For example, let us say you wish to test the hypothesis that the average height of Norwegian men is 180 cm. To do so, you randomly select a sample of 100 men and measure their heights. You calculate the sample mean and standard deviation, and then you construct a 95% confidence interval for the population mean as follows:

$$\bar{x} \pm 1.96 \frac{s}{\sqrt{n}}$$

Where $\bar{x}$ sample mean, s is sample standard deviation, and n is sample size.

Assuming a confidence interval of (179.31, 181.29), we can conclude with 95% confidence that the true mean height of men in Norway falls between 179.31 and 181.29 cm. As the null value of 180 is within this interval, we cannot reject the null hypothesis at the 0.05 significance level. The p-value for this test must be 0.5562, indicating that the observed data are not very unlikely under the null hypothesis. On the other hand, if you obtain a confidence interval that does not include 180, such as (176.5, 179.5), it

would mean that you can be 95% confident that the actual mean height of men in Norway falls outside the hypothesized value of 180 cm. As the null value of 180 would lie outside this interval, you would reject the null hypothesis at the 0.05 significance level. The p-value for this test would be less than 0.05, indicating that it is highly improbable for the observed data to be true under the null hypothesis.

Tutorial 5.13: To illustrate the use of the p-value and confidence intervals, is as follows:

```
1. # import numpy and scipy libraries
2. import numpy as np
3. import scipy.stats as st
4. # set the random seed for reproducibility
5. np.random.seed(0)
6. # generate a random sample of 100 heights from a normal distribution
7. # with mean 180 and standard deviation 5
8. heights = np.random.normal(180, 5, 100)
9. # calculate the sample mean and standard deviation
10. mean = np.mean(heights)
11. std = np.std(heights, ddof=1)
12. # calculate the 95% confidence interval for the population mean
13. # using the formula: mean +/- 1.96 * std / sqrt(n)
14. n = len(heights)
15. lower, upper = st.norm.interval(0.95, loc=mean, scale=std/np.sqrt(n))
16. # print the confidence interval
17. print(f"95% confidence interval for the population mean : ({lower:.2f}, {upper:.2f})")
18. # test the null hypothesis that the population mean is 180
```

```

19. # using a one-sample t-test
20. t_stat, p_value = st.ttest_1samp(heights, 180)
21. # print the p-value
22. print(f"P-value for the one-sample t-test : {p_value:.4f}")
23. # compare the p-
    value with the significance level of 0.05
24. # and draw the conclusion
25. if p_value < 0.05:
26.     print("We reject the null hypothesis that the populati
    on mean is 180")
27. else:
28.     print("We fail to reject the null hypothesis that the po
    pulation mean is 180")

```

Output:

1. 95% confidence interval for the population mean : (179.31, 181.29)
2. P-value for the one-sample t-test : 0.5562
3. We fail to reject the null hypothesis that the population mean is 180

This indicates that the confidence interval includes the null value of 180, and the p-value is greater than 0.05. Therefore, we cannot reject the null hypothesis due to insufficient evidence.

Confidence interval for mean

Some of the concepts have already been described and highlighted in the *Types and Interpretation* section above. Let us see the what, how and when of confidence interval for the mean. Confidence interval for the mean is a range of values that, with a certain level of confidence, is likely to contain the true mean of a population. It is best to use this type of confidence interval when we have a sample of numerical data from a population and we want to estimate the average of the population.

For example, let us say you want to estimate the average height of students in a class. You randomly select 10 students and measure their heights in centimeters. You get the following data:

```
1. heights = [160, 165, 170, 175, 180, 185, 190, 195, 200, 205]
```

To calculate the 95% confidence interval for the population mean, use the **t.interval** function from the **scipy.stats** library. The confidence parameter should be set to 0.95, and the degrees of freedom should be set to the sample size minus one. Additionally, provide the sample mean and the standard error of the mean as arguments.

Tutorial 5.14: An example to compute confidence interval for mean of the average height of students, is as follows:

```
1. import numpy as np
2. import scipy.stats as st
3. mean = np.mean(heights) # sample mean
4. se = st.sem(heights) # standard error of the mean
5. df = len(heights) - 1 # degrees of freedom
6. ci = st.t.interval(confidence=0.95, df=df, loc=mean, scale=se) # confidence interval
7. print(f"Confidence interval: {ci}")
```

Output:

```
1. Confidence interval: (171.67074705193303, 193.32925294806697)
```

This indicates a 95% confidence interval for the true mean height of students in the class, which falls between 171.67 and 193.32 cm.

Confidence interval for proportion

A confidence interval for a proportion is a range of values that, with a certain level of confidence, likely contains the true proportion of a population. The use this type of

confidence interval is when we have a sample of categorical data from a population and we want to estimate the percentage of the population that belongs to a certain category. For example, let us say you want to estimate the proportion of students in a class who prefer chocolate ice cream over vanilla ice cream. You randomly select 50 students and ask them about their preference. You get the following data:

1. `preferences = ['chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate',`
2. `'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla',`
3. `'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate',`
4. `'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate',`
5. `'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla', 'chocolate', 'chocolate', 'vanilla']`

To compute the 95% confidence interval for the population proportion, you can use the **binom.interval** function from the **scipy.stats** library. You need to pass the confidence parameter as 0.95, the number of trials as the sample size, and the probability of success as the sample proportion.

Tutorial 5.15: An example of computing a confidence interval for the proportion of students in a class who prefer chocolate ice cream to vanilla ice cream, using the above list of preferences, is as follows:

1. `import scipy.stats as st`
2. `n = len(preferences) # sample size`

```
3. p = preferences.count('chocolate') / n # sample proportion
4. ci = st.binom.interval(confidence=0.95, n=n, p=p) # confidence interval
5. print(f"Confidence interval: {ci}")
```

Output:

```
1. Confidence interval: (26.0, 39.0)
```

This indicates a 95% confidence level that the actual proportion of students in the class who prefer chocolate ice cream over vanilla ice cream falls between 26/50 and 39/50.

Tutorial 5.16: A Python program that calculates the confidence interval for a proportion. In this case, we are estimating the proportion of people who prefer coffee over tea.

For example, if a survey of 100 individuals is conducted and 60 of them express a preference for coffee over tea, the proportion is 0.6. The confidence interval provides a range within which the actual proportion of coffee enthusiasts in the population are likely to fall.

```
1. import scipy.stats as stats
2. n = 100 # Number of trials
3. x = 60 # Number of successes
4. # Calculate the proportion
5. p = x / n
6. # Confidence level
7. confidence_level = 0.95
8. # Calculate the confidence interval
9. ci_low, ci_high = stats.binom.interval(confidence_level,
    n, p)
10. print(f"The {confidence_level*100}% confidence interval
    for the proportion is ({ci_low/n}, {ci_high/n})")
```

This program uses the **binom.interval** function from the

scipy.stats module to calculate the confidence interval. The **binom.interval** function returns the endpoints of the confidence interval for the Binomial distribution. The confidence interval is then scaled by *n* to give the confidence interval for the proportion.

Output:

1. The 95.0% confidence interval for the proportion is (0.5, 0.69)

Confidence interval for differences

A confidence interval for the difference is a range of values that likely contains the true difference between two population parameters with a certain level of confidence. This confidence interval type is suitable when there are two independent data samples from two populations, and the parameters of the two populations need to be compared. For example, suppose you want to compare the average heights of male and female students in a class. You randomly select 10 male and 10 female students and measure their heights in centimeters. The following data is obtained:

1. male_heights = [170, 175, 180, 185, 190, 195, 200, 205, 210, 215]
2. female_heights = [160, 165, 170, 175, 180, 185, 190, 195, 200, 205]

To calculate the 95% confidence interval for the difference between population means, use the **ttest_ind** function from the **scipy.stats** library. Pass the two samples as arguments and set the **equal_var** parameter to **False** if you assume that the population variances are not equal. The function returns the t-statistic and the p-value of the test. Use the p-value to calculate the confidence interval.

Tutorial 5.17: An example of calculating the confidence interval for differences between two population means, is

as follows:

```
1. import numpy as np
2. import scipy.stats as st
3. t_stat, p_value = st.ttest_ind(male_heights, female_heights, equal_var=False) # t-test
4. mean1 = np.mean(male_heights) # sample mean of male heights
5. mean2 = np.mean(female_heights) # sample mean of female heights
6. se1 = st.sem(male_heights) # standard error of male heights
7. se2 = st.sem(female_heights) # standard error of female heights
8. sed = np.sqrt(se1**2 + se2**2) # standard error of difference
9. confidence = 0.95
10. z = st.norm.ppf((1 + confidence) / 2) # z-score for the confidence level
11. margin_error = z * sed # margin of error
12. ci = ((mean1 - mean2) - margin_error, (mean1 - mean2) + margin_error) # confidence interval
13. print(f"Confidence interval: {ci}")
```

Output:

```
1. Confidence interval: (-3.2690189017555973, 23.269018901755597)
```

This means that there is a 95% confidence that the actual difference between the average heights of male and female students in the class falls between -3.27 and 23.27 cm.

Tutorial 5.18: A Python program that calculates the confidence interval for the difference between two population means, Nepalese and Norwegians.

For example, if you measure the average number of hours of television watched per week by 100 Norwegian and 100

Nepalese, the difference between the means plus or minus the variation provides the confidence interval as follows:

```
1. import numpy as np
2. import scipy.stats as stats
3. # Suppose these are your data
4. norwegian_hours = np.random.normal(loc=10, scale=2,
    size=100) # Normally distributed data with mean=10,
    std dev=2
5. nepalese_hours = np.random.normal(loc=8, scale=2.5,
    size=100) # Normally distributed data with mean=8, st
    d dev=2.5
6. # Calculate the means
7. mean_norwegian = np.mean(norwegian_hours)
8. mean_nepalese = np.mean(nepalese_hours)
9. # Calculate the standard deviations
10. std_norwegian = np.std(norwegian_hours, ddof=1)
11. std_nepalese = np.std(nepalese_hours, ddof=1)
12. # Calculate the standard error of the difference
13. sed = np.sqrt(std_norwegian**2 / len(norwegian_hours)
    + std_nepalese**2 / len(nepalese_hours))
14. # Confidence level
15. confidence_level = 0.95
16. # Calculate the confidence interval
17. ci_low, ci_high = stats.norm.interval(confidence_level, l
    oc=(mean_norwegian - mean_nepalese), scale=sed)
18. print(f"The {confidence_level*100}% confidence interval
    for the difference in means : ({ci_low:.2f}, {ci_high:.2f})
    ")
```

This program uses the **norm.interval** function from the **scipy.stats** module to compute the confidence interval. The **norm.interval** function returns the endpoints of the confidence interval for the normal distribution. The

confidence interval is then used to estimate the range within which the difference in population means is likely to fall.

Output:

1. The 95.0% confidence interval for the difference in means : (1.44, 2.65)

The output is (1.44, 2.65), we can be 95% confident that the true difference in the average number of hours of television watched per week between Norwegians and Nepalese is between 1.44 and 2.65 hours.

Confidence interval estimation for diabetes data

Here we apply the above point and confidence interval estimation in the diabetes dataset. The diabetes dataset contains information on 768 patients, such as their number of pregnancies, glucose level, blood pressure, skin thickness, insulin level, BMI, diabetes pedigree function, age, and outcome (whether they have diabetes or not). The outcome variable is a binary variable, where 0 means no diabetes and 1 means diabetes. The other variables are either numeric or categorical. One way to use point and interval estimation in the diabetes dataset is to estimate the mean and proportion of each variable for the entire population of patients and construct confidence intervals for these estimates. Another way to use point and interval estimates in the diabetes dataset is to compare the mean and proportion of each variable between the two groups of patients, those with diabetes and those without diabetes, and construct confidence intervals for the differences. The implementation is shown in the following tutorials.

Tutorial 5.19: An example to estimate the mean of glucose level for the whole population of patients.

For estimating the mean of glucose level for the whole

population of patients, we can use the sample mean as a point estimate, and construct a 95% confidence interval as an interval estimate as follows:

```
1. import pandas as pd
2. import numpy as np
3. import scipy.stats as st
4. import matplotlib.pyplot as plt
5. # Load the diabetes data from a csv file
6. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
7. # get the glucose column
8. x = data["Glucose"]
9. # get the sample size
10. n = len(x)
11. # get the sample mean
12. mean = x.mean()
13. # get the sample standard deviation
14. std = x.std()
15. # set the confidence level
16. confidence = 0.95
17. # get the critical value
18. z = st.norm.ppf((1 + confidence) / 2)
19. # get the margin of error
20. margin_error = z * std / np.sqrt(n)
21. # get the lower bound of the confidence interval
22. lower = mean - margin_error
23. # get the upper bound of the confidence interval
24. upper = mean + margin_error
25. print(f"Point estimate of the population mean of glucose level is {mean:.2f}")
26. print(f"95% confidence interval of the population mean of glucose level is ({lower:.2f}, {upper:.2f})")
```

Output:

1. Point estimate of the population mean of glucose level is 120.89
2. 95% confidence interval of the population mean of glucose level is (118.63, 123.16)

This means the point estimation is 120.89. And that we are 95% confident that the true mean of glucose level for the whole population of patients is between 118.63 and 123.16. Now, further let us compute the standard error and margin of error of the estimation and see what it shows.

Tutorial 5.20: An implementation to compute the standard error and the margin of error when estimating the mean glucose level for the whole population of patients, is as follows:

```
1. import pandas as pd
2. import numpy as np
3. import scipy.stats as st
4. import matplotlib.pyplot as plt
5. # Load the diabetes data from a csv file
6. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
7. # get the glucose column
8. x = data["Glucose"]
9. # get the sample size
10. n = len(x)
11. # get the sample mean
12. mean = x.mean()
13. # get the sample standard deviation
14. std = x.std()
15. # set the confidence level
16. confidence = 0.95
17. # get the critical value
```

```

18. z = st.norm.ppf((1 + confidence) / 2)
19. # define a function to calculate the standard error
20. def standard_error(std, n):
21.     return std / np.sqrt(n)
22.
23. # define a function to calculate the margin of error
24. def margin_error(z, se):
25.     return z * se
26.
27. # call the functions and print the results
28. se = standard_error(std, n)
29. me = margin_error(z, se)
30. print(f"Standard error of the sample mean is {se:.2f}")
31. print(f"Margin of error for the 95% confidence interval is {me:.2f}")

```

Output:

1. Standard error of the sample mean is 1.15
2. Margin of error for the 95% confidence interval is 2.26

The average glucose level is estimated with a precision of 1.15 units and a 95% confidence interval of plus or minus 2.26 (i.e, 120.89 ± 2.26).

Tutorial 5.21: An implementation for estimating the proportion of patients with diabetes for the whole population of patients.

To estimate the proportion of patients with diabetes for the whole population of patients, we can use the sample proportion as a point estimate, and construct a 95% confidence interval as an interval estimate.

1. import pandas as pd
2. import numpy as np
3. import scipy.stats as st
4. import matplotlib.pyplot as plt

```

5. # Load the diabetes data from a csv file
6. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
7. # get the outcome column
8. y = data["Outcome"]
9. # get the sample size
10. n = len(y)
11. # get the sample proportion
12. p = y.mean()
13. # set the confidence level
14. confidence = 0.95
15. # get the critical value
16. z = st.norm.ppf((1 + confidence) / 2)
17. # get the margin of error
18. margin_error = z * np.sqrt(p * (1 - p) / n)
19. # get the lower bound of the confidence interval
20. lower = p - margin_error
21. # get the upper bound of the confidence interval
22. upper = p + margin_error
23. print(f"Point estimate of the population proportion of patients with diabetes is {p:.2f}")
24. print(f"95% confidence interval of the population proportion of patients with diabetes is ({lower:.2f}, {upper:.2f})")

```

Output:

1. Point estimate of the population proportion of patients with diabetes is 0.35
2. 95% confidence interval of the population proportion of patients with diabetes is (0.32, 0.38)

This means the 35% of patient are diabetic and that we are 95% confident that the true proportion of patients with diabetes for the whole population of patients is between

0.32 and 0.38.

Confidence interval estimate in text

We apply point and confidence interval estimation to analyze the word length in transaction narrative notes. This statistical method helps us examine text file data, specifically analyzing the format of the transaction narratives provided.

The narrative contain text in following format.

1. Date: 2023-08-01
2. Merchant: VideoStream Plus
3. Amount: \$9.99
4. Description: Monthly renewal of VideoStream Plus subscription.
- 5.
6. Your subscription to VideoStream Plus has been successfully renewed for \$9.99.

Tutorial 5.22: An implementation of point and confidence interval in the transaction narrative text to compute the average word and the 95% confidence interval in the text file, is as follows:

1. `import scipy.stats as st`
2. `# Read the text file as a string`
3. `with open("/workspaces/ImplementingStatisticsWithPython/data/chapter1/TransactionNarrative/1.txt", "r") as f:`
4. `text = f.read()`
5. `# Split the text by whitespace characters and remove empty strings`
6. `words = [word for word in text.split() if word]`
7. `# Calculate the length of each word`
8. `lengths = [len(word) for word in words]`

```

9. # Calculate the point estimate of the mean length
10. mean = sum(lengths) / len(lengths)
11. # Calculate the standard error of the mean length
12. sem = st.sem(lengths)
13. # Calculate the 95% confidence interval of the mean length
14. ci = st.t.interval(confidence=0.95, df=len(lengths)-1, loc=mean, scale=sem)
15. # Print the results
16. print(f"Point estimate of the mean length is {mean:.2f} characters")
17. print(
18.     f"95% confidence interval of the mean length is {ci[0]:.2f} to {ci[1]:.2f} characters")

```

Output:

1. Point estimate of the mean length is 6.27 characters
2. 95% confidence interval of the mean length is 5.17 to 7.37 characters

Here, the mean length point estimate is the average length of all the words in the text file. It is a single value that summarizes the data. You calculated it by dividing the sum of the lengths by the number of words. The point estimate of the average length is 6.27 characters. This means that the average word in the text file is about 6 characters long. Similarly, the 95% confidence interval of the mean length is an interval that, with 95% probability, contains the true mean length of the words in the text file. It is a range of values that reflects the uncertainty of the point estimate. You calculated it using the **t.interval** function, which takes as arguments the confidence level, the degrees of freedom, the point estimate, and the standard error of the mean. The standard error of the mean is a measure of how much the point estimate varies from sample to sample. The 95%

confidence interval for the mean is 5.17 to 7.37 characters. This means that you are 95% confident that the true average length of the words in the text file is between 5.17 and 7.37 characters.

Tutorial 5.23: An implementation to visualize computed point and confidence interval in a plot, is as follows:

```
1. import matplotlib.pyplot as plt
2. # Create a figure and an axis
3. fig, ax = plt.subplots()
4. # Plot the point estimate as a horizontal line
5. ax.hlines(mean, xmin=0, xmax=len(lengths), color='blue', label='Point estimate')
6. # Plot the confidence interval as a shaded area
7. ax.fill_between(x=range(len(lengths)), y1=ci[0], y2=ci[1], color='orange', alpha=0.3, label='95% confidence interval')
8. # Add some labels and a legend
9. ax.set_xlabel('Word index')
10. ax.set_ylabel('Word length')
11. ax.set_title('Confidence interval of the mean word length')
12. ax.legend()
13. # Show the plot
14. plt.show()
```

Output:

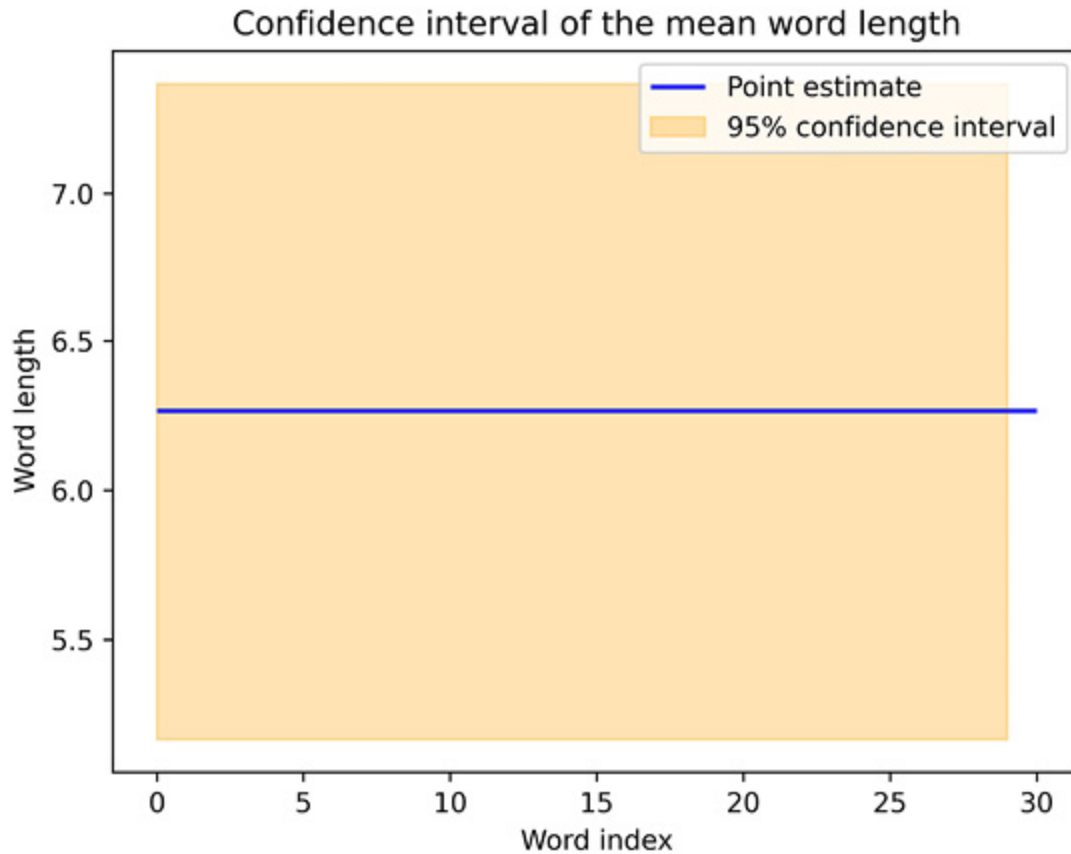


Figure 5.1: Plot showing point estimate and confidence interval of mean word length

The plot shows the confidence interval of the mean word length for some data. The plot has a horizontal line in blue representing the point estimate of the mean, and a shaded area in orange representing the 95% confidence interval around the mean.

Conclusion

In this chapter, we have learned how to estimate unknown population parameters from sample data using various methods. We saw that there are two types of estimation: point estimation and interval estimation. Point estimation gives a single value as the best guess for the parameter, while interval estimation gives a range of values that includes the parameter with a certain degree of confidence.

We have also discussed the errors in estimation and how to measure them using standard error and margin of error. In addition, we have shown how to construct and interpret different confidence intervals for different scenarios, such as comparing means, proportions, or correlations. We learned how to use t-tests and p-values to test hypotheses about population parameters based on confidence intervals. We applied the concepts and methods of estimation to real-world examples using the diabetes dataset and the transaction narrative.

Similarly, estimation is a fundamental and useful tool in data analysis because it allows us to make inferences and predictions about a population based on a sample. By using estimation, we can quantify the uncertainty and variability of our estimates and provide a measure of their reliability and accuracy. Estimation also allows us to test hypotheses and draw conclusions about the population parameters of interest. It is used in a wide variety of fields and disciplines, including economics, medicine, engineering, psychology, and the social sciences.

We hope this chapter has helped you understand and apply the concepts and methods of estimation in data analysis. The next chapter will introduce the concept of hypothesis and significance testing.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Hypothesis and Significance Testing

Introduction

Testing a claim and drawing conclusion from the result is testing association. It is one of the most done work in statistics. For which hypothesis testing defines a claim and using significance level and bunch of different tests. The validity of the claim in relation to the data is checked. Hypothesis testing is a method of making decisions based on data analysis. It involves stating a null hypothesis and an alternative hypothesis, which are mutually exclusive statements about a population parameter. Significance tests are procedures that assess how likely it is that the observed data are consistent with the null hypothesis. There are different types of statistical tests that can be used for hypothesis testing, depending on the nature of the data and the research question. Such as z-test, t-test, chi-square test, ANOVA. These are described later in the chapter, with examples. Sampling techniques and sampling distributions are important concepts, and sometimes they are critical in hypothesis testing because they affect the validity and reliability of the results. Sampling techniques are methods

of selecting a subset of individuals or units from a population that is intended to be representative of the population. Sampling distributions are the probability distributions of the possible values of a sample statistic based on repeated sampling from the population.

Structure

In this chapter, we will discuss the following topics:

- Hypothesis testing
- Significance tests
- Role of p-value and significance level
- Statistical test
- Sampling techniques and sampling distributions

Objectives

The objective of this chapter is to introduce the concept of hypothesis testing, determining significance, and interpreting hypotheses through multiple testing. A hypothesis is a claim or technique for drawing a conclusion, and a significance test checks the likelihood that the claim or conclusion is correct. We will see how to perform them and interpret the result obtained from the data. This chapter also discusses the types of tests used for hypothesis testing and significance testing. In addition, this chapter will explain the role of the p-value and the significance level. Finally, this chapter shows how to use various hypothesis and significance tests and p-values to test hypotheses.

Hypothesis testing

Hypothesis testing is a statistical method that uses data from a sample to draw conclusions about a population. It involves testing an assumption, known as the null hypothesis, to determine whether it is likely to be true or

false. The null hypothesis typically states that there is no effect or difference between two groups, while the alternative hypothesis is the opposite and what we aim to prove. Hypothesis testing checks if an idea about the world is true or not. For example, you might have an idea that men are taller than women on average, and you want to see if the data support your idea or not.

Tutorial 6.1: An illustration of the hypothesis testing using the example 'men are taller than women on average', as mentioned in above example, is as follows:

```
1. import scipy.stats as stats
2. # define the significance level
3. # alpha = 0.05, which means there is a 5% chance of making a type I error (rejecting the null hypothesis when it is true)
4. alpha = 0.05
5. # generate some random data for men and women heights (in cm)
6. # you can replace this with your own data
7. men_heights = stats.norm.rvs(loc=175, scale=10, size=100) # mean = 175, std = 10
8. women_heights = stats.norm.rvs(loc=165, scale=8, size=100) # mean = 165, std = 8
9. # calculate the sample means and standard deviations
10. men_mean = men_heights.mean()
11. men_std = men_heights.std()
12. women_mean = women_heights.mean()
13. women_std = women_heights.std()
14. # print the sample statistics
15. print("Men: mean = {:.2f}, std = {:.2f}".format(men_mean, men_std))
16. print("Women: mean = {:.2f}, std = {:.2f}".format(women_mean, women_std))
```

```

17. # perform a two-sample t-test
18. # the null hypothesis is that the population means are equal
19. # the alternative hypothesis is that the population means are not equal
20. t_stat, p_value = stats.ttest_ind(men_heights, women_heights)
21. # print the test statistic and the p-value
22. print("t-statistic = {:.2f}".format(t_stat))
23. print("p-value = {:.4f}".format(p_value))
24. # compare the p-value with the significance level and make a decision
25. if p_value <= alpha:
26.     print("Reject the null hypothesis: the population means are not equal.")
27. else:
28.     print("Fail to reject the null hypothesis: the population means are equal.")

```

Output: Number and result may vary based on a random generated number. Following is the snippet of output:

1. Men: mean = 174.48, std = 9.66
2. Women: mean = 165.16, std = 7.18
3. t-statistic = 7.70
4. p-value = 0.0000
5. Reject the null hypothesis: the population means are not equal.

Here is a simple explanation of how hypothesis testing works. Suppose you have a jar of candies, and you want to determine whether there are more red candies than blue candies in the jar. Since counting all the candies in the jar is not feasible, you can extract a handful of them and determine the number of red and blue candies. This process is known as **sampling**. Based on the sample, you can make

an inference about the entire jar. This inference is referred to as a **hypothesis**, which is akin to a tentative answer to a question. However, to determine the validity of this hypothesis, a comparison between the sample and the expected outcome is necessary. For instance, consider the hypothesis: *There are more red candies than blue candies in the jar*. This comparison is known as a **hypothesis test**, which determines the likelihood of the sample matching the hypothesis. For instance, if the hypothesis is correct, the sample should contain more red candies than blue candies. However, if the hypothesis is incorrect, the sample should contain roughly the same number of red and blue candies. A test provides a numerical measurement of how well the sample aligns with the hypothesis. This measurement is known as a **p-value**, which indicates the level of surprise in the sample. A low p-value indicates a highly significant result, while a high p-value indicates a result that is not statistically significant. For instance, if you randomly select a handful of candies and they are all red, the result would be highly significant, and the p-value would be low. However, if you randomly select a handful of candies and they are half red and half blue, the result would not be statistically significant, and the p-value would be high. Based on the p-value, one can determine whether the hypothesis is true or false. This determination is akin to a final answer to the question. For instance, if the p-value is low, it can be concluded that the hypothesis is true, and one can state that there are more red candies than blue candies in the jar. Conversely, if the p-value is high, it can be concluded that the hypothesis is false, and one can state: The jar does not contain more red candies than blue candies.

Tutorial 6.2: An illustration of the hypothesis testing using the example *jar of candies*, as mentioned in above example, is as follows:


```

1. # import the scipy.stats library
2. import scipy.stats as stats
3. # define the significance level
4. alpha = 0.05
5. # generate some random data for the number of red and
   blue candies in a handful
6. # you can replace this with your own data
7. n = 20 # number of trials (candies)
8. p = 0.5 # probability of success (red candy)
9. red_candies = stats.binom.rvs(n, p) # number of red can
   dies
10. blue_candies = n - red_candies # number of blue candies
11. # print the sample data
12. print("Red candies: {}".format(red_candies))
13. print("Blue candies: {}".format(blue_candies))
14. # perform a binomial test
15. # the null hypothesis is that the probability of success is
   0.5
16. # the alternative hypothesis is that the probability of suc
   cess is not 0.5
17. p_value = stats.binomtest(red_candies, n, p, alternative=
   'two-sided')
18. # print the p-value
19. print("p-value = {:.4f}".format(p_value.pvalue))
20. # compare the p-
   value with the significance level and make a decision
21. if p_value.pvalue <= alpha:
22.     print("Reject the null hypothesis: the probability of su
   ccess is not 0.5.")
23. else:
24.     print("Fail to reject the null hypothesis: the probabilit
   y of success is 0.5.")

```

Output: Number and result may vary based on generated

random number. Following is the snippet of output:

1. Red candies: 6
2. Blue candies: 14
3. p-value = 0.1153
4. Fail to reject the null hypothesis: the probability of success is 0.5.

Steps of hypothesis testing

Following are the steps to perform hypothesis testing:

1. State your null and alternate hypothesis. Keep in mind that the null hypothesis is what you assume to be true before you collect any data, while the alternate hypothesis is what you want to prove or test. For instance, if you aim to test whether men are taller than women on average, your null hypothesis could be: **There is no significant difference in height between men and women.** The alternate hypothesis could be: **On average, men are taller than women.**

In *Tutorial 6.1*, the following snippet states hypothesis:

1. *# the null hypothesis is that the population means are equal*
2. *# the alternative hypothesis is that the population means are not equal*
3. `t_stat, p_value = stats.ttest_ind(men_heights, women_heights)`

In *Tutorial 6.2*, the following snippet states hypothesis:

1. *# the null hypothesis is that the probability of success is 0.5*
2. *# the alternative hypothesis is that the probability of success is not 0.5*
3. `p_value = stats.binomtest(red_candies, n, p, alternative='two-sided')`
2. Collect data in a way that is designed to test your

hypothesis. For example, you might measure the heights of a random sample of men and women from different regions and social classes.

In *Tutorial 6.1*, the following snippet generates 100 random samples of heights from a normal distribution with a specified mean (**loc**) and a standard deviation (**scale**):

1. `men_heights = stats.norm.rvs(loc=175, scale=10, size=100) # mean = 175, std = 10`
2. `women_heights = stats.norm.rvs(loc=165, scale=8, size=100) # mean = 165, std = 8`

In *Tutorial 6.2*, the following snippet generates random number of candies based on scenario where there are 20 candies, each with a 50% chance of being red:

1. `n = 20 # number of trials (candies)`
2. `p = 0.5 # probability of success (red candy)`
3. `red_candies = stats.binom.rvs(n, p) # number of red candies`
4. `blue_candies = n - red_candies # number of blue candies`

3. Perform a statistical test that compares your data with your null hypothesis. It's crucial to choose the appropriate statistical test based on the nature of your data and the objective of your study, which are described in the *Statistical test* section below. For example, you might use a t-test to see if the average height of men is different from the average height of women in your sample.

In *Tutorial 6.1*, the following snippet performs a test to compute the t-statistic and p-value:

1. `t_stat, p_value = stats.ttest_ind(men_heights, women_heights)`

In *Tutorial 6.2*, the following snippet performs a binomial test to compute the p-value:

1. `p_value = stats.binom_test(red_candies, n, p, alternative='two-sided')`

4. Decide whether to reject or fail to reject your null hypothesis based on your test result. For instance, you can use a significance level of 0.05. This means you are willing to accept a 5% chance of being wrong. If your p-value is less than 0.05, you can reject your null hypothesis and accept your alternate hypothesis. If your p-value is more than 0.05, you cannot reject your null hypothesis and must keep it.

In *Tutorial 6.1*, the following snippet checks the hypothesis based on the p-value:

1. `if p_value <= alpha:`
2. `print("Reject the null hypothesis: the population means are not equal.")`
3. `else:`
4. `print("Fail to reject the null hypothesis: the population means are equal.")`

In *Tutorial 6.2*, the following snippet checks the hypothesis based on the p-value.

1. `if p_value.pvalue <= alpha:`
2. `print("Reject the null hypothesis: the probability of success is not 0.5.")`
3. `else:`
4. `print("Fail to reject the null hypothesis: the probability of success is 0.5.")`

5. Present your findings. For instance, you can report the mean and standard deviation of the heights of men and women in your sample, the t-value and p-value of your test, and your conclusion regarding the hypothesis. In *Tutorial 6.1* and *Tutorial 6.2* all the **print** statement present the findings.

Types of hypothesis testing

There are various types of hypothesis testing, depending on the number and nature of the hypotheses and the data.

Some common types include:

- **One-sided and two-sided tests:** A one-tailed test is when you have a specific direction for your alternative hypothesis, such as *men are on average taller than women*. A two-tailed test is when you have a general direction for your alternative hypothesis, such as *men and women have different average heights*.

For example, suppose you want to know if your class (Class 1) is smarter than another class (Class 2). You could give both classes a math test and compare their scores. A one-tailed test is when you are only interested in one direction, such as my class (Class 1) is smarter than the other class (Class 2). A two-tailed test is when you are interested in both directions, such as Class 1 and the Class 2 are different in smartness.

Tutorial 6.3: An illustration of the one-sided testing using the example *my class (Class 1) is smarter than the other class (Class 2)*, as mentioned in above example, is as follows:

```
1. # Import the scipy.stats module
2. import scipy.stats as stats
3. # Define the scores of both classes as lists
4. class1 = [80, 85, 90, 95, 100, 105, 110, 115, 120, 125]
5. class2 = [75, 80, 85, 90, 95, 100, 105, 110, 115, 120]
6. # Perform a one-sided test to see if class1 is smarter
   than class2
7. # The null hypothesis is that the mean of class1 is less t
   han or
   equal to the mean of class2
8. # The alternative hypothesis is that the mean of class1
   is greater than the mean of class2
9. t_stat, p_value = stats.ttest_ind(class1, class2, alternativ
   e='greater')
10. print('One-sided test results:')
```

```

11. print('t-statistic:', t_stat)
12. print('p-value:', p_value)
13. # Compare the p-value with the significance level
14. if p_value < 0.05:
15.     print('We reject the null hypothesis and conclude that
        class1 is smarter than class2.')
16. else:
17.     print('We fail to reject the null hypothesis and cannot
        conclude that class1 is smarter than class2.')

```

Output:

1. One-sided test results:
2. t-statistic: 0.7385489458759964
3. p-value: 0.23485103640040045
4. We fail to reject the null hypothesis and cannot conclude that class1 is smarter than class2.

Tutorial 6.4: An illustration of the two-sided testing using the example *my class (Class 1) and the other class (Class 2) are different in smartness*, as mentioned in above example, is as follows:

```

1. # Import the scipy.stats module
2. import scipy.stats as stats
3. # Define the scores of both classes as lists
4. class1 = [80, 85, 90, 95, 100, 105, 110, 115, 120, 125]
5. class2 = [75, 80, 85, 90, 95, 100, 105, 110, 115, 120]
6. # Perform a two-
    sided test to see if class1 and class2 are different in smar
    tness
7. # The null hypothesis is that the mean of class1 is equal
    to the mean of class2
8. # The alternative hypothesis is that the mean of class1 is
    not equal to the mean of class2
9. t_stat, p_value = stats.ttest_ind(class1, class2, alternativ
    e='two-sided')

```

```

10. print('Two-sided test results:')
11. print('t-statistic:', t_stat)
12. print('p-value:', p_value)
13. # Compare the p-value with the significance level
14. if p_value < 0.05:
15.     print('We reject the null hypothesis and conclude that
        class1 and class2 are different in smartness.')
16. else:
17.     print('We fail to reject the null hypothesis and cannot
        conclude that class1 and class2 are different in smartness.')

```

Output:

1. Two-sided test results:
2. t-statistic: 0.7385489458759964
3. p-value: 0.4697020728008009
4. We fail to reject the null hypothesis and cannot conclude that class1 and class2 are different in smartness.

- **One-sample and two-sample tests:** A one-sample test is when you compare a single sample to a known population value, such as *the average height of men in Norway is 180 cm*. A two-sample test is when you compare two samples, such as *the average height of men in Norway is different from the average height of men in Japan*.

For example, imagine you want to know if a class is taller than the average height for kids of their age. You can measure the heights of everyone in the class and compare them to the average height of kids of their age. A one-sample test is when you have only one group of data, such as *my class is taller than the average height for kids my age*. A two-sample test is when you have two groups of data, such as *my class is taller than the other*

class.

Tutorial 6.5: An illustration of the one-sample testing using the example *my class (Class 1) is taller than the average height for kids my age*, as mentioned in above example, is as follows:

```
1. # Import the scipy.stats module
2. import scipy.stats as stats
3. # Define the heights of your class as a list
4. my_class = [150, 155, 160, 165, 170, 175, 180, 185, 190,
               195]
5. # Perform a one-
   sample test to see if your class is taller than the average
   height for kids your age
6. # The null hypothesis is that the mean of your class is eq
   ual to the population mean
7. # The alternative hypothesis is that the mean of your cla
   ss is not equal to the population mean (two-sided)
8. # or that the mean of your class is greater than the popul
   ation mean (one-sided)
9. # According to the WHO, the average height for kids ag
   ed 12 years is 152.4 cm for boys
   and 151.3 cm for girls [^1^][1]
10. # We will use the average of these two values as the pop
   ulation mean
11. pop_mean = (152.4 + 151.3) / 2
12. t_stat, p_value = stats.ttest_1samp(my_class, pop_mean,
    alternative='two-sided')
13. print('One-sample test results:')
14. print('t-statistic:', t_stat)
15. print('p-value:', p_value)
16. # Compare the p-value with the significance level
17. if p_value < 0.05:
18.     print('We reject the null hypothesis and conclude that
```


your class is different in height from the average height for kids your age.')

19. `else:`

20. `print('We fail to reject the null hypothesis and cannot conclude that your class is different in height from the average height for kids your age.')`

Output:

1. One-sample test results:

2. t-statistic: 4.313644314582188

3. p-value: 0.0019512458685808432

4. We reject the null hypothesis and conclude that your class is different in height from the average height for kids your age.

Tutorial 6.6: An illustration of the two-sample testing using the example *my class (Class 1) is taller than the other class (Class 2)*, as mentioned in above example, is as follows:

1. *# Import the scipy.stats module*

2. `import` scipy.stats `as` stats

3. *# Define the heights of your class as a list*

4. `my_class = [150, 155, 160, 165, 170, 175, 180, 185, 190, 195]`

5. *# Perform a two-sample test to see if your class is taller than the other class*

6. *# The null hypothesis is that the means of both classes are equal*

7. *# The alternative hypothesis is that the means of both classes are not equal (two-sided)*

8. *# or that the mean of your class is greater than the mean of the other class (one-sided)*

9. *# Define the heights of the other class as a list*

10. `other_class = [145, 150, 155, 160, 165, 170, 175, 180, 185, 190]`

```

11. t_stat, p_value = stats.ttest_ind(my_class, other_class, al
    ternative='two-sided')
12. print('Two-sample test results:')
13. print('t-statistic:', t_stat)
14. print('p-value:', p_value)
15. # Compare the p-value with the significance level
16. if p_value < 0.05:
17.     print('We reject the null hypothesis and conclude that
    your class and the other class are different in height.')
18. else:
19.     print('We fail to reject the null hypothesis and cannot
    conclude that your class and the other class are different
    in height.')

```

Output:

1. Two-sample test results:
2. t-statistic: 0.7385489458759964
3. p-value: 0.4697020728008009
4. We fail to reject the null hypothesis and cannot conclude that your class and the other class are different in height.

- **Paired and independent tests:** A paired test is when you compare two samples that are related or matched in some way, such as *the average height of men before and after growth hormone treatment*. An independent test is when you compare two samples that are unrelated or random, such as *the average height of men and women*.

For example, imagine you want to know if your class is happier after a field trip. You could ask everyone in your class to rate their happiness before and after the field trip and compare their ratings. A paired test is when you have two sets of data that are linked or matched, such as *my happiness before and after the field trip*. An independent test is when you have two sets of data that

are not linked or matched, such as *my happiness and the happiness of the other class*.

Tutorial 6.7: An illustration of the paired testing using the example *my happiness before and after the field trip*, as mentioned in above example, is as follows:

```
1. # We use scipy.stats.ttest_rel to perform a paired t-test
2. # We assume that the happiness ratings are on a scale of 1 to 10
3. import scipy.stats as stats
4. # The happiness ratings of the class before and after the field trip
5. before = [7, 8, 6, 9, 5, 7, 8, 6, 7, 9]
6. after = [8, 9, 7, 10, 6, 8, 9, 7, 8, 10]
7. # Perform the paired t-test
8. t_stat, p_value = stats.ttest_rel(before, after)
9. # Print the results
10. print("Paired t-test results:")
11. print("t-statistic:", t_stat)
12. print("p-value:", p_value)
```

Output:

```
1. Paired t-test results:
2. t-statistic: -inf
3. p-value: 0.0
```

Tutorial 6.8: An illustration of the independent test using the example *my happiness and the happiness of the other class*, as mentioned in above example, is as follows:

```
1. # We use scipy.stats.ttest_ind to perform an independent t-test
2. # We assume that the happiness ratings of the other classes are also on a scale of 1 to 10
3. import scipy.stats as stats
4. # The happiness ratings of the other class before and after
```

er the field trip

```
5. other_before = [6, 7, 5, 8, 4, 6, 7, 5, 6, 8]
6. other_after = [7, 8, 6, 9, 5, 7, 8, 6, 7, 9]
7. # Perform the independent t-test
8. t_stat, p_value = stats.ttest_ind(after, other_after)
9. # Print the results
10. print("Independent t-test results:")
11. print("t-statistic:", t_stat)
12. print("p-value:", p_value)
```

Output:

```
1. Independent t-test results:
2. t-statistic: 1.698415551216892
3. p-value: 0.10664842826837892
```

- **Parametric and nonparametric tests:** A parametric test is when you assume that your data follow a certain distribution, such as a normal distribution, and you use parameters such as mean and standard deviation to describe your data. A nonparametric test is when you do not assume that your data follow a particular distribution, and you use ranks or counts to describe your data.

For example, imagine you want to know if your class likes chocolate or vanilla ice cream more. You could ask everyone in your class to choose their favorite flavor and count how many people like each flavor. A parametric test is when you assume that your data follow a pattern or shape, such as a bell curve, and you use numbers like mean and standard deviation to describe your data. A nonparametric test is when you do not assume that your data follow a pattern or shape, and you use ranks or counts to describe your data.

Tutorial 6.9: An illustration of the parametric test, as mentioned in above example, is as follows:

```
1. # We use scipy.stats.ttest_ind to perform a parametric t-
```

test

```
2. # We assume that the data follows a normal distribution
3. import scipy.stats as stats
4. # The number of students who like chocolate and vanilla
   ice cream
5. chocolate = [25, 27, 29, 28, 26, 30, 31, 24, 27, 29]
6. vanilla = [22, 23, 21, 24, 25, 26, 20, 19, 23, 22]
7. # Perform the parametric t-test
8. t_stat, p_value = stats.ttest_ind(chocolate, vanilla)
9. # Print the results
10. print("Parametric t-test results:")
11. print("t-statistic:", t_stat)
12. print("p-value:", p_value)
```

Output:

```
1. Parametric t-test results:
2. t-statistic: 5.190169516378603
3. p-value: 6.162927154861931e-05
```

Tutorial 6.10: An illustration of the nonparametric test, as mentioned in above example, is as follows:

```
1. # We use scipy.stats.mannwhitneyu to perform a nonpar
   ametric Mann-Whitney U test
2. # We do not assume any distribution for the data
3. import scipy.stats as stats
4. # The number of students who like chocolate and vanilla
   ice cream
5. chocolate = [25, 27, 29, 28, 26, 30, 31, 24, 27, 29]
6. vanilla = [22, 23, 21, 24, 25, 26, 20, 19, 23, 22]
7. # Perform the nonparametric Mann-Whitney U test
8. u_stat, p_value = stats.mannwhitneyu(chocolate, vanilla)
9. # Print the results
10. print("Nonparametric Mann-Whitney U test results:")
11. print("U-statistic:", u_stat)
```

```
12. print("p-value:", p_value)
```

Output:

1. Nonparametric Mann-Whitney U test results:
2. U-statistic: 95.5
3. p-value: 0.0006480405677249192

Significance testing

Significance testing evaluates the likelihood of a claim or statement about a population being true using data. For instance, it can be used to test if a new medicine is more effective than a placebo or if a coin is biased. The p-value is a measure used in significance testing that indicates how frequently you would obtain the observed data or more extreme data if the claim or statement were false. The smaller the p-value, the stronger the evidence against the claim or statement. Significance testing is different from hypothesis testing, although they are often confused and used interchangeably. Hypothesis testing is a formal procedure for comparing two competing statements or hypotheses about a population, and making a decision based on the data. One of the hypotheses is called the **null hypothesis**, the other hypothesis is called the **alternative hypothesis**, as described above in **hypothesis testing**. Hypothesis testing involves choosing a significance level, which is the maximum probability of making a wrong decision when the null hypothesis is true. Usually, the significance level is set to 0.05. Hypothesis testing also involves calculating a test statistic, which is a number that summarizes the data and measures how far it is from the null hypothesis. Based on the test statistic, a p-value is computed, which is the probability of getting the data (or more extreme) if the null hypothesis is true. If the p-value is less than the significance level, the null hypothesis is rejected and the alternative hypothesis is accepted. If the p-

value is greater than the significance level, the null hypothesis is not rejected and the alternative hypothesis is not accepted.

Suppose, you have a friend who claims to be able to guess the outcome of a coin toss correctly more than half the time, you can test their claim using significance testing. Ask them to guess the outcome of 10-coin tosses and record how many times they are correct. If the coin is fair and your friend is just guessing, you would expect them to be right about 5 times out of 10, on average. However, if they get 6, 7, 8, 9, or 10 correct guesses, how likely is it to happen by chance? The p-value answers the question of the probability of getting the same or more correct guesses as your friend did, assuming a fair coin and random guessing. A smaller p-value indicates a lower likelihood of this happening by chance, and therefore raises suspicion about your friend's claim. Typically, a p-value cutoff of 0.05 is used. If the p-value is less than 0.05, we consider the result statistically significant and reject the claim that the coin is fair, and the friend is guessing. If the p-value is greater than 0.05, we consider the result not statistically significant and do not reject the claim that the coin is fair, and the friend is guessing.

Tutorial 6.11: An illustration of the significance testing, based on above coin toss example, is as follows:

1. *# Import the binom_test function from scipy.stats*
2. **from** scipy.stats **import** binomtest
3. *# Ask the user to input the number of correct guesses by their friend*
4. correct = int(input("How many correct guesses did your friend make out of 10 coin tosses? "))
5. *# Calculate the p-value using the binom_test function*
6. *# The arguments are: number of successes, number of trials, probability of success, alternative hypothesis*

```

7. p_value = binomtest(correct, 10, 0.5, "greater")
8. # Print the p-value
9. print("p-value = {:.4f}".format(p_value.pvalue))
10. # Compare the p-value with the cutoff of 0.05
11. if p_value.pvalue < 0.05:
12.     # If the p-value is less than 0.05, reject the
        claim that the coin is fair and the friend is guessing
13.     print("This result is statistically significant. We
        reject the claim that the coin is fair and the friend
        is guessing.")
14. else:
15.     # If the p-
        value is greater than 0.05, do not reject the claim that th
        e coin is fair and the friend
        is guessing
16.     print("This result is not statistically significant.
        We do not reject the claim that the coin is fair and the
        friend is guessing.")

```

Output: For nine correct guesses, is as follows:

1. How many correct guesses did your friend make out of 10 coin tosses? 9
2. p-value = 0.0107
3. This result is statistically significant.
We reject the claim that the coin is fair and the friend is guessing.

For two correct guesses, the output is not statistically significant as follows:

1. How many correct guesses did your friend make out of 10 coin tosses? 2
2. p-value = 0.9893
3. This result is not statistically significant. We do not reject the claim that the coin is fair and the friend is guessing.

The following is another example to better understand the relation between hypothesis and significance testing. Suppose, you want to know whether a new candy makes children smarter. You have two hypotheses: The null hypothesis is that *the candy has no effect on children's intelligence*. The alternative hypothesis is that *the candy increases children's intelligence*.

You decide to test your hypotheses by giving the candy to 20 children and a placebo to another 20 children. You then measure their IQ scores before and after the treatment. You choose a significance level of 0.05, meaning that you are willing to accept a 5% chance of being wrong if the candy has no effect. You calculate a test statistic, which is a number that tells you how much the candy group improved compared to the placebo group. Based on the test statistic, you calculate a p-value, which is the probability of getting the same or greater improvement than you observed if the candy had no effect.

If the p-value is less than 0.05, you reject the null hypothesis and accept the alternative hypothesis. You conclude that the candy makes the children smarter.

If the p-value is greater than 0.05, you do not reject the null hypothesis and you do not accept the alternative hypothesis. You conclude that the candy has no effect on the children's intelligence.

Tutorial 6.12: An illustration of the significance testing, based on above candy and smartness example, is as follows:

1. `# Import the ttest_rel function from scipy.stats`
2. `from scipy.stats import ttest_rel`
3. `# Define the IQ scores of the candy group before and after the treatment`
4. `candy_before = [100, 105, 110, 115, 120, 125, 130, 135, 140]`
5. `candy_after = [104, 105, 110, 120, 123, 125, 135, 135, 1`

```

44]
6. # Define the IQ scores of the placebo group before and after the treatment
7. placebo_before = [101, 106, 111, 116, 121, 126, 131, 136, 141]
8. placebo_after = [100, 104, 109, 113, 117, 121, 125, 129, 133]
9. # Calculate the difference in IQ scores for each group
10. candy_diff = [candy_after[i] - candy_before[i] for i in range(9)]
11. placebo_diff = [placebo_after[i] - placebo_before[i] for i in range(9)]
12. # Perform a paired t-test on the difference scores
13. # The null hypothesis is that the mean difference is zero
14. # The alternative hypothesis is that the mean difference is positive
15. t_stat, p_value = ttest_rel(candy_diff, placebo_diff, alternative="greater")
16. # Print the test statistic and the p-value
17. print(f"The test statistic is {t_stat:.4f}")
18. print(f"The p-value is {p_value:.4f}")
19. # Compare the p-value with the significance level of 0.05
20. if p_value < 0.05:
21.     # If the p-value is less than 0.05, reject the null hypothesis and accept the alternative hypothesis
22.     print("This result is statistically significant. We reject the null hypothesis and accept the alternative hypothesis.")
23.     print("We conclude that the candy makes the children smarter.")
24. else:

```

```

25.     # If the p-
       value is greater than 0.05, do not reject the null hypothe
       sis and do not accept the alternative hypothesis
26.     print("This result is not statistically significant. We do
       not reject the null hypothesis and do not accept the alter
       native hypothesis.")
27.     print("We conclude that the candy has no effect on th
       e
       children's intelligence.")

```

Output:

1. The test statistic is 5.6127
2. The p-value is 0.0003
3. This result is statistically significant.
We reject the null hypothesis and accept the alternative hypothesis.
4. We conclude that the candy makes the children smarter.

The above output can be changed by changing the p-value, as indicated. The p-value depends on the before and after values.

Steps of significance testing

The steps to perform significance testing in statistics is described by the example below:

- **Question:** Does drinking coffee make you more alert than drinking water?
- **Guess:** There is no difference in alertness between coffee and water. Coffee will make you more alert than water.
- **Chance:** 5%, meaning you are willing to accept a 5% chance of being wrong if there is no difference in alertness between coffee and water.
- **Number:** Suppose -3.2 is test statistic, based on the difference in average alertness scores between two groups of 20 students each who drank coffee or water

before taking a test. The assumed mean scores are 75 and 80, and the standard deviations are 10 and 12, respectively.

- **Probability:** 0.003, which is the probability of getting the same or greater difference in scores than you observed if there is no difference in alertness between coffee and water.
- **Decision:** Since the probability is less than chance, you do not believe the conjecture that there is no difference in alertness between coffee and water, and you believe the conjecture that coffee makes you more alert than water.
- **Answer:** You have strong evidence that coffee makes you more alert than water, with a 5% chance of being wrong. The average difference in alertness is -5, with a assumed range of (-8.6, -1.4).

Further explanation of significance testing along with the *candy makes the children smarter* example, is as follows:

1. **State the claim or statement that you want to test:**
This is usually the research question or the effect of interest.

Claim: A new candy makes the children smarter.

State the null and alternative hypotheses. The null hypothesis is the opposite of the claim or statement, and it usually represents no effect or no difference. The alternative hypothesis is the same as the claim or statement, and it usually represents the effect or difference of interest as follows:

- **Null hypothesis:** The candy has no effect on the children's intelligence, so the mean difference is zero.
- **Alternative hypothesis:** The candy increases the children's intelligence, so the mean difference is positive.

In *Tutorial 6.12*, the following snippet states the claim and hypothesis:

1. *# The null hypothesis is that the mean difference is zero*
2. *# The alternative hypothesis is that the mean difference is positive*
3. `t_stat, p_value = ttest_rel(candy_diff, placebo_diff, alternative="greater")`

2. **Choose a significance level:** This is the maximum probability of rejecting the null hypothesis when it is true. Usually, the significance level is set to 0.05, but it can be higher or lower depending on the context and the consequences of making a wrong decision.

Significance level: 0.05

3. **Choose and compute a test statistic and p-value:** This is a number that summarizes the data and measures how far it is from the null hypothesis. Different types of data and hypotheses require different types of test statistics, such as z, t, F, or chi-square. The test statistic depends on the sample size, the sample mean, the sample standard deviation, and the population parameters.

Test statistic: test statistic is 5.6127.

P-value is the probability of getting the data (or more extreme) if the null hypothesis is true. The p-value depends on the test statistic and the distribution that it follows under the null hypothesis. The p-value can be calculated using formulas, tables, or software.

P-value: p-value is 0.0003.

In *Tutorial 6.12*, the following snippet computes the p-value and test statistic:

1. `t_stat, p_value = ttest_rel(candy_diff, placebo_diff, alternative="greater")`
2. *# Print the test statistic and the p-value*

```
3. print(f"The test statistic is {t_stat:.4f}")
4. print(f"The p-value is {p_value:.4f}")
```

4. **Compare the p-value to the significance level and decide:** If the p-value is less than the significance level, reject the null hypothesis and accept the alternative hypothesis. If the p-value is greater than the significance level, do not reject the null hypothesis and do not accept the alternative hypothesis.

Decision: Since the p-value is less than the significance level, reject the null hypothesis and accept the alternative hypothesis.

In *Tutorial 6.12*, the following snippet compares p-value and significance level:

```
1. # Compare the p-  
   value with the significance level of 0.05  
2. if p_value < 0.05:  
3.   # If the p-value is less than 0.05, reject the null  
   hypothesis and accept the alternative hypothesis  
4.   print("This result is statistically significant. We  
   reject the null hypothesis and accept the alternative  
   hypothesis.")  
5.   print("We conclude that the candy makes the  
   children smarter.")  
6. else:  
7.   # If the p-value is greater than 0.05, do not  
   reject the null hypothesis and do not accept the  
   alternative hypothesis  
8.   print("This result is not statistically  
   significant. We do not reject the null hypothesis and  
   do not  
   accept the alternative hypothesis.")  
9.   print("We conclude that the candy has no  
   effect on the children's intelligence.")
```

5. Interpret the results and draw conclusions: Explain what the decision means in the context of the problem and the data. Address the original claim or statement and the effect of interest. Report the test statistic, the p-value, and the significance level. Discuss the limitations and assumptions of the analysis and suggest possible directions for further research.

Summary: There is sufficient evidence to conclude that the new candy makes children smarter, at the 0.05 significance level.

Types of significance testing

Depending on the data and the hypotheses you want to test, there are different types. Some common types are as follows:

- **T-test:** Compares the means of two independent samples with a continuous dependent variable. For example, you might use a t-test to see if there is a difference in blood pressure (continuous dependent variable) between patients taking a new drug and those taking a placebo.
- **ANOVA:** Compare the means of more than two independent samples with a continuous dependent variable. For example, you can use ANOVA to see if there is a difference in test scores (continuous dependent variable) between students who study using different methods.
- **Chi-square test:** Evaluate the relationship between two categorical variables. For example, you can use a chi-squared test to see if there is a relationship between gender (male/female) and voting preference (A party/B party).
- **Correlation test:** Measures the strength and direction of a linear relationship between two continuous variables. For example, you can use a correlation test to see how height and weight are related.

- **Regression test:** Estimate the effect of one or more predictor (independent) variables on an outcome (dependent) variable. For example, you might use a regression test to see how age, education, and income affect life satisfaction.

Role of p-value and significance level

P-values and **significance levels** are tools that help to decide whether to reject the null hypothesis. A p-value is the probability of getting the data you observe, or more extreme data, if the null hypothesis is true. A significance level is a threshold you choose before the test, usually 0.05 or 0.01.

To illustrate these concepts, consider the example of coin flipping. Suppose, you want to test whether a coin is fair, meaning that it has a 50% chance of landing heads or tails. The null hypothesis is that the coin is fair, and the alternative hypothesis is that the coin is not fair. You decide to flip the coin 10 times and count the number of heads. You also choose a significance level of 0.05 for the test. A significance level of 0.05 indicates that there is a 5% risk of rejecting the null hypothesis if it is true. In other words, you are willing to accept a 5% chance of reaching the wrong conclusion.

You flip the coin 10 times and get 8 heads and 2 tails. Is this result unusual if the coin is fair? To answer this question, you need to calculate the p-value. The p-value is the probability of getting 8 or more heads in 10 flips if the coin is fair. You can use a binomial calculator to find this probability. The p-value is 0.0547, which means that there is a 5.47% chance of getting 8 or more heads in 10 flips when the coin is fair. Now, compare the p-value with the significance level. The p-value is 0.0547, which is slightly greater than the significance level of 0.05. This means that you cannot reject the null hypothesis. You have to say that the data is not enough to prove that the coin is not fair.

Maybe you just got lucky with the tosses, or maybe you need more data to detect a difference.

Tutorial 6.13: To compute the p-value of getting 8 heads and 2 tails when a coin is flipped 10 times, with a significance level of 0.05, as in the example above, is as follows:

```
1. # Import the scipy library for statistical functions
2. import scipy.stats as stats
3. # Define the parameters of the binomial distribution
4. n = 10 # number of flips
5. k = 8 # number of heads
6. p = 0.5 # probability of heads
7. # Calculate the p-
   value using the cumulative distribution function (cdf)
8. # The p-
   value is the probability of getting at least k heads, so we
   use 1 - cdf(k-1)
9. p_value = 1 - stats.binom.cdf(k-1, n, p)
10. # Print the p-value
11. print(f"The p-value is {p_value:.4f}")
12. # Compare the p-value with the significance level
13. alpha = 0.05 # significance level
14. if p_value < alpha:
15.     print("The result is statistically significant.")
16. else:
17.     print("The result is not statistically significant.")
```

Output:

1. The p-value is 0.0547
2. The result is not statistically significant.

The result means that the outcome of the experiment (8 heads and 2 tails) is not very unlikely to occur by chance, assuming the coin is fair. In other words, there is not

enough evidence to reject the null hypothesis that the coin is fair.

Statistical tests

Commonly used statistical tests include the z-test, t-test, and chi-square test, which are typically applied to different types of data and research questions. Each of these tests plays a crucial role in the field of statistics, providing a framework for making inferences and drawing conclusions from data. Z-test, t-test and chi-square test, one-way ANOVA, and two-way ANOVA are used for both hypothesis and assessing significance testing in statistics.

Z-test

The z-test is a statistical test that compares the mean of a sample to the mean of a population or the means of two samples when the population standard deviation is known. It can determine if the difference between the means is statistically significant. For example, you can use a z-test to determine if the average height of students in your class differs from the average height of all students in your school, provided you know the standard deviation of the height of all students. To explain it simply, imagine you have two basketball teams, and you want to know if one team is taller than the other. You can measure the height of each player on both teams, calculate the average height for each team, and then use a z-test to determine if the difference between the averages is significant or just due to chance.

Tutorial 6.14: To illustrate the z-test test, based on above student height example, is as follows:

1. *# import the ztest function from statsmodels package*
2. *from statsmodels.stats.weightstats import ztest*
3. *# create a list of heights (in cm) for each team*
4. *teamA = [180, 182, 185, 189, 191, 191, 192,*

```

194, 199, 199, 205, 209, 209, 209, 210, 212, 212, 213, 2
14, 214]
5. teamB = [190, 191, 191, 191, 195, 195, 199, 199,
208, 209, 209, 214, 215, 216, 217, 217, 228, 229, 230, 2
33]
6. # perform a two sample z-
test to compare the mean heights of the two teams
7. # the null hypothesis is that the mean heights are equal
8. # the alternative hypothesis is that the mean heights are
different
9. # we use a two-
tailed test with a significance level of 0.05
10. z_stat, p_value = ztest(teamA, teamB, value=0)
11. # print the test statistic and the p-value
12. print("Z-statistic:", z_stat)
13. print("P-value:", p_value)
14. # interpret the result
15. if p_value < 0.05:
16.     print("We reject the null hypothesis and conclude that
the mean heights of the two teams are significantly diffe
rent.")
17. else:
18.     print("We fail to reject the null hypothesis and conclu
de that the mean heights of the two teams are not signifi
cantly different.")

```

Output:

```

1. Z-statistic: -2.020774406815312
2. P-value: 0.04330312332391124
3. We reject the null hypothesis and conclude that
the mean heights of the two teams are significantly diffe
rent.

```

This means that, based on the sample data, there is enough evidence to suggest that Team B is, on average, taller than Team A, and that this difference is not due to chance.

T-test

A t-test is a statistical test that compares the mean of a sample to the mean of a population or the means of two samples. It can determine if the difference between the means is statistically significant or not, even when the population standard deviation is unknown and estimated from the sample. Here is a simple example: Suppose, you want to compare the delivery times of two different pizza places. You can order a pizza from each restaurant and record the time it takes for each pizza to arrive. Then, you can use a t-test to determine if the difference between the times is significant or if it could have occurred by chance. Another example is, you can use a t-test to determine whether *the average score of students who took a math test online differs from the average score of students who took the same test on paper*, provided that you are unaware of the standard deviation of the scores of all students who took the test.

Tutorial 6.15: To illustrate the t-test, based on above *student score* example, is as follows:

1. *# import the ztest function from statsmodels package*
2. **from** statsmodels.stats.weightstats **import** ztest
3. *# create a list of delivery times (in minutes) for each pizza place*
4. placeA = [15, 18, 20, 22, 25, 28, 30, 32, 35, 40]
5. placeB = [12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
6. *# perform a two sample z-test to compare the mean delivery times of the two pizza places*
7. *# the null hypothesis is that the mean delivery times are equal*
8. *# the alternative hypothesis is that the mean delivery times are*

different

```
9. # we use a two-  
   #tailed test with a significance level of 0.05  
10. z_stat, p_value = ztest(placeA, placeB, value=0)  
11. # print the test statistic and the p-value  
12. print("Z-statistic:", z_stat)  
13. print("P-value:", p_value)  
14. # interpret the result  
15. if p_value < 0.05:  
16.     print("We reject the null hypothesis and conclude that  
           the mean delivery times of the two pizza places are  
           significantly different.")  
17. else:  
18.     print("We fail to reject the null hypothesis and  
           conclude that the mean delivery times of the two pizza  
           places are not significantly different.")
```

Output:

```
1. Z-statistic: 1.7407039045950503  
2. P-value: 0.08173549351419786  
3. We fail to reject the null hypothesis and conclude that the  
   mean  
   delivery times of the two pizza places are not significantly  
   different.
```

This means that based on the sample data, there is enough evidence to suggest that location B delivers faster than location A on average, and that this difference is not due to chance.

Chi-square test

The chi-square test is a statistical tool that compares observed and expected frequencies of categorical data under a null hypothesis. It can determine if there is a significant association between two categorical variables or

if the distribution of a categorical variable differs from the expected distribution. To determine if there is a relationship between the type of pet a person owns and their favorite color, or if the proportion of people who prefer chocolate ice cream is different from 50%, you can use a chi-square test.

Tutorial 6.16: Suppose, based on the above example of pets and favorite colors, you have data consisting of the observed frequencies of categories in [Table 6.1](#), then implementation of the chi-square test on it, is as follows:

Pet	Red	Blue	Green	Yellow
Cat	12	18	10	15
Dog	8	14	12	11
Bird	5	9	15	6

Table 6.1: *Pet a person owns, and their favorite color observed frequencies*

```
1. # import the chi2_contingency function
2. from scipy.stats import chi2_contingency
3. # create a contingency table as a list of lists
4. data = [[12, 18, 10, 15], [8, 14, 12, 11], [5, 9, 15, 6]]
5. # perform the chi-square test
6. stat, p, dof, expected = chi2_contingency(data)
7. # print the test statistic, the p-value, and the expected frequencies
8. print("Test statistic:", stat)
9. print("P-value:", p)
10. print("Expected frequencies:")
11. print(expected)
12. # interpret the result
13. significance_level = 0.05
14. if p <= significance_level:
15.     print("We reject the null hypothesis and conclude that
```

```
there is a significant association between the type of pet
and the favorite color.")
```

```
16. else:
```

```
17.     print("We fail to reject the null hypothesis and conclu
de that there is no significant association between the ty
pe of pet and the favorite color.")
```

Output:

1. Test statistic: 6.740632143071166
2. P-value: 0.34550083293175876
3. Expected frequencies:
4. [[10.18518519 16.7037037 15.07407407 13.03703704]
5. [8.33333333 13.66666667 12.33333333 10.66666667]
6. [6.48148148 10.62962963 9.59259259 8.2962963]]
7. We fail to reject the null hypothesis and conclude that there is no significant association between the type of pet and the favorite color.

Here, expected frequencies are the theoretical frequencies. We would expect to observe in each cell of the contingency table if the null hypothesis is true. They are calculated based on the row and column sums and the total number of observations. The chi-square test compares the observed frequencies ([Table 6.1](#)) with the expected frequencies (shown in the output) to see if there is a significant difference between them. Based on the sample data, there is insufficient evidence to suggest a correlation between a person's favorite color and the type of pet they own.

Another example is, to determine if a dice is fair, one can use the analogy of a dice game. You can roll the dice many times and count how many times each number comes up. You can use a chi-square test to determine if the observed counts are similar enough to the expected counts, which are equal for a fair dice, or if they differ too much to be attributed to chance. More about chi-square test is also in

One-way ANOVA

A one-way ANOVA is a statistical test that compares the means of three or more groups that have been split on one independent variable. A one-way ANOVA can tell you if there is a significant difference among the group means or not. For example, you can use a one-way ANOVA to see if the average weight of dogs varies by breed, if you have data on the weight of dogs from three or more breeds. Another example is, you can use an analogy of a baking contest to know if the type of flour you use affects the taste of your cake. You can bake three cakes using different types of flour and ask some judges to rate the taste of each cake. Then you can use a one-way ANOVA to see if the average rating of the cakes is different depending on the type of flour, or if they are all similar.

Tutorial 6.17: To illustrate the one-way ANOVA test, based on above baking contest example, is as follows.

```
1. import numpy as np
2. import scipy.stats as stats
3. # Define the ratings of the cakes by the judges
4. cake1 = [8.4, 7.6, 9.2, 8.9, 7.8] # Cake made with flour t
   ype 1
5. cake2 = [6.5, 5.7, 7.3, 6.8, 6.4] # Cake made with flour t
   ype 2
6. cake3 = [7.1, 6.9, 8.2, 7.4, 7.0] # Cake made with flour t
   ype 3
7. # Perform one-way ANOVA
8. f_stat, p_value = stats.f_oneway(cake1, cake2, cake3)
9. # Print the results
10. print("F-statistic:", f_stat)
11. print("P-value:", p_value)
```


Output:

1. F-statistic: 11.716117216117217
2. P-value: 0.001509024295003377

The p-value is very small, which means that we can reject the null hypothesis that the means of the ratings are equal. This suggests that the type of flour affects the taste of the cake.

Two-way ANOVA

A two-way ANOVA is a statistical test that compares the means of three or more groups split on two independent variables. It can determine if there is a significant difference among the group means, if there is a significant interaction between the two independent variables, or both. For example, if you have data on the blood pressure of patients from different genders and age groups, you can use a two-way ANOVA to determine if the average blood pressure of patients varies by gender and age group. Another example is, analogy of a science fair project. Imagine, you want to find out if the type of music you listen to and the time of day you study affect your memory. Volunteers can be asked to memorize a list of words while listening to different types of music (such as classical, rock, or pop) at various times of the day (such as morning, afternoon, or evening). Their recall of the words can then be tested, and their memory score measured. A two-way ANOVA can be used to determine if the average memory score of the volunteers differs depending on the type of music and time of day, or if there is an interaction between these two factors. For instance, it may show, listening to classical music may enhance memory more effectively in the morning than in the evening, while rock music may have the opposite effect.

Tutorial 6.18: The implementation of two-way ANOVA test, based on above baking contest example, is as follows:

```

1. import pandas as pd
2. import statsmodels.api as sm
3. from statsmodels.formula.api import ols
4. from statsmodels.stats.anova import anova_lm
5. # Define the data
6. data = {"music": ["classical", "classical", "classical", "classical", "classical",
7.                  "rock", "rock", "rock", "rock", "rock",
8.                  "pop", "pop", "pop", "pop", "pop"],
9.         "time": ["morning", "morning", "afternoon", "afternoon", "evening",
10.                 "morning", "morning", "afternoon", "afternoon", "evening",
11.                 "morning", "morning", "afternoon", "afternoon", "evening"],
12.         "score": [12, 14, 11, 10, 9,
13.                  8, 7, 9, 8, 6,
14.                  10, 11, 12, 13, 14]}
15. # Create a pandas DataFrame
16. df = pd.DataFrame(data)
17. # Perform two-way ANOVA
18. model = ols("score ~ C(music) + C(time) + C(music):C(time)", data=df).fit()
19. aov_table = anova_lm(model, typ=2)
20. # Print the results
21. print(aov_table)

```

Output:

	sum_sq	df	F	PR(>F)
1. C(music)	54.933333	2.0	36.622222	0.000434
3. C(time)	1.433333	2.0	0.955556	0.436256
4. C(music):C(time)	24.066667	4.0	8.022222	0.013788
5. Residual	4.500000	6.0	NaN	NaN

Since the p-value for music is less than 0.05, the music has a significant effect on memory score, while time has no significant effect. And since the p-value for the interaction effect (0.013788) is less than 0.05, this tells us that there is a significant interaction effect between music and time.

Hypothesis and significance testing in diabetes dataset

Let us use the diabetes dataset, containing information on 768 patients. Out of it, let us take **body mass index (BMI)** and outcome (whether they have diabetes or not) where 0 means no diabetes and 1 means diabetes.

Now, to perform testing, we will define a research question in the form of a hypothesis, as follows:

- **Null hypothesis:** The mean BMI of diabetic patients is equal to the mean BMI of non-diabetic patients.
- **Alternative hypothesis:** The mean BMI of diabetics is not equal to the mean BMI of non-diabetics.

Tutorial 6.19: The implementation of hypothesis testing and significance on diabetes dataset to test is as follows:

```
1. import pandas as pd
2. from scipy import stats
3. # Load the diabetes data from a csv file
4. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
5. # Null hypothesis: There is a significant difference in the
   mean BMI of diabetic and non-diabetic patients
6. # Separate the BMI values for diabetic and non-
   diabetic patients
7. bmi_diabetic = data[data["Outcome"] == 1]["BMI"]
8. bmi_non_diabetic = data[data["Outcome"] == 0]["BMI"]
9. # Perform a two-sample t-
   test to compare the means of the two groups
```

```

10. t, p = stats.ttest_ind(bmi_diabetic, bmi_non_diabetic)
11. # Print the test statistic and the p-value
12. print("Test statistic:", t)
13. print("P-value:", p)
14. # Set a significance level
15. alpha = 0.05
16. # Compare the p-value with the significance level and make a decision
17. if p <= alpha:
18.     print("We reject the null hypothesis and conclude that
        there is a significant difference in the mean BMI of diabetic
        and non-diabetic patients.")
19. else:
20.     print("We fail to reject the null hypothesis and conclude that
        there is not enough evidence to support a significant difference
        in the mean BMI of diabetic and non-diabetic patients.")

```

Output:

1. Test statistic: 8.47183994786525
2. P-value: 1.2298074873116022e-16
3. We reject the null hypothesis and conclude that there is a significant difference in the mean BMI of diabetic and non-diabetic patients.

The output shows the mean BMI of diabetics is not equal to the mean BMI of non-diabetics, which means the BMI of diabetic and non-diabetic person is different.

Tutorial 6.20: To measure if there is an association between the number of pregnancies and the outcome, we define **null hypothesis:** *There is no association between the number of pregnancies and the outcome (diabetic and non-diabetic patients).* **Alternative hypothesis:** *There is association between the number of pregnancies and the*

outcome (diabetic and non-diabetic patients). Then the implementation of hypothesis testing and the significance on diabetes dataset, is as follows:

```
1. import pandas as pd
2. from scipy import stats
3. # Load the diabetes data from a csv file
4. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter1/diabetes.csv")
5. # Separate the number of pregnancies and the outcome for each patient
6. pregnancies = data["Pregnancies"]
7. outcome = data["Outcome"]
8. # Perform a chi-square test to test the independence of the two variables
9. chi2, p, dof, expected = stats.chi2_contingency(pd.crosstab(pregnancies, outcome))
10. # Print the test statistic and the p-value
11. print("Test statistic:", chi2)
12. print("P-value:", p)
13. # Set a significance level
14. alpha = 0.05
15. # Compare the p-value with the significance level and make a decision
16. if p <= alpha:
17.     print("We reject the null hypothesis and conclude that there is a significant association between the number of pregnancies and the outcome.")
18. else:
19.     print("We fail to reject the null hypothesis and conclude that there is not enough evidence to support a significant association between the number of pregnancies and the outcome.")
```

Output:

1. Test statistic: 64.59480868723006
2. P-value: 8.648349123362548e-08
3. We reject the null hypothesis and conclude that there is a significant association between the number of pregnancies and the outcome.

Sampling techniques and sampling distributions

Sampling techniques involve selecting a subset of individuals or items from a larger population. Sampling distributions display how a sample statistic, such as the mean, proportion, or standard deviation, varies across many random samples from the same population. These techniques and distributions are used in statistics to make inferences or predictions about the entire population based on the sample data. To determine the average height of all students in your school, measuring each student's height would be impractical and time-consuming. Instead, you can use a sampling technique, such as simple random sampling, to select a smaller group of students, for example 100, and measure their heights. This smaller group is called a **sample**, and the average height of this sample is called a **sample mean**.

Imagine repeating this process multiple times, selecting a different random sample of 100 students each time, and calculating their average height. Each sample is different, resulting in different sample means. Plotting all these sample means on a graph creates a sampling distribution of the sample mean. This graph will show how the sample mean varies across different samples and the most likely value of the sample mean.

The sampling distribution of the sample mean has several interesting properties. One of these is that its mean is equal to the population mean. This implies that the average of all the sample means is the same as the average of all the students in the school. Additionally, the shape of the

sampling distribution of the sample mean approaches a bell curve (also known as a normal distribution) as the sample size increases. The central limit theorem enables us to use the normal distribution to predict the population mean based on the sample mean.

Tutorial 6.21: A simple illustration of the sampling technique using 15 random numbers, is as follows:

```
1. import random
2. # Sampling technique
3. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
4. sample_size = 5
5. sample = random.sample(data, sample_size)
6. print(f"The sample of size {sample_size} is: {sample}")
```

Output:

```
1. The sample of size 5 is: [8, 11, 9, 14, 4]
```

Tutorial 6.22: A simple illustration of the sampling distribution using 1000 samples of size 5 generated from a list of 15 integers. We then calculate the mean of each sample and store it in a list, as follows:

```
1. import random
2. # Sampling distribution
3. sample_size = 5
4. num_samples = 1000
5. data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
6. sample_means = []
7. for i in range(num_samples):
8.     sample = random.sample(data, sample_size)
9.     sample_mean = sum(sample) / sample_size
10.    sample_means.append(sample_mean)
11. print(f"The mean of the sample means is: {sum(sample_means) / num_samples}")
```

Output:

```
1. The mean of the sample means is: 8.0060000000000002
```

Further to understand it in simple words, let us take another example of *rolling dice*. To determine the average number of dots when rolling a die, we must first define the population as the set of all possible outcomes: To determine the average number of dots when rolling a die, we must first define the population as the set of all possible outcomes: To determine the average number of dots when rolling a die, we must first define the population as the set of all possible outcomes: 1, 2, 3, 4, 5, and 6. To determine the average number of dots when rolling a die, we must first define the population as the set of all possible outcomes. By using this sample, we can estimate the population mean. In case of rolling a die population mean is 3.5, however, since it is impossible to roll a 3.5, we need to use a sample to estimate it. One method is to roll the die once and record the number of dots. This is a sample of size 1. The sample mean is equal to the number of dots. If you repeat this process multiple times, you will obtain different sample means each time, ranging from 1 to 6.

Plotting these sample means on a graph will result in a sampling distribution of the sample mean that appears as a flat line, with equal chances of obtaining any number from 1 to 6. However, this sampling distribution is not very informative as it does not provide much insight into the population mean. One way to obtain a sample of size 2 is by rolling a die twice and adding up the dots. The sample mean is then calculated by dividing the sum of the dots by 2. If this process is repeated multiple times, different sample means will be obtained, each with a probability of occurrence. The probabilities range from 1 to 6, depending on the sample mean. For instance, the probability of obtaining a sample mean of 2 is $1/36$, as it requires rolling two ones, which has a probability of $1/6$ multiplied by $1/6$. The probability of obtaining a sample mean of 3 is $2/36$. This is because you can roll a one and a two, or a two and a one, which has a probability of $2/6$ times $1/6$.

If you plot these sample means on a graph, you will get a sampling distribution of the sample mean that looks like a triangle. The distribution has higher chances of obtaining numbers closer to 3.5. This sampling distribution is more useful because it indicates that the population mean is more likely to be around 3.5 than around 1 or 6. To increase the sample size, roll a die three or more times and calculate the sample mean each time. As the sample size increases, the sampling distribution of the sample mean becomes more bell-shaped, with a narrower and taller curve, indicating greater accuracy and consistency. The central limit theorem is demonstrated here, allowing you to predict the population mean using the normal distribution based on the sample mean.

For instance, if you roll a die 30 times and obtain a sample mean of 3.8, you can use the normal distribution to determine the likelihood that the population mean falls within a specific range of 3.5 to 4.1. This is a confidence interval. It provides an idea of how certain you are that your sample mean is close to the population mean. The confidence interval becomes narrower with a larger sample size, increasing your confidence.

Tutorial 6.23: To explore sampling distributions and confidence intervals through dice rolls, is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Roll Dice
4. def roll_die(num_rolls):
5.     return np.random.randint(1, 7, num_rolls)
6. # Function to generate sample means for rolling dice
7. def dice_sample_means(num_rolls, num_samples):
8.     means = []
9.     for _ in range(num_samples):
10.         sample = roll_die(num_rolls)
```

```

11.     means.append(np.mean(sample))
12.     return means
13. # Generate sampling distribution for rolling a die
14. num_rolls = 30
15. num_samples = 1000
16. dice_means = dice_sample_means(num_rolls, num_samples)
17. # Convert dice_means to a NumPy array
18. dice_means = np.array(dice_means)
19. # Plotting the sampling distribution of the sample mean for dice rolls
20. plt.figure(figsize=(10, 6))
21. plt.hist(dice_means, bins=30, density=True, alpha=0.6, color='b')
22. plt.axvline(3.5, color='r', linestyle='--')
23. plt.title('Sampling Distribution of the Sample Mean (Dice Rolls)')
24. plt.xlabel('Sample Mean')
25. plt.ylabel('Frequency')
26. plt.show()
27. # Confidence Interval Example
28. sample_mean = np.mean(dice_means)
29. sample_std = np.std(dice_means)
30. # Calculate 95% confidence interval
31. conf_interval = (sample_mean - 1.96 * (sample_std / np.sqrt(num_rolls)),
32.                  sample_mean + 1.96 * (sample_std / np.sqrt(num_rolls)))
33. print(f"Sample Mean: {sample_mean}")
34. print(f"95% Confidence Interval: {conf_interval}")

```

Output:

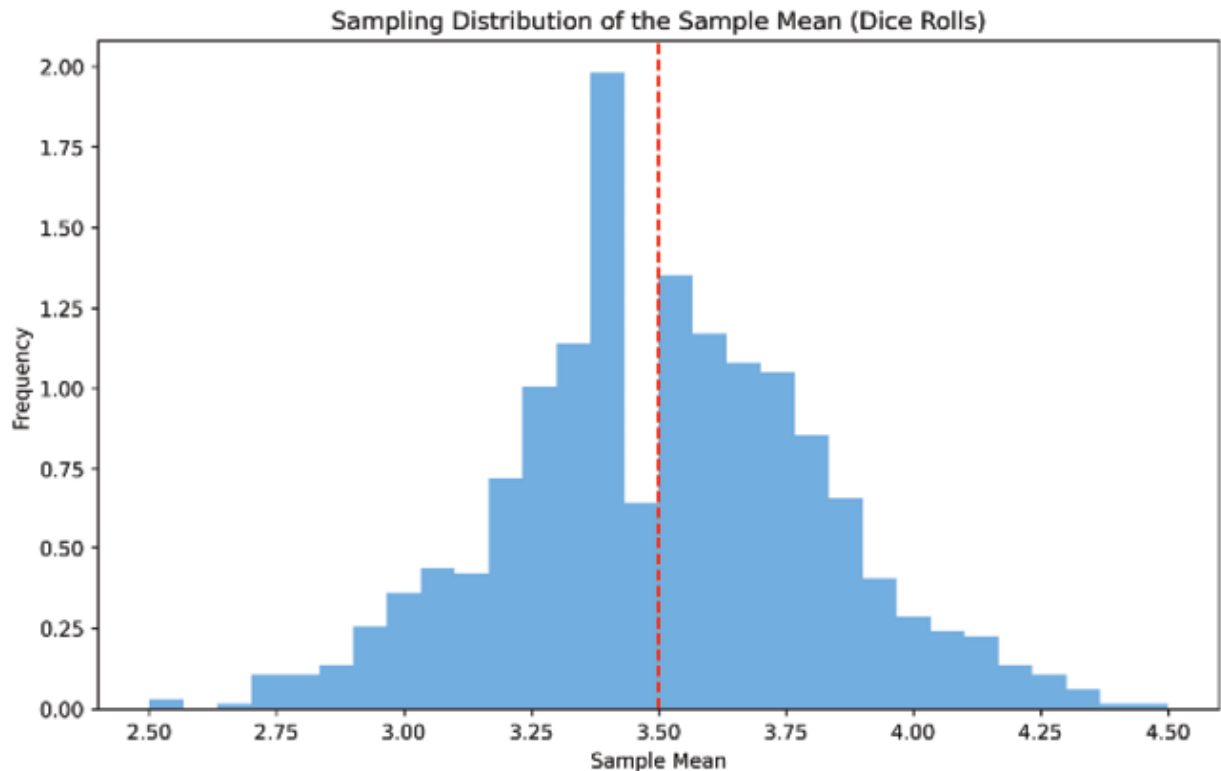


Figure 6.1: Sampling distribution of the sample mean

Conclusion

In this chapter, we learned about the concept and process of hypothesis testing, which is a statistical method for testing whether or not a statement about a population parameter is true. Hypothesis testing is important because it allows us to draw conclusions from data and test the validity of our claims.

We also learned about significance tests, which are used to evaluate the strength of evidence against the null hypothesis based on the p-value and significance level. Significance testing uses the p-value and significance level to determine whether the observed effect is statistically significant, meaning that it is unlikely to occur by chance. We explored different types of statistical tests, such as z-test, t-test, chi-squared test, one-way ANOVA, and two-way

ANOVA, and how to choose the appropriate test based on the research question, data type, and sample size. We also discussed the importance of sampling techniques and sampling distributions, which are essential for conducting valid and reliable hypothesis tests. To illustrate the application of hypothesis testing, we conducted two examples using a diabetes dataset. The first example tested the null hypothesis that the mean BMI of diabetic patients is equal to the mean BMI of non-diabetic patients using a two-sample t-test. The second example tests the null hypothesis that there is no association between the number of pregnancies and the outcome (diabetic versus non-diabetic) using a chi-squared test.

Chapter 7, Statistical Machine Learning discusses the concept of machine learning and how to apply it to make artificial intelligent models and evaluate them.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Statistical Machine Learning

Introduction

Statistical **Machine Learning (ML)** is a branch of **Artificial Intelligence (AI)** that combines statistics and computer science to create models that can learn from data and make predictions or decisions. Statistical machine learning has many applications in fields as diverse as computer vision, speech recognition, bioinformatics, and more.

There are two main types of learning problems: supervised and unsupervised learning. Supervised learning involves learning a function that maps inputs to outputs, based on a set of labeled examples. Unsupervised learning involves discovering patterns or structure in unlabeled data, such as clustering, dimensionality reduction, or generative modeling. Evaluating the performance and generalization of different machine learning models is also important. This can be done using methods such as cross-validation, bias-variance tradeoff, and learning curves. And sometimes when supervised and unsupervised are not useful semi and self-supervised techniques may be useful. This chapters cover

only supervised machine learning, semi-supervised and self-supervised learning. Topics covered in this chapter are listed in the *Structure* section below.

Structure

In this chapter, we will discuss the following topics:

- Machine learning
- Supervised learning
- Model selection and evaluation
- Semi-supervised and self-supervised leanings
- Semi-supervised techniques
- Self-supervised techniques

Objectives

By the end of this chapter, readers will be introduced to the concept of machine learning, its types, and the topic associated with supervised machine learning with simple examples and tutorials. At the end of this chapter, you will have a solid understanding of the principles and methods of statistical supervised machine learning and be able to apply and evaluate them to various real-world problems.

Machine learning

ML is a prevalent form of AI. It powers many of the digital goods and services we use daily. Algorithms trained on data sets create models that enable machines to perform tasks that would otherwise only be possible for humans. Deep learning is also popular subbranch of machine learning that uses neural networks with multiple layers. Facebook uses machine learning to suggest friends, pages, groups, and events based on your activities, interests, and preferences. Additionally, it employs machine learning to detect and

remove harmful content, such as hate speech, misinformation, and spam. Amazon, on the other hand, utilizes machine learning to analyze your browsing history, purchase history, ratings, reviews, and other factors to suggest products that may interest or benefit you. In healthcare it is used to detect cancer, diabetes, heart disease, and other conditions from medical images, blood tests, and other data sources. It can also monitor patient health, predict outcomes, and suggest optimal treatments and many more. Types of learning include supervised, unsupervised, reinforcement, self-supervised, and semi-supervised.

Understanding machine learning

ML allows computers to learn from data and do things that humans can do, such as recognize faces, play games, or translate languages. As mentioned above, it uses special rules called algorithms that can find patterns in the data and use them to make predictions or decisions. For example, if you want to teach a computer to recognize cats, provide it with numerous pictures of cats and other animals, and indicate which ones are cats and which ones are not. The computer will use an algorithm to learn the distinguishing features of a cat, such as the shape of its ears, eyes, nose, and whiskers. When presented with a new image, it can use the learned features to determine if it is a cat or not. This is how machine learning works.

ML is an exciting field that has enabled us to accomplish incredible feats, such as identifying faces in a swimming pool or teaching robots new skills. It is an intelligent technology that learns from data, allowing it to improve every day, from playing games of darts to driving on the highway. It is also a source of inspiration, encouraging curiosity and creativity, whether it's drawing a smiling sun or writing a descriptive poem. Additionally, many of us are

familiar with ChatGPT, which is also powered by data, statistics, and machine learning.

Role of data, algorithm, statistics

Data, algorithms, and statistics are the three main components of machine learning. And as we know about these. Let us try to understand their roles with an example. Suppose we want to create a machine learning model that can classify emails as spam or not spam. The role of data here is that first we need a dataset of emails that are labeled as spam or not spam. This is our data. Then we need to choose an algorithm that can learn from the labeled data and predict the labels for new emails. This can be a supervised algorithm like logistic regression, decision tree, or neural network. This is our algorithm. Along with these two, we need to use statistics to evaluate the performance of our algorithm on the data. We can use metrics such as accuracy, precision, recall, or F1 score to measure how well our algorithm can classify emails as spam or not spam. We can also use statistics to tune the parameters of our algorithm, such as the learning rate, the number of layers, or the activation function. These are our statistics. This is how data, algorithm, and statistics play a role in machine learning. We further discuss this a lot in this chapter with tutorials and examples.

Inference, prediction and fitting models to data

ML has two common applications: inference and prediction. These require different approaches and considerations. It is important to note that inference and prediction are two different goals of machine learning. Inference involves using a model to learn about the relationship between input and output variables. It includes the effect of each feature on the outcome, the uncertainty of the estimates, or the causal mechanisms behind the data. Prediction involves using a

model to forecast the output for new or unseen input data. This can include determining the probability of an event, classifying an image, or recommending a product.

Fitting models to data is a general process that applies to both inference and prediction. The specific approach can vary depending on the problem and data. By fitting models to data, we can identify the best model to represent the data and perform the desired task, whether it be inference or prediction. Fitting models to data involves choosing the type of model, the parameters of the model, the evaluation metrics, and the validation methods.

Supervised learning

Supervised learning uses labeled data sets to train algorithms to classify data or predict outcomes accurately. For example, using labeled data of dogs and cats to train a model to classify them, sentiment analysis, hospital readmission prediction, spam email filtering.

Fitting models to independent data

Fitting models to independent data involves data points that are not related to each other. The model does not consider any correlation or dependency between them. For example, when fitting a linear regression model to the height and weight of different people, we can assume that one person's height and weight are independent of another person. Fitting models to independent data is more common and easier than fitting models to dependent data. Another example is, suppose you want to find out how the number of study hours affects test scores. You collect data from 10 students and record how many hours they studied and what score they got on the test. You want to fit a model that can predict the test score based on the number of hours studied. This is an example of fitting models to independent data, because one student's hours and test score are not related

to another student's hours and test score. You can assume that each student is different and has his or her own study habits and abilities.

Tutorial 7.1: To implement and illustrate the concept of fitting models to independent data, is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Define the data
4. x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) # Number of h
    ours studied
5. y = np.array([50, 60, 65, 70, 75, 80, 85, 90, 95, 100]) #
    Test score
6. # Fit the linear regression model
7. m, b = np.polyfit(x, y, 1) # Find the slope and the interc
    ept
8. # Print the results
9. print(f"The slope of the line is {m:.2f}")
10. print(f"The intercept of the line is {b:.2f}")
11. print(f"The equation of the line is y = {m:.2f}x + {b:.2f}")
12. # Plot the data and the line
13. # Data represent the actual values of the number of hou
    rs studied and the test score for each student
14. # Line represents the linear regression model that predi
    cts the test score based on the number of hours studied
15. plt.scatter(x, y, color="blue", label="Data") # Plot the d
    ata points
16. plt.plot(x, m*x + b, color="red", label="Linear regressio
    n model") # Plot the line
17. plt.xlabel("Number of hours studied") # Label the x-axis
18. plt.ylabel("Test score") # Label the y-axis
19. plt.legend() # Show the legend
20. plt.savefig('fitting_models_to_independent_data.jpg',dpi
    =600,bbox_inches='tight') # Show the figure
```

```
21. plt.show() # Show the plot
```

Output:

1. The slope of the line is 5.27
2. The intercept of the line is 48.00
3. The equation of the line is $y = 5.27x + 48.00$

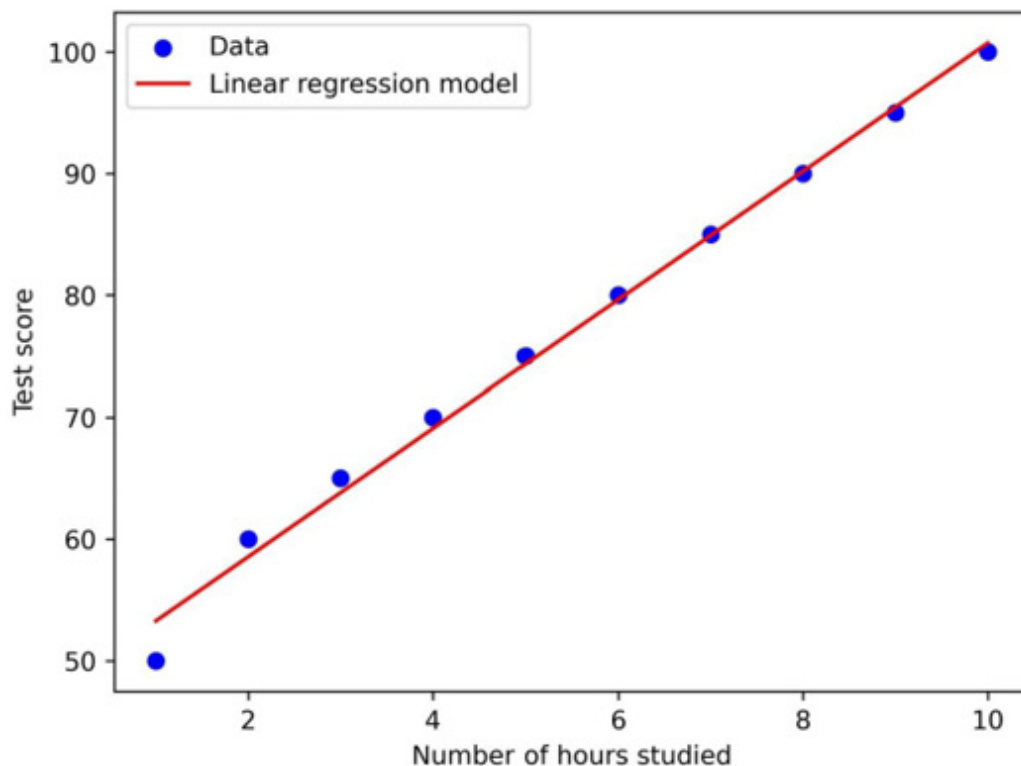


Figure 7.1: Plot fitting number of hours studies and test score

In [Figure 7.1](#), the data (dots) points represent the actual values of the number of hours studied and the test score for each student and the red line represents the fitted linear regression model that predicts the test score based on the number of hours studied. [Figure 7.1](#) shows that the line fits the data well and that the student's test score increases by almost five points for every hour they study. The line also predicts that if students did not study at all, their score would be around 45.

Linear regression

Linear regression uses linear models to predict the target variable based on the input characteristics. A linear model is a mathematical function that assumes a linear relationship between the variables, meaning that the output can be expressed as a weighted sum of the inputs plus a constant term. For example, a linear model could be used to predict the price of a house based on its size and location can be represented as follows:

$$price = w1 * size + w2 * location + b$$

Where $w1$ and $w2$ are the weights or coefficients that measure the influence of each feature on the price, and b is the bias or intercept that represents the base price.

Before moving to the tutorials let us look at the syntax for implementing linear regression with **sklearn**, which is as follows:

1. *# Import linear regression*
2. `from sklearn.linear_model import LinearRegression`
3. *# Create a linear regression model*
4. `linear_regression = LinearRegression()`
5. *# Train the model*
6. `linear_regression.fit(X_train, y_train)`

Tutorial 7.2: To implement and illustrate the concept of linear regression models to fit a model to predict house price based on size and location as in the example above, is as follows:

1. *# Import the sklearn linear regression library*
2. `import sklearn.linear_model as lm`
3. *# Create some fake data*
4. `x = [[50, 1], [60, 2], [70, 3], [80, 4], [90, 5]]`
Size and location of the houses
5. `y = [100, 120, 140, 160, 180]` *# Price of the houses*
6. *# Create a linear regression model*

```

7. model = lm.LinearRegression()
8. # Fit the model to the data
9. model.fit(x, y)
10. # Print the intercept (b) and the slope (w1 and w2)
11. print(f"Intercept: {model.intercept_}") # b
12. print(f"Coefficient/Slope: {model.coef_}") # w1 and w2
13. # Predict the price of a house with size 75 and location
    3
14. print(f"Prediction: {model.predict([[75, 3]])}") # y

```

Output:

```

1. Intercept: 0.7920792079206933
2. Coefficient/Slope: [1.98019802 0.1980198 ]
3. Prediction: [149.9009901]

```

Now let us see how the above fitted house price prediction model looks like in a plot.

Tutorial 7.3: To visualize the fitted line in *Tutorial 7.2* and the data points in a scatter plot, is as follows:

```

1. import matplotlib.pyplot as plt
2. # Extract the x and y values from the data
3. x_values = [row[0] for row in x]
4. y_values = y
5. # Plot the data points as a scatter plot
6. plt.scatter(x_values, y_values, color="blue", label="Data
    points")
7. # Plot the fitted line as a line plot
8. plt.plot(x_values, model.predict(x), color="red", label="
    Fitted linear regression model")
9. # Add some labels and a legend
10. plt.xlabel("Size of the house")
11. plt.ylabel("Price of the house")
12. plt.legend()

```

13. `plt.savefig('fitting_models_to_independent_data.jpg',dpi=600,bbox_inches='tight')` *# Show the figure*
14. `plt.show()` *# Show the plot*

Output:

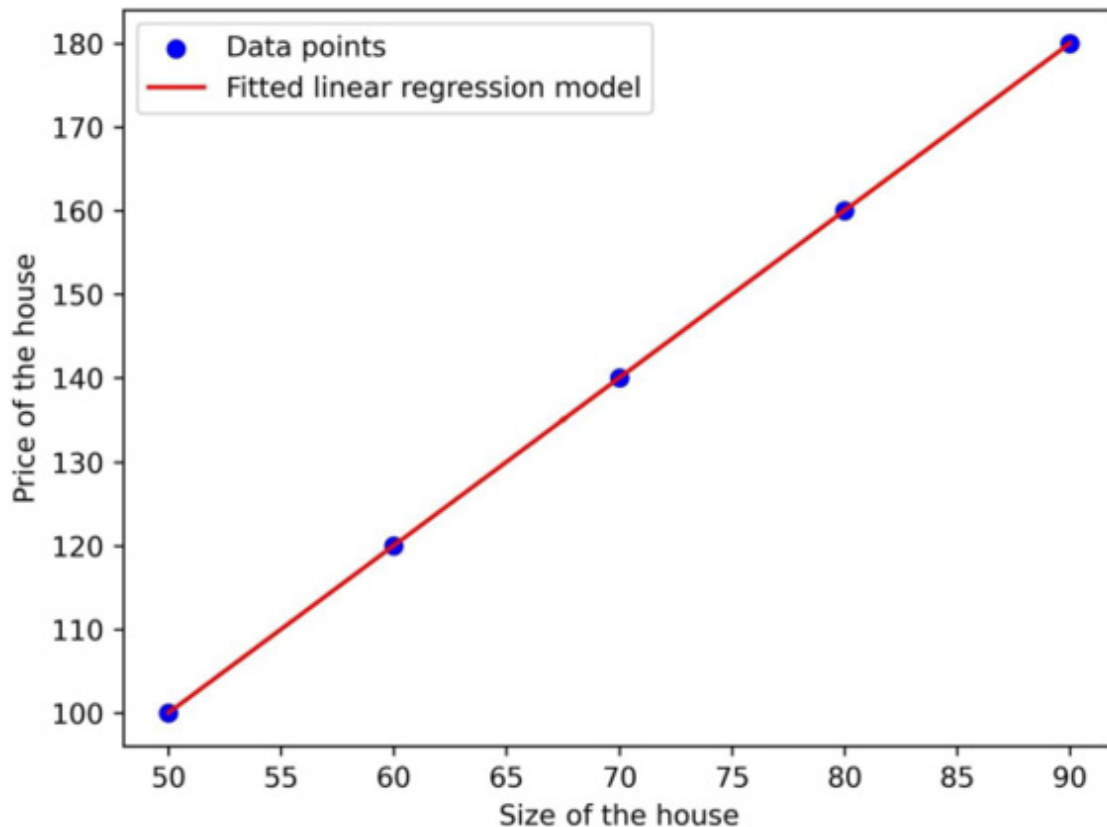


Figure 7.2: Plot fitting size of house and price of house

Linear regression is a suitable method for analyzing the relationship between a numerical outcome variable and one or more numerical or categorical characteristics. It is best used for data that exhibit a linear trend, where the change in the dependent variable is proportional to the change in the independent variables. If the data is non-linear as shown in [Figure 7.3](#), linear regression may not be the most appropriate method, logistic regression, neural network and other algorithms may be more suitable. Linear regression is not suitable for data that follows a curved pattern, such as an exponential or logarithmic function, as it will not be able

to capture the true relationship and will produce a poor fit.

Tutorial 7.4: To show a scatter plot where data follow curved pattern, is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Some data that follows a curved pattern
4. x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
5. y = np.sin(x)
6. # Plot the data as a scatter plot
7. plt.scatter(x, y, color='blue', label='Data')
8. # Fit a polynomial curve to the data
9. p = np.polyfit(x, y, 6)
10. y_fit = np.polyval(p, x)
11. # Plot the curve as a red line
12. plt.plot(x, y_fit, color='red', label='Curve')
13. # Add some labels and a legend
14. plt.xlabel('X')
15. plt.ylabel('Y')
16. plt.legend()
17. # Save the figure
18. plt.savefig('scatter_curve.png', dpi=600, bbox_inches='tight')
19. plt.show()
```

Output:

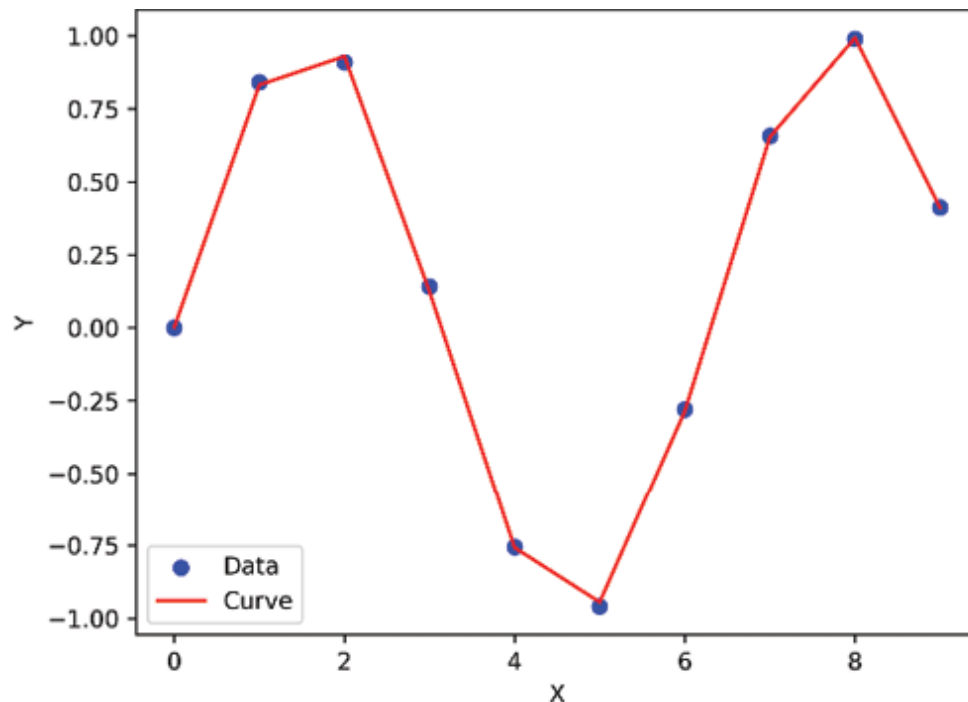


Figure 7.3: Plot where X and Y data form a curved pattern line

Therefore, it is important to check the assumptions of linear regression before applying it to the data, such as linearity, normality, homoscedasticity, and independence. Linearity can be easily viewed by plotting the data and looking for a linear pattern as shown in [Figure 7.4](#).

Tutorial 7.5: To implement viewing of the linearity (linear pattern) in the data by plotting the data in a scatterplot, as follows:

1. `import matplotlib.pyplot as plt`
2. `# Define the x and y variables`
3. `x = [1, 2, 3, 4, 5, 6, 7, 8]`
4. `y = [2, 4, 6, 8, 10, 12, 14, 16]`
5. `# Create a scatter plot`
6. `plt.scatter(x, y, color="red", marker="o")`
7. `# Add labels and title`
8. `plt.xlabel("x")`
9. `plt.ylabel("y")`
10. `plt.title("Linear relationship between x and y")`


```
11. # Save the figure
12. plt.savefig('linearity.png', dpi=600, bbox_inches='tight')
13. plt.show()
```

Output:

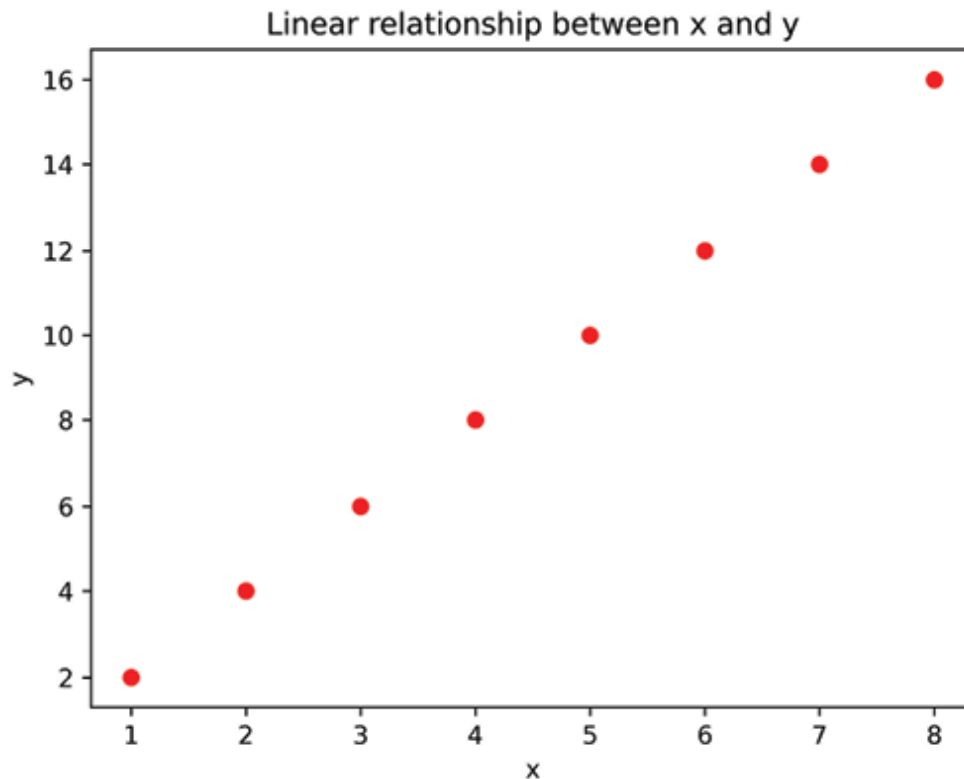


Figure 7.4: Plot showing linearity (linear pattern) in the data

It is also important that the residuals (the differences between the observed and predicted values) are normally distributed, have equal variances (homoscedasticity), and are independent of each other.

Tutorial 7.6: To check the normality of data, is as follows:

1. `import matplotlib.pyplot as plt`
2. `import statsmodels.api as sm`
3. *# Define data*
4. `x = [1, 2, 3, 4, 5, 6, 7, 8]`
5. `y = [2, 4, 6, 8, 10, 12, 14, 16]`
6. *# Fit a linear regression model using OLS*

```
7. model = sm.OLS(y, x).fit() # Create and fit an OLS object
8. # Get the predicted values
9. y_pred = model.predict()
10. # Calculate the residuals
11. residuals = y - y_pred
12. # Plot the residuals
13. plt.scatter(y_pred, residuals, alpha=0.5)
14. plt.title('Residual Plot')
15. plt.xlabel('Predicted values')
16. plt.ylabel('Residuals')
17. # Save the figure
18. plt.savefig('normality.png', dpi=600, bbox_inches='tight')
19. plt.show()
```

sm.OLS() from the **statsmodels** module that performs **ordinary least squares (OLS)** regression, which is a method of finding the best-fitting linear relationship between a dependent variable and one or more independent variables.

The **output** is [Figure 7.5](#), it does not fulfill the normality test or indicate that the residuals are normally distributed. It is a perfect fit, where the predicted values match exactly the observed values, and the residuals are all zero as follows:

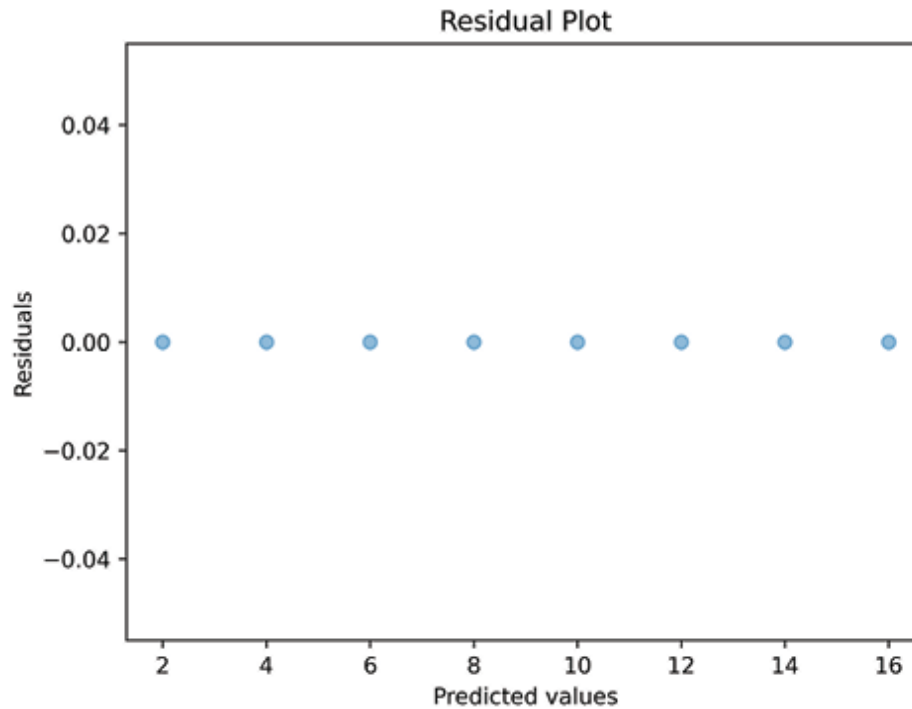


Figure 7.5: Plot to view the normality in the data

Further to check homoscedasticity create a scatter plot of the residuals and the predicted values to visually check if the residuals have constant variance at every level of the independent variables. Where independence means that the error for one observation does not affect the error for another observation, and is more useful to see for time-series data.

Logistic regression

Logistic regression is a type of statistical model that estimates the probability of an event occurring based on a given set of independent variables. It is often used for classification and predictive analytics, such as predicting whether an email is spam or not, or whether a customer will default on a loan or not. Logistic regression predicts the probability of an event or outcome using a set of predictor variables based on the concept of a logistic (sigmoid) function mapping a linear combination into a probability score between 0 and 1. Here, the predicted probability can

be used to classify the observation into one of the categories by choosing a cutoff value. For example, if the probability is greater than 0.5, the observation is classified as a success, otherwise it is classified as a failure.

For example, a simple example of logistic regression is to predict whether a student will pass an exam based on the number of hours they studied. Suppose we have the following data:

Hours studied	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
Passed	0	0	0	0	0	1	1	1	1	1

Table 7.1: Hours studied and student exam result

We can fit a logistic regression model to this data, using hours studied as the independent variable and passed as the dependent variable.

Before moving to the tutorials let us look at the syntax for implementing logistic regression with **sklearn**, which is as follows:

1. *# Import logistic regression*
2. `from sklearn.linear_model import LogisticRegression`
3. *# Create a logistic regression model*
4. `logistic_regression = LogisticRegression()`
5. *# Train the model*
6. `logistic_regression.fit(X_train, y_train)`

Tutorial 7.7: To implement logistic regression based on above example, to predict whether a student will pass an exam based on the number of hours they studied, is as follows:

1. `import numpy as np`
2. `import pandas as pd`
3. *# Import libraries from sklearn for logistic regression prediction*
4. `from sklearn.linear_model import LogisticRegression`

```

5. # Create the data
6. data = {"Hours studied": [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5],
7.         "Passed": [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]}
8. df = pd.DataFrame(data)
9. # Define the independent and dependent variables
10. X = df["Hours studied"].values.reshape(-1, 1)
11. y = df["Passed"].values
12. # Fit the logistic regression model
13. model = LogisticRegression()
14. model.fit(X, y)
15. # Predict the probabilities for different values of hours studied
16. x_new = np.linspace(0, 6, 100).reshape(-1, 1)
17. y_new = model.predict_proba(x_new)[:, 1]
18. # Take the number of hours 3.76 to predict the probability of passing
19. x_fixed = 3.76
20. # Predict the probability of passing for the fixed number of hours
21. y_fixed = model.predict_proba([[x_fixed]])[0, 1]
22. # Print the fixed number of hours and the predicted probability
23. print(f"The fixed number of hours : {x_fixed:.2f}")
24. print(f"The predicted probability of passing : {y_fixed:.2f}")

```

Output:

1. The fixed number of hours : 3.76
2. The predicted probability of passing : 0.81

Tutorial 7.8: To visualize *Tutorial 7.7*, logistic regression model to predict whether a student will pass an exam based on the number of hours they studied in a plot is as follows:

1. `import matplotlib.pyplot as plt`

```

2. # Plot the data and the logistic regression curve
3. plt.scatter(X, y, color="blue", label="Data")
4. plt.plot(x_new, y_new, color="red", label="Logistic regression
   ssion model")
5. plt.xlabel("Hours studied")
6. plt.ylabel("Probability of passing")
7. plt.legend()
8. # Show the figure
9. plt.savefig('student_reasult_prediction_model.jpg',dpi=6
   00,bbox_inches='tight')
10. plt.show()

```

Output:

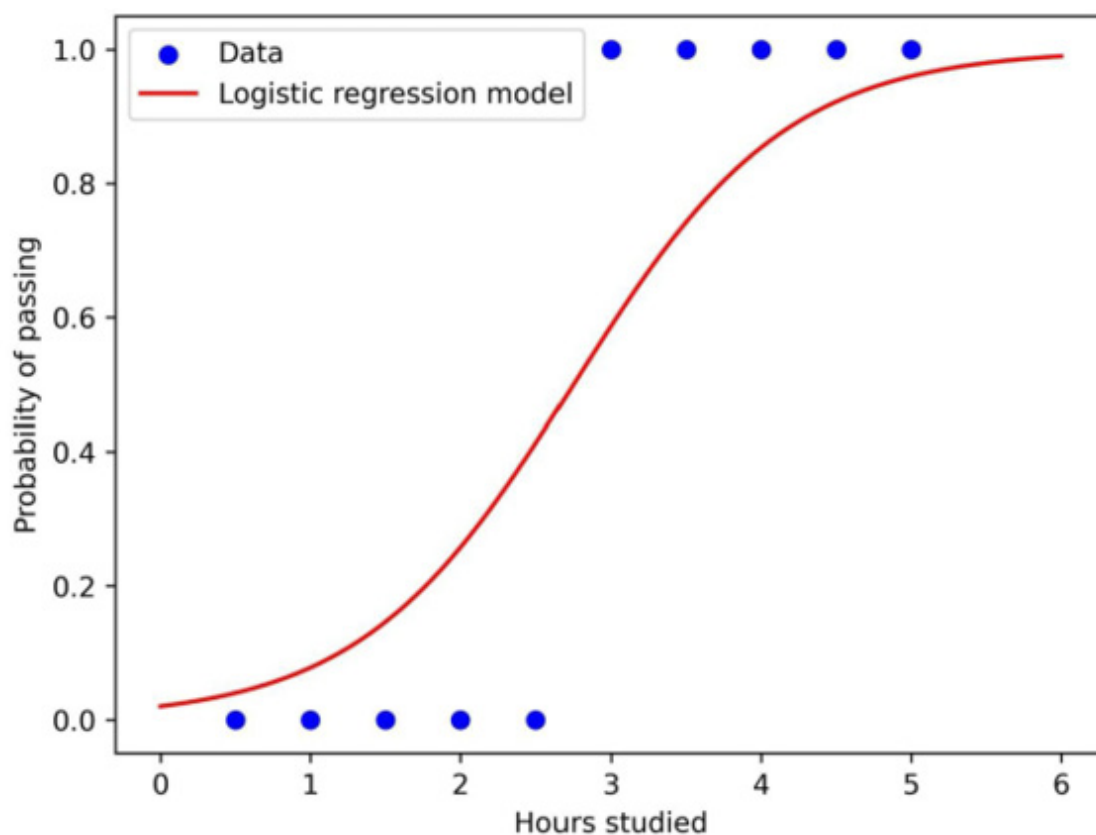


Figure 7.6. Plot of fitted logistic regression model for prediction of student score

Figure 7.6. shows that the probability of passing the final

exam increases as the number of hours studied increases, and that the logistic regression curve captures this trend well.

Fitting models to dependent data

Dependent data refers to related data points, such as repeated measurements on the same subject, clustered measurements from the same group, or spatial measurements from the same location. When fitting models to dependent data, it is important to account for the correlation structure among the data points. This can affect the estimation of the model parameters and the inference of the model effects. For example, fitting models to dependent data is to analyze the blood pressure of patients over time, who are assigned to different treatments. The blood pressure measurements of the same patient are likely to be correlated, and the patients may have different baseline blood pressure levels.

Linear mixed effect model

Linear Mixed-Effects Models (LMMs) are statistical models that can handle dependent data, such as data from longitudinal, multilevel, hierarchical, or correlated studies. They allow for both fixed and random effects. Fixed effects are the effects of variables that are assumed to have a constant effect on the outcome variable, while random effects are the effects of variables that have a varying effect on the outcome variable across groups or individuals. For example, suppose we have a data set of blood pressure measurements from 20 patients who are randomly assigned to one of two treatments: A or B. Blood pressure is measured at four time points: baseline, one month, two months, and three months. We can then fit a linear mixed effects model that predicts blood pressure based on treatment, time, and the interaction between them, while

accounting for correlation within each patient.

Tutorial 7.9: To implement linear mixed effect model to predict blood pressure from 20 patients, as follows:

```
1. import statsmodels.api as sm
2. # Generate some dummy data
3. import numpy as np
4. np.random.seed(50)
5. n_patients = 10 # Number of patients
6. n_obs = 5 # Number of observations per patient
7. x = np.random.randn(n_patients * n_obs) # Covariate
8. patient = np.repeat(np.arange(n_patients), n_obs) # Patient ID
9. bp = 100 + 5 * x + 10 * np.random.randn(n_patients * n_obs) # Blood pressure
10. # Create a data frame
11. import pandas as pd
12. df = pd.DataFrame({"bp": bp, "x": x, "patient": patient})
13. # Fit a linear mixed effect model with a random intercept for each patient
14. model = sm.MixedLM.from_formula("bp ~ x", groups="patient", data=df)
15. result = model.fit()
16. # Print the summary
17. print(result.summary())
```

Here we used **statsmodels** package, which provides a **MixedLM** class for fitting and analyzing mixed effect models.

Output:

```
1.      Mixed Linear Model Regression Results
2. =====
   =====
3. Model:      MixedLM Dependent Variable: bp
```



```

4. No. Observations: 50      Method:          REML
5. No. Groups:      10      Scale:           132.8671
6. Min. group size: 5       Log-Likelihood:  -189.7517
7. Max. group size: 5       Converged:       Yes
8. Mean group size: 5.0
9. -----
10.          Coef. Std.Err.  z    P>|z| [0.025  0.975]
11. -----
12. Intercept  99.960    1.711 58.427 0.000 96.607 103.314
13. x          4.021    1.686  2.384 0.017  0.716  7.326
14. patient Var 2.450    1.345
15. =====
    =====

```

Output shows a linear mixed effect model with a random intercept for each patient, using total 50 observations from 10 patients. The model estimates a fixed intercept of 99.960, a fixed slope of 4.021, and a random intercept variance of 2.450 for each patient. The p-value for the slope is 0.017, which means that it is statistically significant at the 5% level. This implies that there is a positive linear relationship between the covariate x and the blood pressure bp, after accounting for the patient-level variability.

Similarly for fitting dependent data machine learning algorithms like logistic mixed-effects, K-nearest neighbors, multilevel logistic regression, marginal logistic regression, marginal linear regression can also be used.

Decision tree

Decision tree is a way of making decisions based on some data, they are used for both classification and regression problems. It looks like a tree with branches and leaves. Each branch represents a choice or a condition, and each leaf represents an outcome or a result. For example,

suppose you want to decide whether to play tennis or not based on the weather, if the weather is nice and sunny, you want to play tennis, if not, you do not want to play tennis. The decision tree works by starting with the root node, which is the top node. The root node asks a question about the data, such as **Is it sunny?** If the answer is yes, follow the branch to the right. If the answer is no, you follow the branch to the left. You keep doing this until you reach a leaf node that tells you the final decision, such as **Play tennis** or **Do not play tennis**.

Before moving to the tutorials let us look at the syntax for implementing decision tree with **sklearn**, which is as follows:

```
1. # Import decision tree
2. from sklearn.tree import DecisionTreeClassifier
3. # Create a decision tree classifier
4. tree = DecisionTreeClassifier()
5. # Train the classifier
6. tree.fit(X_train, y_train)
```

Tutorial 7.10: To implement a decision tree algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high is as follows:

```
1. import pandas as pd
2. from sklearn.tree import DecisionTreeClassifier
3. # Read the data
4. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
5. # Separate the features and the target
6. X = data.drop("blood_pressure", axis=1)
7. y = data["blood_pressure"]
8. # Encode the categorical features
9. X["gender"] = X["gender"].map({"M": 0, "F": 1})
```

10. *# Build and train the decision tree*

11. `tree = DecisionTreeClassifier()`

12. `tree.fit(X, y)`

Tutorial 7.11: To view graphical representation of the above fitted decision tree (*Tutorial 7.10*), showing the features, thresholds, impurity, and class labels at each node, is as follows:

1. `import matplotlib.pyplot as plt`

2. *# Import the plot_tree function from the sklearn.tree module*

3. `from sklearn.tree import plot_tree`

4. *# Plot the decision tree*

5. `plt.figure(figsize=(10, 8))`

6. *# Fill the nodes with colors, round the corners, and add feature and class names*

7. `plot_tree(tree, filled=True, rounded=True, feature_names=X.columns, class_names=["Low", "Normal", "High"], fontsize=12)`

8. *# Show the figure*

9. `plt.savefig('decision_tree.jpg', dpi=600, bbox_inches='tight')`

10. `plt.show()`

Output:

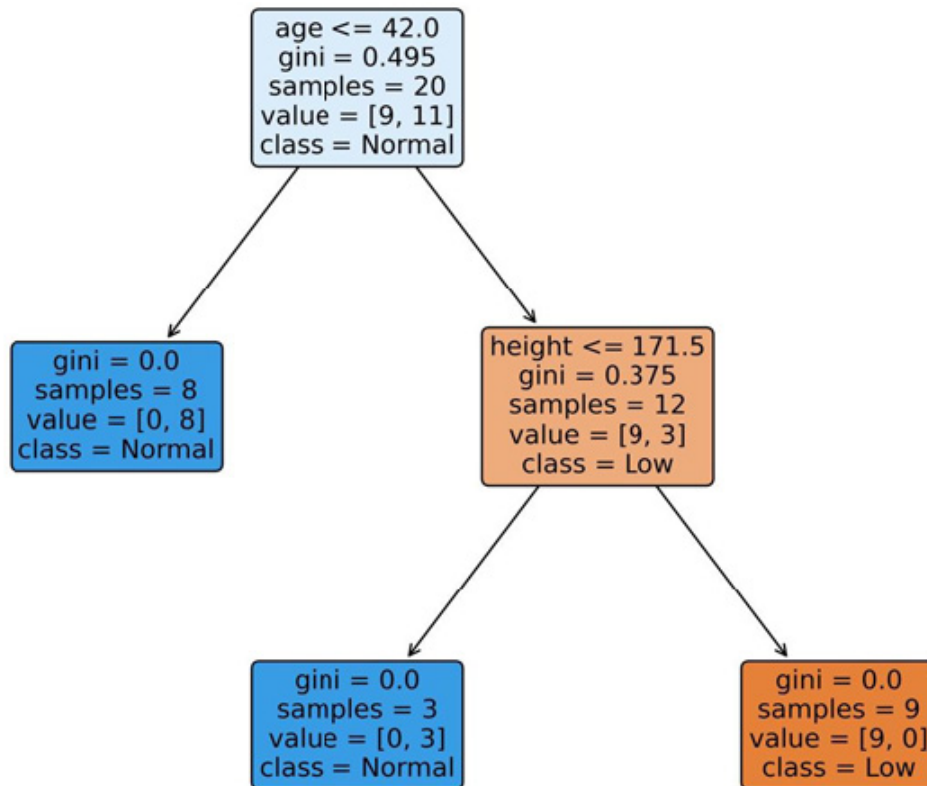


Figure 7.7: Fitted decision tree plot with features, thresholds, impurity, and class labels at each node

It is often a better idea to separate dependent and independent variables and split the dataset into train and test split before fitting the model. Independent data are the features or variables that are used as input to the model, and dependent data are the target or outcome that is predicted by the model. Splitting data into train test split is important because it allows us to evaluate the performance of the model on unseen data and avoid overfitting or underfitting. From the split, train set is used to fit or train the model and test set is used for evaluation of the model.

Tutorial 7.12: To implement decision tree by including the separation of dependent and independent variables, train test split and then fitting data on train set, based on *Tutorial 7.10* is as follows:

1. `import pandas as pd`
2. `from sklearn.tree import DecisionTreeClassifier`

```
3. from sklearn.model_selection import train_test_split
4. # Import the accuracy_score function
5. from sklearn.metrics import accuracy_score
6. # Read the data
7. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
8. # Separate the features and the target
9. X = data.drop("blood_pressure", axis=1) # independent variables
10. y = data["blood_pressure"] # dependent variable
11. # Encode the categorical features
12. X["gender"] = X["gender"].map({"M": 0, "F": 1})
13. # Split the data into training and test sets
14. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
15. # Build and train the decision tree on the training set
16. tree = DecisionTreeClassifier()
17. tree.fit(X_train, y_train)
18. # Further test set can be used to evaluate the model
19. # Predict the values for the test set
20. y_pred = tree.predict(X_test) # Get the predicted values for the test data
21. # Calculate the accuracy score on the test set
22. accuracy = accuracy_score(y_test, y_pred) # Compare the predicted values with the actual values
23. # Print the accuracy score
24. print("Accuracy of the decision tree model on the test set:", accuracy)
```

After fitting the model on the training set, to use the remaining test set for evaluation of fitted model you need to import the **accuracy_score()** from the **sklearn.metrics**

module. Then use the **predict()** of the model on the test set to get the predicted values for the test data. Compare the predicted values with the actual values in the test set using the **accuracy_score()**, which returns a fraction of correct predictions. Finally print the accuracy score to see how well the model performs on the test data. More of this is discussed in the *Model selection and evaluation*.

Output:

1. Accuracy of the decision tree model on the test set : 1.0

This accuracy is quite high because we only have 20 data points in this dataset. Once we have adequate data, the above script will present more realistic results.

Random forest

Random forest is an ensemble learning method that combines multiple decision trees to make predictions. It is highly accurate and robust, making it a popular choice for a variety of tasks, including classification and regression, and other tasks that work by constructing a large number of decision trees at training time. Random forest works by building individual trees and then averaging the predictions of all the trees. To prevent overfitting, each tree is trained on a random subset of the training data and uses a random subset of the features. The random forest predicts by averaging the predictions of all the trees after building them. Averaging reduces prediction variance and improves accuracy.

For example, you have a large dataset of student data, including information about their grades, attendance, and extracurricular activities. As a teacher, you can use random forest to predict which students are most likely to pass their exams. To build a model, you would train a group of decision trees on different subsets of your data. Each tree would use a random subset of the features to make its predictions. After training all of the trees, you would

average their predictions to get your final result. This is like having a group of experts who each look at different pieces of information about your students. Each expert is like a decision tree, and they all make predictions about whether each student will pass or fail. After all the experts have made their predictions, you take an average of all the expert answers to give you the most likely prediction for each student.

Before moving to the tutorials let us look at the syntax for implementing random forest classifier with sklearn, which is as follows:

```
1. # Import RandomForestClassifier
2. from sklearn.ensemble import RandomForestClassifier
3. # Create a Random Forest classifier
4. rf = RandomForestClassifier()
5. # Train the classifier
6. rf.fit(X_train, y_train)
```

Tutorial 7.13. To implement a random forest algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high is as follows:

```
1. import pandas as pd
2. from sklearn.ensemble import RandomForestClassifier
3. # Read the data
4. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
5. # Separate the features and the target
6. X = data.drop("blood_pressure", axis=1) # independent variables
7. y = data["blood_pressure"] # dependent variable
8. # Encode the categorical features
9. X["gender"] = X["gender"].map({"M": 0, "F": 1})
10. # Split the data into training and test sets
11. X_train, X_test, y_train, y_test = train_test_split(X, y, test
```

```
    _size=0.2, random_state=42)
12. # Create a Random Forest classifier
13. rf = RandomForestClassifier()
14. # Train the classifier
15. rf.fit(X_train, y_train)
```

Tutorial 7.14: To evaluate *Tutorial 7.13*, fitted random forest classifier on the test set of data append these lines of code at the end of *Tutorial 7.13*:

```
1. from sklearn.model_selection import train_test_split
2. from sklearn.metrics import accuracy_score
3. # Further test set can be used to evaluate the model
4. # Predict the values for the test set
5. y_pred = tree.predict(X_test) # Get the predicted values
   for the test data
6. # Calculate the accuracy score on the test set
7. accuracy = accuracy_score(y_test, y_pred) # Compare t
   he predicted values with the actual values
8. # Print the accuracy score
9. print("Accuracy of the Random Forest classifier model o
   n the test set :", accuracy)
```

Support vector machine

Support Vector Machines (SVMs) are a type of supervised machine learning algorithm used for classification and regression tasks. They find a hyperplane that separates data points of different classes, maximizing the margin between them. SVMs map data points into a high-dimensional space to make separation easier. A kernel function is used to map data points and measure their similarity in high-dimensional space. SVMs then find the hyperplane that maximizes the margin between the two classes. SVMs are versatile. They can be used for classification, regression, and anomaly detection. They are

particularly well-suited for tasks where the data is nonlinear or has high dimensionality. They are also quite resilient to noise and outliers.

For example, imagine you are a doctor trying to diagnose a patient with a certain disease. patient records that include information about their symptoms, medical history, and blood test results. To predict whether a new patient has the disease or not, you can use SVM to build a model. First, train the SVM on the dataset of patient records. SVM would identify the most important features of the data to distinguish between patients with and without the disease. Then, it could predict whether a new patient has the disease based on their symptoms, medical history, and blood test results.

Before moving to the tutorials let us look at the syntax for implementing support vector classifier from SVM with sklearn, which is as follows:

```
1. # Import Support vector classifier from SVM  
2. from sklearn.svm import SVC  
3. # Create a Support Vector Classifier object  
4. svm = SVC()  
5. # Train the classifier  
6. svm.fit(X_train, y_train)
```

Tutorial 7.15. To implement SVM, support vector classifier algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high and evaluate the result is as follows:

```
1. import pandas as pd  
2. # Import the SVC class  
3. from sklearn.svm import SVC  
4. from sklearn.model_selection import train_test_split  
5. from sklearn.metrics import accuracy_score  
6. # Read the data
```

```

7. data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
8. # Separate the features and the target
9. X = data.drop("blood_pressure", axis=1) # independent variables
10. y = data["blood_pressure"] # dependent variable
11. # Encode the categorical features
12. X["gender"] = X["gender"].map({"M": 0, "F": 1})
13. # Split the data into training and test sets
14. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
15. # Create an SVM classifier
16. svm = SVC(kernel="rbf", C=1, gamma=0.1) # You can change the parameters as you wish
17. # Train the classifier
18. svm.fit(X_train, y_train)
19. # Predict the values for the test set
20. y_pred = svm.predict(X_test) # Get the predicted values for the test data
21. # Calculate the accuracy score on the test set
22. accuracy = accuracy_score(y_test, y_pred) # Compare the predicted values with the actual values
23. # Print the accuracy score
24. print("Accuracy of the SVM classifier model on the test set :", accuracy)

```

The output of this will be a trained model and accuracy score of classifiers.

K-nearest neighbor

K-Nearest Neighbor (KNN) is a machine learning algorithm used for classification and regression. It finds the k nearest neighbors of a new data point in the training data

and uses the majority class of those neighbors to classify the new data point. KNN is useful when the data is not linearly separable, meaning that there is no clear boundary between different classes or outcomes. KNN is useful when dealing with data that has many features or dimensions because it makes no assumptions about the distribution or structure of the data. However, it can be slow and memory-intensive since it must store and compare all the training data for each prediction.

A simpler example to explain it is, suppose you want to predict the color of a shirt based on its size and price. The training data consists of ten shirts, each labeled as either red or blue. To classify a new shirt, we need to find the k closest shirts in the training data, where k is a number chosen by us. For example, if $k = 3$, we look for the 3 nearest shirts based on the difference between their size and price. Then, we count how many shirts of each color are among the 3 nearest neighbors, and assign the most frequent color to the new shirt. For example, if 2 of the 3 nearest neighbors are red, and 1 is blue, we predict that the new shirt is red.

Let us see a tutorial to predict the type of flower based on its features, such as petal length, petal width, sepal length, and sepal width. The training data consists of 150 flowers, each labeled as one of three types: Iris setosa, Iris versicolor, or Iris virginica. The number of k is chosen by us. For instance, if $k = 5$, we look for the 5 nearest flowers based on the Euclidean distance between their features. We count the number of flowers of each type among the 5 nearest neighbors and assign the most frequent type to the new flower. For instance, if 3 out of the 5 nearest neighbors are Iris versicolor and 2 are Iris virginica, we predict that the new flower is Iris versicolor.

Tutorial 7.16: To implement KNN on iris dataset to predict the type of flower based on its features, such as petal

length, petal width, sepal length, and sepal width and also evaluate the result, is as follows:

```
1. # Load the Iris dataset
2. from sklearn.datasets import load_iris
3. # Import the KNeighborsClassifier class
4. from sklearn.neighbors import KNeighborsClassifier
5. # Import train_test_split for data splitting
6. from sklearn.model_selection import train_test_split
7. # Import accuracy_score for evaluating model performance
8. from sklearn.metrics import accuracy_score
9. # Load the Iris dataset
10. iris = load_iris()
11. # Separate the features and the target variable
12. X = iris.data # Features (sepal length, sepal width, petal length, petal width)
13. y = iris.target # Target variable (species: Iris-setosa, Iris-versicolor, Iris-virginica)
14. # Encode categorical features (if any)
15. # No categorical features in the Iris dataset
16. # Split the data into training (90%) and test sets (10%)
17. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
18. # Create a KNeighborsClassifier object
19. knn = KNeighborsClassifier(n_neighbors=5) # Set number of neighbors to 5
20. # Train the classifier
21. knn.fit(X_train, y_train)
22. # Make predictions on the test data
23. y_pred = knn.predict(X_test)
24. # Evaluate the model's performance using accuracy
```

```
25. accuracy = accuracy_score(y_test, y_pred)
26. # Print the accuracy score
27. print("Accuracy of the KNN classifier on the test set :", accuracy)
```

Output:

```
1. Accuracy of the KNN classifier on the test set : 1.0
```

Model selection and evaluation

Model selection and evaluation methods are techniques used to measure the performance and quality of machine learning models. Supervised learning methods commonly use evaluation metrics such as accuracy, precision, recall, F1-score, mean squared error, mean absolute error, and area under the curve. Unsupervised learning methods commonly use evaluation metrics such as silhouette score, Davies-Bouldin index, Calinski-Harabasz index, and adjusted Rand index.

Evaluation metrics and model selection for supervised

As mentioned above and to summarize choosing a single candidate machine learning model for a predictive modeling challenge is known as model selection. Performance, complexity, interpretability, and resource requirements are some examples of selection criteria. As mentioned, accuracy, precision, recall, F1-score, and area under the curve are highly relevant to evaluate classifier result. **Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-Squared (R²)** are useful for evaluating prediction models. *Tutorial 7.30* demonstrates for each section, shows how to use some common model selection and evaluation techniques for supervised learning using the scikit-learn library.

Tutorial 7.30: To implement a tutorial that illustrates

model selection and evaluation in supervised machine learning using iris data, is as follows:

To begin, we need to import modules and load the iris dataset. This dataset contains 150 samples of three different types of iris flowers, each with four features: sepal length, sepal width, petal length, and petal width. Our goal is to construct a classifier that can predict the species of a new flower based on its features as follows:

```
1. import numpy as np # For numerical operations
2. import pandas as pd # For data manipulation and analysis
3. import matplotlib.pyplot as plt # For data visualization
4. from sklearn.datasets import load_iris # For loading the iris dataset
5. from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV, RandomizedSearchCV # For splitting data, cross-validation, and hyperparameter tuning
6. from sklearn.linear_model import LogisticRegression # For logistic regression model
7. from sklearn.tree import DecisionTreeClassifier # For decision tree model
8. from sklearn.svm import SVC # For support vector machine model
9. from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report # For evaluating model performance
10. # Load dataset
11. iris = load_iris()
12. # Extract the features & labels as a numpy array
13. X = iris.data
14. y = iris.target
```

```
15. print(iris.feature_names)
16. print(iris.target_names)
```

Output:

1. ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
2. ['setosa' 'versicolor' 'virginica']

Continuing the *Tutorial 7.30* we will now split the data set into training and test sets. The data will be divided into 70% for training and 30% for testing. Additionally, we will set a random seed for reproducibility as follows:

1. *# Split dataset*
2. `X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)`
3. `print(X_train.shape, y_train.shape)`
4. `print(X_test.shape, y_test.shape)`

Output:

1. (105, 4) (105,)
2. (45, 4) (45,)

Now, we will define candidate models in *Tutorial 7.30* to compare, including logistic regression, decision tree, and support vector machine classifiers. Candidate model refers to a machine learning algorithm that is being considered or tested to solve a particular problem. We will use default values for hyperparameters. However, they can be adjusted to find the optimal solution as follows:

1. *# Define candidate models*
2. `models = {`
3. `'Logistic Regression': LogisticRegression(),`
4. `'Decision Tree': DecisionTreeClassifier(),`
5. `'Support Vector Machine': SVC()`
6. `}`

To evaluate the performance of each model in *Tutorial 7.30*, we can use cross-validation. This technique involves

splitting the training data into k folds. One-fold is used as the validation set, and the rest is used as the training set. This process is repeated **k** times, and the average score across the k folds is reported. Cross-validation helps to reduce the variance of the estimate and avoid overfitting. In this case, we will use 5-fold cross-validation and accuracy as the scoring metric as follows:

```
1. # Evaluate models using cross-validation
2. scores = {}
3. for name, model in models.items():
4.     score = cross_val_score(model, X_train, y_train, cv=5,
5.                             scoring='accuracy')
6.     scores[name] = np.mean(score)
7.     print(f'{name}: {np.mean(score):.3f} (+/- {np.std(score):.3f})')
```

Output:

```
1. Logistic Regression: 0.962 (+/- 0.036)
2. Decision Tree: 0.933 (+/- 0.023)
3. Support Vector Machine: 0.952 (+/- 0.043)
```

The logistic regression and support vector machine models have comparatively almost similar and high accuracy scores. However, the decision tree model has a slightly lower score, indicating overfitting and poor generalization. To better compare the results, a bar chart can be plotted as follows:

```
1. # Plot scores
2. plt.bar(scores.keys(), scores.values())
3. plt.ylabel('Accuracy')
4. plt.show()
```

Output:

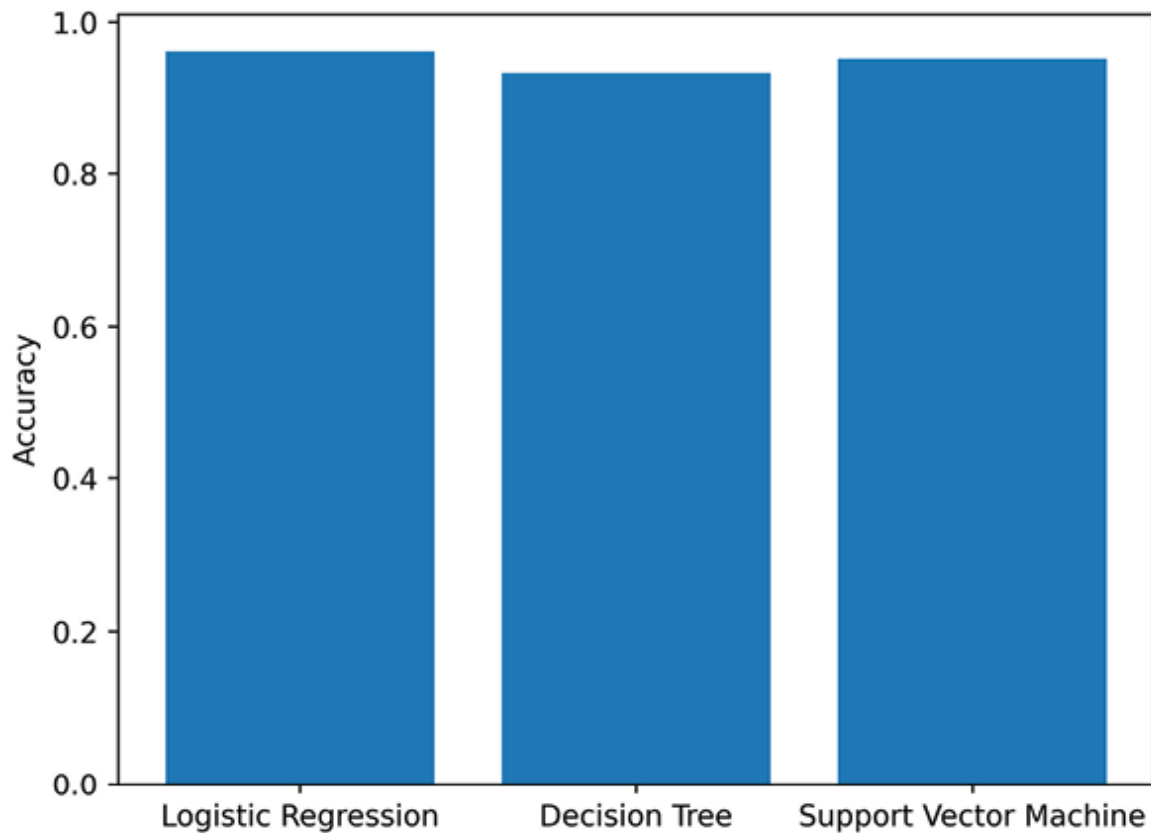


Figure 7.8: Accuracy comparison of supervised algorithms on the Iris dataset

As evaluation measures how well, the model performs on unseen data, such as a test set, by comparing its predictions to the actual results using various metrics. Now we evaluate each model's performance, use the testing set. Fit each model on the training set, make predictions on the testing set, and compare them with the true labels. Compute metrics such as accuracy, confusion matrix, and classification report. The confusion matrix shows the number of accurate and inaccurate predictions for each class, while the classification report presents the precision, recall, f1-score, and support for each class as follows:

1. # Evaluate models using testing set
2. `for name, model in models.items():`
3. `model.fit(X_train, y_train)` # Fit model on training set
4. `y_pred = model.predict(X_test)` # Predict on testing set

```

t
5. acc = accuracy_score(y_test, y_pred) # Compute accuracy
6. cm = confusion_matrix(y_test, y_pred) # Compute confusion matrix
7. prec = precision_score(y_test, y_pred, average='weighted') # Compute precision
8. recall = recall_score(y_test, y_pred, average='weighted') # Compute recall
9. f1score = f1_score(y_test, y_pred, average='weighted') # Compute f1score
10. print(f'\n{name}')
11. print(f'Accuracy: {acc:.3f}')
12. print(f'Precision: {prec:.3f}')
13. print(f'Recall: {recall:.3f}')
14. print(f'Confusion matrix:\n{cm}')

```

Output:

```

1. Logistic Regression
2. Accuracy: 1.000
3. Precision: 1.000
4. Recall: 1.000
5. Confusion matrix:
6. [[19 0 0]
7. [ 0 13 0]
8. [ 0 0 13]]
9. Decision Tree
10. Accuracy: 1.000
11. Precision: 1.000
12. Recall: 1.000
13. Confusion matrix:
14. [[19 0 0]
15. [ 0 13 0]
16. [ 0 0 13]]

```

17. Support Vector Machine

18. Accuracy: 1.000

19. Precision: 1.000

20. Recall: 1.000

21. Confusion matrix:

22. $\begin{bmatrix} 19 & 0 & 0 \end{bmatrix}$

23. $\begin{bmatrix} 0 & 13 & 0 \end{bmatrix}$

24. $\begin{bmatrix} 0 & 0 & 13 \end{bmatrix}$

The logistic regression, decision tree, and support vector machine models all have the highest accuracy, precision, recall, and f1 score of 1.0. All of these and the confusion matrix indicate that all models have perfect predictions for all classes. Therefore, all models are equally effective for this classification problem. However, it is important to consider factors other than performance, such as complexity, interpretability, and resource requirements. For example, the logistic regression model is the simplest and most interpretable model. On the other hand, the support vector machine model and the decision tree model are the most complex and least interpretable models. The resource requirements for each model depend on the size and dimensionality of the data, the number and range of hyperparameters, and the available computing power. Therefore, the selection of the final model depends on the trade-off between these factors.

Semi-supervised and self-supervised learnings

Semi-supervised learning is a paradigm that combines both labeled and unlabeled data for training machine learning models. In this approach, we have a limited amount of labeled data (with ground truth labels) and a larger pool of unlabeled data. The goal is to leverage the unlabeled data to improve model performance. It bridges the gap between fully supervised (only labeled data) and unsupervised (no

labels) learning. Imagine you're building a spam email classifier. You have a small labeled dataset of spam and non-spam emails, but a vast number of unlabeled emails. By using semi-supervised learning, you can utilize the unlabeled emails to enhance the classifier's accuracy.

Self-supervised learning is a type of unsupervised learning where the model generates its own labels from the input data. Instead of relying on external annotations, the model creates its own supervision signal. Common self-supervised tasks include predicting missing parts of an input (e.g., masked language models) or learning representations by solving pretext tasks (e.g., word embeddings). Consider training a neural network to predict the missing word in a sentence. Given the sentence: **The cat chased the blank**, the model learns to predict the missing word **mouse**. Here, the model generates its own supervision by creating a masked input. Thus, the key difference lies in semi-supervised and self-supervised is the source of supervision.

- **Semi-supervised** uses a small amount of labeled data and a larger pool of unlabeled data.
 - **Use case:** When labeled data is scarce or expensive to obtain.
 - **Example:** Pretraining language models like BERT on large text corpora without explicit labels.
- **Self-supervised learning** creates its own supervision signal from the input data.
 - **Use case:** When you have some labeled data but want to leverage additional unlabeled data.
 - **Example:** Enhancing image classification models by incorporating unlabeled images alongside labeled ones.

Semi-supervised techniques

Semi-supervised learning bridges the gap between fully supervised and unsupervised learning. It leverages both labeled and unlabeled data to improve model performance. Semi-supervised techniques allow us to make the most of limited labeled data by incorporating unlabeled examples. By combining these methods, we achieve better generalization and performance in real-world scenarios. In this chapter, we explore three essential semi-supervised techniques which are **self-training**, **co-training**, and **graph-based methods**, each with a specific task or idea, along with examples to address or solve them.

- **Self-training:** Self-training is a simple yet effective approach. It starts with an initial model trained on the limited labeled data available. The model then predicts labels for the unlabeled data, and confident predictions are added to the training set as pseudo-labeled examples. The model is retrained using this augmented dataset, iteratively improving its performance. Suppose we have a sentiment analysis task with a small labeled dataset of movie reviews. We train an initial model on this data. Next, we apply the model to unlabeled reviews, predict their sentiments, and add the confident predictions to the training set. The model is retrained, and this process continues until convergence.
 - **Idea:** Iteratively label unlabeled data using model predictions.
 - **Example:** Train a classifier on labeled data, predict labels for unlabeled data, and add confident predictions to the labeled dataset.

Tutorial 7.32: To implement self-training classifier on Iris dataset, as follows:

1. `from sklearn.semi_supervised import SelfTrainingClassifier`
2. `from sklearn.datasets import load_iris`

```

3. from sklearn.model_selection import train_test_split
4. from sklearn.linear_model import LogisticRegression
5. # Load the Iris dataset (labeled data)
6. X, y = load_iris(return_X_y=True)
7. # Split data into labeled and unlabeled portions
8. X_labeled, X_unlabeled, y_labeled, y_unlabeled = train_test_split(X, y, test_size=0.8, random_state=42)
9. # Initialize a base classifier (e.g., logistic regression)
10. base_classifier = LogisticRegression()
11. # Create a self-training classifier
12. self_training_clf = SelfTrainingClassifier(base_classifier)
13. # Fit the model using labeled data
14. self_training_clf.fit(X_labeled, y_labeled)
15. # Predict on unlabeled data
16. y_pred_unlabeled = self_training_clf.predict(X_unlabeled)
17. # Print the original labels for the unlabeled data
18. print("Original labels for unlabeled data:")
19. print(y_unlabeled)
20. # Print the predictions
21. print("Predictions on unlabeled data:")
22. print(y_pred_unlabeled)

```

Output:

```

1. Original labels for unlabeled data:
2. [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0
   0 0 1 0 0 2 1
3. 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
   2 0 1 2 0 2 2
4. 1 1 2 1 0 1 2 0 0 1 1 0 2 0 0 1 1 2 1 2 2 1 0 0 2 2 0 0 0 1
   2 0 2 2 0 1 1
5. 2 1 2 0 2 1 2 1 1]
6. Predictions on unlabeled data:
7. [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0

```

```

0 0 1 0 0 2 1
8. 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
   2 0 1 2 0 2 2
9. 1 1 2 1 0 1 2 0 0 1 2 0 2 0 0 2 1 2 2 2 2 1 0 0 1 2 0 0 0 1
   2 0 2 2 0 1 1
10. 2 1 2 0 2 1 2 1 1]

```

The above outputs have few wrong predictions. Now, let us see the evaluation metrics.

Tutorial 7.33: To evaluate the trained self-training classifier performance using appropriate metrics (e.g., accuracy, F1-score, etc.), as follows:

```

1. from sklearn.metrics import accuracy_score, f1_score, pr
   ecision_score, recall_score
2. # Assuming y_unlabeled_true contains true labels for un
   labeled data
3. accuracy = accuracy_score(y_unlabeled, y_pred_unlabel
   ed)
4. f1 = f1_score(y_unlabeled, y_pred_unlabeled, average='
   weighted')
5. precision = precision_score(y_unlabeled, y_pred_unlabel
   ed, average='weighted')
6. recall = recall_score(y_unlabeled, y_pred_unlabeled, ave
   rage='weighted')
7. print(f"Accuracy: {accuracy:.2f}")
8. print(f"F1-score: {f1:.2f}")
9. print(f"Precision: {precision:.2f}")
10. print(f"Recall: {recall:.2f}")

```

Output:

```

1. Accuracy: 0.97
2. F1-score: 0.97
3. Precision: 0.97
4. Recall: 0.97

```

Here, we see an accuracy of 0.97 means that approximately

97% of the predictions were correct. F1-score of 0.97 suggests a good balance between precision and recall, where higher values indicate better performance. A precision of 0.97 means that 97% of the positive predictions were accurate. A recall of 0.97 indicates that 97% of the positive instances were correctly identified. Further calibration of the classifier is essential for better results. You can fine-tune hyperparameters or use techniques like Platt scaling or isotonic regression to improve calibration.

- **Co-training:** Co-training leverages multiple views of the data. It assumes that different features or representations can provide complementary information. Two or more classifiers are trained independently on different subsets of features or views. During training, they exchange their confident predictions on unlabeled data, reinforcing each other's learning. Consider a text classification problem where we have both textual content and associated metadata, for example, author, genre. We train one classifier on the text and another on the metadata. They exchange predictions on unlabeled data, improving their performance collectively.
 - **Idea:** Train multiple models on different views of data and combine their predictions.
 - **Example:** Train one model on text features and another on image features, then combine their predictions for a joint task.

Tutorial 7.34: To show an easy implementation of co-training with two views of data, on **UCImultifeature** dataset from **mvlearn.datasets**, as follows:

1. `from mvlearn.semi_supervised import CTClassifier`
2. `from mvlearn.datasets import load_UCImultifeature`
3. `from sklearn.linear_model import LogisticRegression`
4. `from sklearn.ensemble import RandomForestClassifier`
5. `from sklearn.model_selection import train_test_split`


```

6. data, labels = load_UCImultifeature(select_labeled=
    [0,1])
7. X1 = data[0] # Text view
8. X2 = data[1] # Metadata view
9. X1_train, X1_test, X2_train, X2_test, l_train, l_test = train_test_split(X1, X2, labels)
10. # Co-
    training with two views of data and 2 estimator types
11. estimator1 = LogisticRegression()
12. estimator2 = RandomForestClassifier()
13. ctc = CTClassifier(estimator1, estimator2, random_state
    =1)
14. # Use different matrices for each view
15. ctc = ctc.fit([X1_train, X2_train], l_train)
16. preds = ctc.predict([X1_test, X2_test])
17. print("Accuracy: ", sum(preds==l_test) / len(preds))

```

This code snippet illustrates the application of co-training, a semi-supervised learning technique, using the **CTClassifier** from **mvlearn.semi_supervised**. Initially, a multi-view dataset is loaded, focusing on two specified classes. The dataset is divided into two views: text and metadata. Following this, the data is split into training and testing sets. Two distinct classifiers, logistic regression and random forest, are instantiated. These classifiers are then incorporated into the **CTClassifier**. After training on the training data from both views, the model predicts labels for the test data. Finally, the accuracy of the co-training model on the test data is computed and displayed. Output will display the accuracy of the model as follows:

- **Graph-based methods:** Graph-based methods exploit the inherent structure in the data. They construct a graph where nodes represent instances (labeled and unlabeled), and edges encode similarity or relationships. Label propagation or graph-based regularization is then

used to propagate labels across the graph, benefiting from both labeled and unlabeled data. In a recommendation system, users and items can be represented as nodes in a graph. Labeled interactions (e.g., user-item ratings) provide initial labels. Unlabeled interactions contribute to label propagation, enhancing recommendations as follows:

- **Idea:** Leverage data connectivity (e.g., graph Laplacians) for label propagation.
- **Example:** Construct a graph where nodes represent data points, and edges represent similarity. Propagate labels across the graph.

Self-supervised techniques

Self-supervised learning techniques empower models to learn from unlabeled data, reducing the reliance on expensive labeled datasets. These methods exploit inherent structures within the data itself to create meaningful training signals. In this chapter, we delve into three essential self-supervised techniques: **word embeddings**, **masked language models**, and **language models**.

- **Word embeddings:** A word embedding is a representation of a word as a real-valued vector. These vectors encode semantic meaning, allowing similar words to be close in vector space. Word embeddings are crucial for various **Natural Language Processing (NLP)** tasks. They can be obtained using techniques like neural networks, dimensionality reduction, and probabilistic models. For instance, **Word2Vec** and **GloVe** are popular methods for generating word embeddings. Let us consider an example, suppose we have a corpus of text. Word embeddings capture relationships between words. For

instance, the vectors for king and queen should be similar because they share a semantic relationship.

- **Idea:** Pretrained word representations.
- **Use:** Initializing downstream models, for example natural language processing tasks.

Tutorial 7.35: To implement word embeddings using self-supervised task using **Word2Vec** method, as follows:

```
1. # Install Gensim and import word2vec for word embeddings
2. import gensim
3. from gensim.models import Word2Vec
4. # Example sentences
5. sentences = [
6.     ["I", "love", "deep", "learning"],
7.     ["deep", "learning", "is", "fun"],
8.     ["machine", "learning", "is", "easy"],
9.     ["deep", "learning", "is", "hard"],
10.    # Add more sentences, embedding changes with new words...
11. ]
12. # Train Word2Vec model
13. model = Word2Vec(sentences, vector_size=10, window=5, min_count=1, sg=1)
14. # Get word embeddings
15. word_vectors = model.wv
16. # Example: Get embedding for the each word in sentence "I love deep learning"
17. print("Embedding for 'I':", word_vectors["I"])
18. print("Embedding for 'love':", word_vectors["love"])
19. print("Embedding for 'deep':", word_vectors["deep"])
20. print("Embedding for 'learning':", word_vectors["learning"])
```

Output:

1. Embedding for 'I': [-0.00856557 0.02826563 0.05401429
0.07052656 -0.05703121 0.0185882
0.06088864 -0.04798051 -0.03107261 0.0679763]
3. Embedding for 'love': [0.05455794 0.08345953 -0.01453741
-0.09208143 0.04370552 0.00571785
0.07441908 -0.00813283 -0.02638414 -0.08753009]
5. Embedding for 'deep': [0.07311766 0.05070262 0.06757693
0.00762866 0.06350891 -0.03405366
-0.00946401 0.05768573 -0.07521638 -0.03936104]
7. Embedding for 'learning': [-0.00536227 0.00236431
0.0510335 0.09009273 -0.0930295 -0.07116809
0.06458873 0.08972988 -0.05015428 -0.03763372]
- 8.

- **Masked Language Models (MLM):** MLM is a powerful self-supervised technique used by models like **Bidirectional Encoder Representations from Transformers (BERT)**. In MLM, some tokens in an input sequence are masked, and the model learns to predict these masked tokens based on context. It considers both preceding and following tokens, making it bidirectional. Given the sentence: **The cat sat on the [MASK]**. The model predicts the masked token, which could be **mat**, **chair**, or any other valid word based on context as follows:
 - **Idea:** Unidirectional pretrained language representations.
 - **Use:** Full downstream model initialization for various language understanding tasks.
- **Language models:** A language model is a probabilistic model of natural language. It estimates the likelihood of a sequence of words. Large language models, such as GPT-4 and ELMo, combine neural

networks and transformers. They have superseded earlier models like n-gram language models. These models are useful for various NLP tasks, including speech recognition, machine translation, and information retrieval. Imagine a language model trained on a large corpus of text. Given a partial sentence, it predicts the most likely next word. For instance, if the input is **The sun is shining**, the model might predict **brightly** as follows:

- **Idea:** Bidirectional pretrained language representations.
- **Use:** Full downstream model initialization for tasks like text classification and sentiment analysis.

Conclusion

In this chapter, we explored the basics and applications of statistical machine learning. Supervised machine learning is a powerful and versatile tool for data analysis and AI for labeled data. Knowing the type of problem, whether supervised or unsupervised, solves half the learning problems; the next step is to implement different models and algorithms. Once this is done, it is critical to evaluate and compare the performance of different models using techniques such as cross-validation, bias-variance trade-off, and learning curves. Some of the best known and most commonly used supervised machine learning techniques have been demonstrated. These techniques include decision trees, random forests, support vector machines, K-nearest neighbors, linear and logistic regression. We've also talked about semi-supervised and self-supervised, and techniques for implementing them. We have also mentioned the advantages and disadvantages of each approach, as well as some of the difficulties and unanswered questions in the field of machine learning.

Chapter 8, Unsupervised Machine Learning explores the other type of statistical machine learning, unsupervised machine learning.

CHAPTER 8

Unsupervised Machine Learning

Introduction

Unsupervised learning is a key area within statistical machine learning that focuses on uncovering patterns and structures in unlabelled data. This includes techniques like clustering, dimensionality reduction, and generative modelling. Given that most real-world data is unstructured, extensive preprocessing is often required to transform it into a usable format, as discussed in previous chapters. The abundance of unstructured and unlabelled data makes unsupervised learning increasingly valuable. Unlike supervised learning, which relies on labelled examples and predefined target variables, unsupervised learning operates without such guidance. It can group similar items together, much like sorting a collection of coloured marbles into distinct clusters, or reduce complex datasets into simpler forms through dimensionality reduction, all without sacrificing important information. Evaluating the performance and generalization in unsupervised learning also requires different metrics compared to supervised learning.

Structure

In this chapter, we will discuss the following topics:

- Unsupervised learning
- Model selection and evaluation

Objectives

The objective of this chapter is to introduce unsupervised machine learning, ways to evaluate a trained unsupervised model. With real-world examples and tutorials to better explain and demonstrate the implementation.

Unsupervised learning

Unsupervised learning is a machine learning technique where algorithms are trained on unlabeled data without human guidance. The data has no predefined categories or labels and the goal is to discover patterns and hidden structures. Unsupervised learning works by finding similarities or differences in the data and grouping them into clusters or categories. For example, an unsupervised algorithm can analyze a collection of images and sort them by color, shape or size. This is useful when there is a lot of data and labeling them is difficult. For example, imagine you have a bag of 20 candies with various colors and shapes. You wish to categorize them into different groups, but you are unsure of the number of groups or their appearance. Unsupervised learning can help find the optimal way to sort or group items.

Another example is, let us take the iris dataset without flower type labels. Suppose from iris dataset you take a data of 100 flowers with different features, such as petal length, petal width, sepal length and sepal width. You want to group the flowers into different types, but you do not know how many types there are or what they look like. You

can use unsupervised learning to find the optimal number of clusters and assign each flower to one of them. You can use any of unsupervised learning algorithm, for example K-means algorithm for clustering, which is described in the **K-means** section. The algorithm will randomly be choosing K points as the centers of the clusters, and then assigning each flower to the nearest center. Then, it will update the centers by taking the average of the features of the flowers in each cluster. It will repeat this process until the clusters are stable and no more changes occur.

There are many unsupervised learning algorithms some most common ones are described in this chapter. Unsupervised learning models are used for three main tasks: clustering, association, and dimensionality reduction.

Table 8.1 summarizes these tasks:

Algorithm	Task	Description
K-means	Clustering	Divides data into a predefined number of clusters based on similarity.
K-prototype	Clustering	Similar to K-means, but can handle numerical, categorical, and text data.
Hierarchical clustering	Clustering	Creates a hierarchy of clusters by repeatedly merging or splitting groups of data points.
Gaussian mixture models	Clustering	Models data as a mixture of Gaussian distributions, allowing for more flexible clustering.
Principal component analysis	Dimensionality reduction	Finds a lower-dimensional representation of data while preserving as much information as possible.
Singular value decomposition	Dimensionality reduction	Factorizes a data matrix into three matrices, allowing for dimensionality reduction and data visualization.
DBSCAN	Clustering	Finds clusters of overlapping data points based on density.
t-Distributed	Dimensionality	Creates a two- or three-dimensional

Stochastic Neighbor Embedding (t-SNE)	reduction	representation of high-dimensional data while preserving local relationships.
Autoencoders	Dimensionality reduction and dimensionality increase	Learn a compressed representation of data and then reconstruct the original data, allowing for dimensionality reduction or dimensionality increase.
Apriori	Association	Uncovers frequent item sets in transactional datasets.
Eclat	Association	Similar to Apriori, but uses a more efficient algorithm for large datasets.
FP-Growth	Association	A more memory-efficient algorithm for finding frequent item sets.

Table 8.1: *Summary of unsupervised learning algorithms and their tasks*

As described in [Table 8.1](#), the primary applications of unsupervised learning include clustering, dimensionality reduction, and association rule mining. Association rule mining aims to uncover interesting relationships between items in a dataset, similar to identifying patterns in grocery shopping lists. High-dimensional data can be overwhelming, but dimensionality reduction simplifies it while retaining the most important information.

K-means

K-means clustering is an iterative algorithm that divides data points into a predefined number of clusters. It works by first randomly selecting K centroids, one for each cluster. It then assigns each data point to the nearest centroid. The centroids are then updated to be the average of the data points in their respective clusters. This process is repeated until the centroids no longer change. It is used to cluster numerical data. It is often used in marketing to segment customers, in finance to detect fraud and in data mining to discover hidden patterns in data.

For example, K-means can be applied here. Imagine you have a shopping cart dataset of items purchased by customers. You want to group customers into clusters based on the items they tend to buy together.

Before moving to the tutorials let us look at the syntax for implementing K-means with sklearn, which is as follows:

```
1. from sklearn.cluster import KMeans
2. # Load the dataset
3. data = ...
4. # Create and fit the k-
   means model, n_clusters can be any number of clusters
5. kmeans = KMeans(n_clusters=...)
6. kmeans.fit(data)
```

Tutorial 8.1: To implement K-means clustering using **sklearn** on a sample data, is as follows:

```
1. from sklearn.cluster import KMeans
2. # Load the dataset
3. data = [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]]
4. # Create and fit the k-means model
5. kmeans = KMeans(n_clusters=3)
6. kmeans.fit(data)
7. # Predict the cluster labels for each data point
8. labels = kmeans.predict(data)
9. print(f"Clusters labels for data: {labels}")
```

Following is an output which shows the respective cluster label for the above six data:

```
1. Clusters labels for data: [1 1 2 2 0 0]
```

K-prototype

K-prototype clustering is a generalization of K-means clustering that allows for mixed clusters with both numerical and categorical data. It works by first randomly

selecting K centroids, just like K-means. It then assigns each data point to the nearest centroid. The centroids are then updated to be the mean of the data points in their respective clusters. This process is repeated until the centroids no longer change. It is used for clustering data that has both numerical and categorical characteristics. And also, for textual data.

For example, K-prototype can be applied here. Imagine you have a social media dataset of users and their posts. You want to group users into clusters based on both their demographic information (e.g., age, gender) and their posting behavior (e.g., topics discussed, sentiment).

Before moving to the tutorials let us look at the syntax for implementing K-prototype with K modes, which is as follows:

```
1. from kmodes.kprototypes import KPrototypes
2. # Load the dataset
3. data = ...
4. # Create and fit the k-prototypes model
5. kproto = KPrototypes(n_clusters=3, init='Cao')
6. kproto.fit(data, categorical=[0, 1])
```

Tutorial 8.2: To implement K-prototype using **K modes** on a sample data, is as follows:

```
1. import numpy as np
2. from kmodes.kmodes import KModes
3. # Load the dataset
4. data = [[1, 2, 'A'], [2, 3, 'B'], [3, 4, 'A'], [4, 5, 'B'], [5, 6, 'B'], [6, 7, 'A']]
5. # Convert the data to a NumPy array
6. data = np.array(data)
7. # Define the number of clusters
8. num_clusters = 3
```

```
9. # Create and fit the k-prototypes model
10. kprototypes = KModes(n_clusters=num_clusters, init='random')
11. kprototypes.fit(data)
12. # Predict the cluster labels for each data point
13. labels = kprototypes.predict(data)
14. print(f"Clusters labels for data: {labels}")
```

Output:

```
1. Clusters labels for data: [2 0 2 1 0 2]
```

Hierarchical clustering

Hierarchical clustering is an algorithm that creates a tree-like structure of clusters by merging or splitting groups of data points. There are two main types of hierarchical clustering, that is, agglomerative and divisive. Agglomerative hierarchical clustering starts with each data point in its own cluster and then merges clusters until the desired number of clusters is reached. On the other hand, divisive hierarchical clustering starts with all data points in a single cluster and then splits clusters until the desired number of clusters is reached. It is a versatile algorithm. It can cluster any type of data. Often used in social network analysis to identify communities. Additionally, it is used in data mining to discover hierarchical relationships in data.

For example, hierarchical clustering can be applied here. Imagine you have a network of people connected by friendship ties. You want to group people into clusters based on the strength of their ties.

Before moving to the tutorials let us look at the syntax for implementing hierarchical clustering with sklearn, which is as follows:

```
1. from sklearn.cluster import AgglomerativeClustering
2. # Load the dataset
```

```
3. data = ...
4. # Create and fit the hierarchical clustering model
5. hier = AgglomerativeClustering(n_clusters=3)
6. hier.fit(data)
```

Tutorial 8.3: To implement hierarchical clustering using **sklearn** on a sample data, is as follows:

```
1. from sklearn.cluster import AgglomerativeClustering
2. # Load the dataset
3. data = [[1, 1], [1, 2], [2, 2], [2, 3], [3, 3], [3, 4]]
4. # Create and fit the hierarchical clustering model
5. cluster = AgglomerativeClustering(n_clusters=3)
6. cluster.fit(data)
7. # Predict the cluster labels for each data point
8. labels = cluster.labels_
9. print(f"Clusters labels for data: {labels}")
```

Output:

```
1. Clusters labels for data: [2 0 2 1 0 2]
```

Gaussian mixture models

Gaussian Mixture Models (GMMs) are a type of soft probabilistic clustering algorithm that models' data as a mixture of Gaussian distributions. Each cluster is represented by a Gaussian distribution, and the algorithm estimates the parameters of these distributions to maximize the likelihood of the data given the model. GMMs are a powerful clustering algorithm that can be used to cluster any type of data that can be modeled by a Gaussian distribution. They are widely used in marketing to segment customers, in finance to detect fraud and in data mining to discover hidden patterns. For example, GMMs can be applied here. Imagine you have a dataset of customer transactions. You want to group customers into clusters

based on their spending patterns.

Before moving to the tutorials let us look at the syntax for implementing Gaussian mixture models with sklearn, which is as follows:

```
1. from sklearn.mixture import GaussianMixture
2. # Load the dataset
3. data = ...
4. # Create and fit the Gaussian mixture model
5. gmm = GaussianMixture(n_components=3)
6. gmm.fit(data)
```

Tutorial 8.4: To implement Gaussian mixture models using **sklearn** on a generated sample data, is as follows:

```
1. import numpy as np
2. from sklearn.mixture import GaussianMixture
3. from sklearn.datasets import make_blobs
4. # Generate some data
5. X, y = make_blobs(n_samples=100, n_features=2, centers=3, cluster_std=1.5)
6. # Create a GMM with 3 components/clusters
7. gmm = GaussianMixture(n_components=3)
8. # Fit the GMM to the data
9. gmm.fit(X)
10. # Predict the cluster labels for each data point
11. labels = gmm.predict(X)
12. print(f"Clusters labels for data: {labels}")
```

Output:

```
1. Clusters labels for data: [2 0 1 0 0 0 1 2 1 1 0 0 2 2 2 1
0
1 2 1 0 0 2 1 1 1 0 2 1 2 2 1 2 2 2 2 0
2. 1 1 1 2 0 1 2 0 0 1 1 2 0 1 0 1 0 1 2 1 0 0 1 1
2 1 2 2 0 0 2 1 0 0 2
```

```
3. 0 1 2 2 0 2 2 2 0 1 2 0 0 0 0 1 2 1 1 0 2 2 1 2 0 2]
```

Principal component analysis

Principal Component Analysis (PCA) is a linear dimensionality reduction algorithm that identifies the principal components of the data. These components represent the directions of maximum variance in the data and can be used to represent the data in a lower-dimensional space. PCA is a widely used algorithm in data visualization, machine learning and signal processing. Principal usage of dimensionality reduction is to decrease the dimensionality of high-dimensional data, such as images or text, and to preprocess data for machine learning algorithms.

For example, PCA can be applied here. Imagine you have a dataset of customer transactions. You want to group customers into clusters based on their spending patterns.

Before moving to the tutorials let us look at the syntax for implementing principal component analysis with **sklearn**, which is as follows:

```
1. from sklearn.decomposition import PCA
2. # Load the dataset
3. data = ...
4. # Create and fit the PCA model
5. pca = PCA(n_components=2)
6. pca.fit(data)
```

Tutorial 8.5: To implement principal component analysis using **sklearn** on an iris flower dataset, is as follows:

```
1. import numpy as np
2. from sklearn.datasets import load_iris
3. from sklearn.decomposition import PCA
4. # Load the Iris dataset
5. iris = load_iris()
```



```
6. X = iris.data
7. # Create a PCA model with 2 components
8. pca = PCA(n_components=2)
9. # Fit the PCA model to the data
10. X_pca = pca.fit_transform(X) #Transform the data into 2 principal components
11. print(f"Variance explained by principal components: {pca.explained_variance_ratio_}")
```

X_pca is a 2D **numpy** array of shape **(n_samples, 2)**, that contains principal component. Each row represents a sample and each column is a principal component.

Output:

```
1. Variance explained by principal components: [0.92461872 0.05306648]
```

As output shows first principal component explains 92.46% of the variance in the data. Second principal component explains 5.30% of the variance in the data.

Singular value decomposition

Singular Value Decomposition (SVD) is a linear dimensionality reduction algorithm that decomposes a matrix into three matrices: U , Σ , and V . The U matrix contains the left singular vectors of the original matrix, the Σ matrix contains the singular values of the original matrix, and the V matrix contains the right singular vectors of the original matrix. SVD can be applied to a range of tasks, such as reducing dimensionality, compressing data and extracting features. It is commonly utilized in text mining, image processing and signal processing.

For example, SVD can be applied here. Imagine you have a dataset of customer reviews. You want to summarize the reviews using a smaller set of features.

Before moving to the tutorials let us look at the syntax for

implementing singular value decomposition with sklearn, which is as follows:

1. `from numpy.linalg import svd`
2. *# Load the dataset*
3. `data = ...`
4. *# Perform the SVD*
5. `u, s, v = svd(data)`

Tutorial 8.6: To implement singular value decomposition using **sklearn** on an iris flower dataset is as follows:

1. `import numpy as np`
2. `from sklearn.decomposition import TruncatedSVD`
3. *# Load the Iris dataset*
4. `iris = load_iris()`
5. `X = iris.data`
6. *# Create a truncated SVD model with 2 components*
7. `svd = TruncatedSVD(n_components=2)`
8. *# Fit the truncated SVD model to the data*
9. `X_svd = svd.fit_transform(X)`
10. `print(f"Variance explained after singular value decomposition: {svd.explained_variance_ratio_}")`

Output:

1. `Variance explained after singular value decomposition: [0.52875361 0.44845576]`

DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm that identifies groups of data points that are densely packed together. It works by identifying core points, which are points that have a minimum number of neighbors within a specified radius. These core points form the basis of clusters and other points are assigned to clusters based

on their proximity to core points. It is useful when the number of clusters is unknown. Commonly used for data that is not well-separated, particularly in computer vision, natural language processing, and social network analysis.

For example, DBSCAN can be applied here. Imagine you have a dataset of customer locations. You want to group customers into clusters based on their proximity to each other.

Before moving to the tutorials let us look at the syntax for implementing DBSCAN with sklearn, which is as follows:

```
1. from sklearn.cluster import DBSCAN
2. # Load the dataset
3. data = ...
4. # Create and fit the DBSCAN model
5. dbscan = DBSCAN(eps=0.5, min_samples=5)
6. dbscan.fit(data)
```

Tutorial 8.7: To implement DBSCAN using **sklearn** on a generated sample data, is as follows:

```
1. import numpy as np
2. from sklearn.cluster import DBSCAN
3. from sklearn.datasets import make_moons
4. # Generate some data
5. X, y = make_moons(n_samples=200, noise=0.1)
6. # Create a DBSCAN clusterer
7. dbscan = DBSCAN(eps=0.3, min_samples=10)
8. # Fit the DBSCAN clusterer to the data
9. dbscan.fit(X)
10. # Predict the cluster labels for each data point
11. labels = dbscan.labels_
12. print(f"Clusters labels for data: {labels}")
```

Output:


```
4. # Create and fit the t-SNE model
5. tsne = TSNE(n_components=2, perplexity=30)
6. tsne.fit(data)
```

Tutorial 8.8: To implement t-SNE to reduce four dimensions into two dimensions using **sklearn** on an iris flower dataset, is as follows:

```
1. import numpy as np
2. from sklearn.datasets import load_iris
3. from sklearn.manifold import TSNE
4. # Load the Iris dataset
5. iris = load_iris()
6. X = iris.data
7. # Create a t-SNE model
8. tsne = TSNE()
9. # Fit the t-SNE model to the data
10. X_tsne = tsne.fit_transform(X)
11. # Plot the t-SNE results
12. import matplotlib.pyplot as plt
13. plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=iris.target)
14. # Define labels and colors
15. labels = ['setosa', 'versicolor', 'virginica']
16. colors = ['blue', 'orange', 'green']
17. # Create a list of handles for the legend
18. handles = [plt.plot([], [], color=c, marker='o', ls='')
               [0] for c in colors]
19. # Add the legend to the plot
20. plt.legend(handles, labels, loc='upper right')
21. # x and y labels
22. plt.xlabel('t-SNE dimension 1')
23. plt.ylabel('t-SNE dimension 2')
```

```
24. # Title
25. plt.title('t-SNE visualization of the Iris dataset')
26. # Show the figure
27. plt.savefig('TSNE.jpg',dpi=600,bbox_inches='tight')
28. plt.show()
```

Output:

Plot shows output after t-SNE technique, which reduces the dimensionality of the data from four features (sepal length, sepal width, petal length, and petal width) to two dimensions that can be visualized. Each color corresponds to flower species. It gives an idea of how the data is clustered and how the species are separated in the reduced space.

Following figure shows plot showing cluster of flowers:

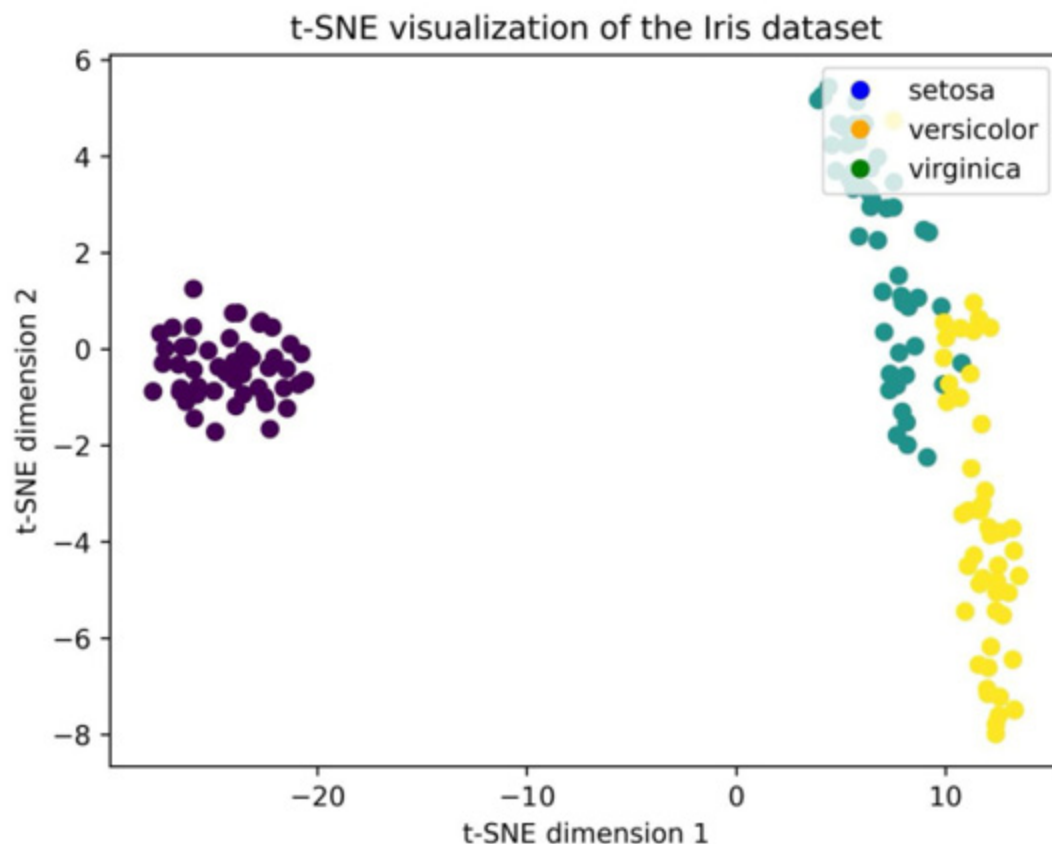


Figure 8.1: Plot showing cluster of flowers after t-SNE technique on Iris dataset

Apriori

Apriori is a frequent itemset mining algorithm that identifies frequent item sets in transactional datasets. It works by iteratively finding item sets that meet a minimum support threshold. It is often used in market basket analysis to identify patterns in customer behavior. It can also be used in other domains, such as recommender systems and fraud detection. For example, apriori can be applied here. Imagine you have a dataset of customer transactions. You want to identify common patterns of items that customers tend to buy together.

Before moving to the tutorials let us look at the syntax for implementing Apriori with **apyori** package, which is as follows:

```
1. from apyori import apriori
2. # Load the dataset
3. data = ...
4. # Create and fit the apriori model
5. rules = apriori(data, min_support=0.01, min_confidence=0.5)
```

Tutorial 8.9: To implement Apriori to find the all the frequently bought item from a grocery item dataset, is as follows:

```
1. import pandas as pd
2. from apyori import apriori
3. # Load the dataset
4. data = pd.read_csv(
5.     '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
6. # Reshape the data from wide to long format
```

```

7. data = pd.melt(data, id_vars='Channel',
8.               var_name='Product', value_name='Quantity')
9. # Group the data by customer and aggregate the product categories into a list
10. data = data.groupby('Channel')['Product'].apply(list)
11. # Convert the data into a list of lists
12. data = data.tolist()
13. # Create the apriori model
14. rules = apriori(data, min_support=0.00003)
15. # Print the rules
16. for rule in rules:
17.     for rule in rules:
18.         print(list(rule.items))

```

Tutorial 8.9 output will display the items in each frequent item set as a list.

Tutorial 8.10: To implement Apriori, to view only the first five frequent items from a grocery item dataset, is as follows:

```

1. import pandas as pd
2. from apyori import apriori
3. # Load the dataset
4. data = pd.read_csv(
5.     '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
6. # Reshape the data from wide to long format
7. data = pd.melt(data, id_vars='Channel',
8.               var_name='Product', value_name='Quantity')
9. # Group the data by customer and aggregate the product categories into a list
10. data = data.groupby('Channel')['Product'].apply(list)
11. # Convert the data into a list of lists

```



```

12. data = data.tolist()
13. # Create the apriori model
14. rules = apriori(data, min_support=0.00003)
15. # Print the rules and the first 5 elements
16. rules = list(rules)
17. rules = rules[:5]
18. for rule in rules:
19.     for item in rule.items:
20.         print(item)

```

Output:

1. Delicassen
2. Detergents_Paper
3. Fresh
4. Frozen
5. Grocery

Tutorial 8.11: To implement Apriori, to view all most frequent items with the support value of each itemset from the grocery item dataset, is as follows:

```

1. import pandas as pd
2. from apyori import apriori
3. # Load the dataset
4. data = pd.read_csv(
5.     '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
6. # Reshape the data from wide to long format
7. data = pd.melt(data, id_vars='Channel',
8.                 var_name='Product', value_name='Quantity')
9. # Group the data by customer and aggregate the product categories into a list
10. data = data.groupby('Channel')['Product'].apply(list)
11. # Convert the data into a list of lists

```

```

12. data = data.tolist()
13. # Create the apriori model
14. rules = apriori(data, min_support=0.00003)
15. # Print the rules
16. for rule in rules:
17.     # Join the items in the itemset with a comma
18.     itemset = ", ".join(rule.items)
19.     # Get the support value of the itemset
20.     support = rule.support
21.     # Print the itemset and the support in one line
22.     print("{}: {}".format(itemset, support))

```

Eclat

Eclat is a frequent itemset mining algorithm similar to Apriori, but more efficient for large datasets. It works by using a vertical data format to represent transactions. It is also used in market basket analysis to identify patterns in customer behavior. It can also be used in other areas such as recommender systems and fraud detection. For example, Eclat can be applied here. Imagine you have a dataset of customer transactions. You want to identify frequent item sets in transactional datasets efficiently.

Tutorial 8.12: To implement frequent item data mining using a sample data set of transactions, is as follows:

```

1. # Define a function to convert the data from horizontal to vertical format
2. def horizontal_to_vertical(data):
3.     # Initialize an empty dictionary to store the vertical format
4.     vertical = {}
5.     # Loop through each transaction in the data
6.     for i, transaction in enumerate(data):

```

```
7.     # Loop through each item in the transaction
8.     for item in transaction:
9.         # If the item is already in the dictionary, append the
           transaction ID to its value
10.        if item in vertical:
11.            vertical[item].append(i)
12.        # Otherwise, create a new key-
           value pair with the item and the transaction ID
13.        else:
14.            vertical[item] = [i]
15.    # Return the vertical format
16.    return vertical
17. # Define a function to generate frequent item sets using
    the ECLAT algorithm
18. def eclat(data, min_support):
19.    # Convert the data to vertical format
20.    vertical = horizontal_to_vertical(data)
21.    # Initialize an empty list to store the frequent item sets
22.    frequent = []
23.    # Initialize an empty list to store the candidates
24.    candidates = []
25.    # Loop through each item in the vertical format
26.    for item in vertical:
27.        # Get the support count of the item by taking the leng
           th of its value
28.        support = len(vertical[item])
29.        # If the support count is greater than or equal to the
           minimum support, add the item to the frequent list and t
           he candidates list
30.        if support >= min_support:
31.            frequent.append((item, support))
```

```

32.     candidates.append((item, vertical[item]))
33.     # Loop until there are no more candidates
34.     while candidates:
35.         # Initialize an empty list to store the new candidates
36.         new_candidates = []
37.         # Loop through each pair of candidates
38.         for i in range(len(candidates) - 1):
39.             for j in range(i + 1, len(candidates)):
40.                 # Get the first item set and its transaction IDs from the first candidate
41.                 itemset1, tidset1 = candidates[i]
42.                 # Get the second item set and its transaction IDs from the second candidate
43.                 itemset2, tidset2 = candidates[j]
44.                 # If the item sets have the same prefix, they can be combined
45.                 if itemset1[:-1] == itemset2[:-1]:
46.                     # Combine the item sets by adding the last element of the second item set to the first item set
47.                     new_itemset = itemset1 + itemset2[-1]
48.                     # Intersect the transaction IDs to get the support count of the new item set
49.                     new_tidset = list(set(tidset1) & set(tidset2))
50.                     new_support = len(new_tidset)
51.                     # If the support count is greater than or equal to the minimum support, add the new item set to the frequent list and the new candidates list
52.                     if new_support >= min_support:
53.                         frequent.append((new_itemset, new_support))
54.                         new_candidates.append((new_itemset, new_tidset))

```

```
55.  # Update the candidates list with the new candidates
56.  candidates = new_candidates
57.  # Return the frequent item sets
58.  return frequent
59. # Define a sample data set of transactions
60. data = [
61.  ["A", "B", "C", "D"],
62.  ["A", "C", "E"],
63.  ["A", "B", "C", "E"],
64.  ["B", "C", "D"],
65.  ["A", "B", "C", "D", "E"]
66. ]
67. # Define a minimum support value
68. min_support = 3
69. # Call the eclat function with the data and the minimum support
70. frequent = eclat(data, min_support)
71. # Print the frequent item sets and their support counts
72. for itemset, support in frequent:
73.  print(itemset, support)
```

Output:

```
1. A 4
2. B 4
3. C 5
4. D 3
5. E 3
6. AB 3
7. AC 4
8. AE 3
9. BC 4
```

10. BD 3
11. CD 3
12. CE 3
13. ABC 3
14. ACE 3
15. BCD 3

FP-Growth

FP-Growth is a frequent itemset mining algorithm based on the FP-tree data structure. It works by recursively partitioning the dataset into smaller subsets and then identifying frequent item sets in each subset. FP-Growth is a popular association rule mining algorithm that is often used in market basket analysis to identify patterns in customer behavior. It is also used in recommendation systems and fraud detection. For example, FP-Growth can be applied here. Imagine you have a dataset of customer transactions. You want to identify frequent item sets in transactional datasets efficiently using a pattern growth approach.

Before moving to the tutorials let us look at the syntax for implementing FP-Growth with **mlxtend.frequent_patterns**, which is as follows:

1. `from mlxtend.frequent_patterns import fpgrowth`
2. *# Load the dataset*
3. `data = ...`
4. *# Create and fit the FP-Growth model*
5. `patterns = fpgrowth(data, min_support=0.01, use_colnames=True)`

Tutorial 8.13: To implement frequent item for data mining using FP-Growth using **mlxtend.frequent_patterns**, as follows:

1. `import pandas as pd`

```
2. # Import fpgrowth function from mlxtend library for frequent pattern mining
3. from mlxtend.frequent_patterns import fpgrowth
4. # Import TransactionEncoder class from mlxtend library for encoding data
5. from mlxtend.preprocessing import TransactionEncoder
6. # Define a list of transactions, each transaction is a list of items
7. data = [["A", "B", "C", "D"],
8. ["A", "C", "E"],
9. ["A", "B", "C", "E"],
10. ["B", "C", "D"],
11. ["A", "B", "C", "D", "E"]]
12. # Create an instance of TransactionEncoder
13. te = TransactionEncoder()
14. # Fit and transform the data to get a boolean matrix
15. te_ary = te.fit(data).transform(data)
16. # Convert the matrix to a pandas dataframe with column names as items
17. df = pd.DataFrame(te_ary, columns=te.columns_)
18. # Apply fpgrowth algorithm on the dataframe with a minimum support of 0.8
19. # and return the frequent itemsets with their corresponding support values
20. fpgrowth(df, min_support=0.8, use_colnames=True)
```

Output:

```
1. support itemsets
2. 0 1.0 (C)
3. 1 0.8 (B)
4. 2 0.8 (A)
5. 3 0.8 (B, C)
```

Model selection and evaluation

Unlike supervised learning, unsupervised learning methods commonly use evaluation metrics such as **Silhouette Score (SI)**, **Davies-Bouldin Index (DI)**, **Calinski-Harabasz Index (CI)** and **Adjusted Rand Index (RI)** to check performance and quality of machine learning models.

Evaluation metrics and model selection for unsupervised

Evaluation matrices may vary depending on type of unsupervised learning problem. Although, SI, DI, CI and RI are useful to evaluate the clustering results. The silhouette score measures how well each data point fits into its assigned cluster, based on the average distance to other data points in the same cluster and the nearest cluster. Its score ranges from -1 to 1 with higher values indicating better clustering. DI measures the average similarity between each cluster and its most similar cluster, based on the ratio of intra-cluster distances to inter-cluster distances. The index ranges from zero to infinity with lower values indicating better clustering. CI measures the ratio of the between-cluster variance to the within-cluster variance, based on the sum of the squared distances of the data points to their cluster centroids. The index ranges from zero to infinity, with higher values indicating better clustering. RI measures the similarity between two clusters of the same data set, based on the number of pairs of examples assigned to the same or different clusters in both clustering. Its index ranges from -1 to 1, with higher values indicating better agreement. Here too, the performance, complexity, interpretability and resource requirements remain the selection criteria.

Tutorial 8.14 with snippets, explains how to use some common model selection and evaluation techniques for unsupervised learning.

Tutorial 8.14: To implement a tutorial that illustrates model selection and evaluation in unsupervised machine learning using iris data is as follow:

To begin, we import the required modules and load the iris dataset, by taking all the features only excluding the label, as demonstrated. The aim is to determine the optimal number of clusters from the dataset and assess the result with evaluation matrices as follows:

```
1. import numpy as np
2. import pandas as pd
3. import matplotlib.pyplot as plt
4. from sklearn.datasets import load_iris
5. from sklearn.cluster import KMeans, AgglomerativeClustering
6. from sklearn.metrics import silhouette_score, davies_bouldin_score, calinski_harabasz_score, adjusted_rand_score
7. # Load dataset
8. iris = load_iris()
9. X = iris.data # Features
10. y = iris.target # True labels
11. print(iris.feature_names)
```

Output:

```
1. ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

We define clustering models to compare, including K-means, agglomerative clustering and define SI, DI, CI and RI metrics to evaluate the models, as follows:

```
1. # Define candidate models
```

```

2. models = {
3.     'K-means': KMeans(),
4.     'Agglomerative Clustering': AgglomerativeClustering(
5.     )
6. }
7. # Evaluate models using multiple metrics
8. metrics = {
9.     'Silhouette score': silhouette_score,
10.    'Davies-Bouldin index': davies_bouldin_score,
11.    'Calinski-Harabasz index': calinski_harabasz_score,
12.    'Adjusted Rand index': adjusted_rand_score
13. }

```

To evaluate the quality of each cluster, we fit the model and plot the results.

```

1. # Fit model, get cluster labels, compare results
2. scores = {}
3. for name, model in models.items():
4.     labels = model.fit_predict(X)
5.     scores[name] = {}
6.     for metric_name, metric in metrics.items():
7.         if metric_name == 'Adjusted Rand index':
8.             score = metric(y, labels) # Compare true labels
9.             and predicted labels
10.        else:
11.            score = metric(X, labels) # Compare features and
12.            predicted labels
13.            scores[name][metric_name] = score
14.            print(f'{name}, {metric_name}: {score:.3f}')
15. # Plot scores
16. fig, ax = plt.subplots(2, 2, figsize=(10, 10))
17. for i, metric_name in enumerate(metrics.keys()):

```

```

16. row = i // 2
17. col = i % 2
18. ax[row, col].bar(scores.keys(), [score[metric_name] f
    or score in scores.values()])
19. ax[row, col].set_ylabel(metric_name)
20. # Save the figure
21. plt.savefig('Clustering_model_selection_and_evaluation.
    png', dpi=600, bbox_inches='tight')

```

Output:

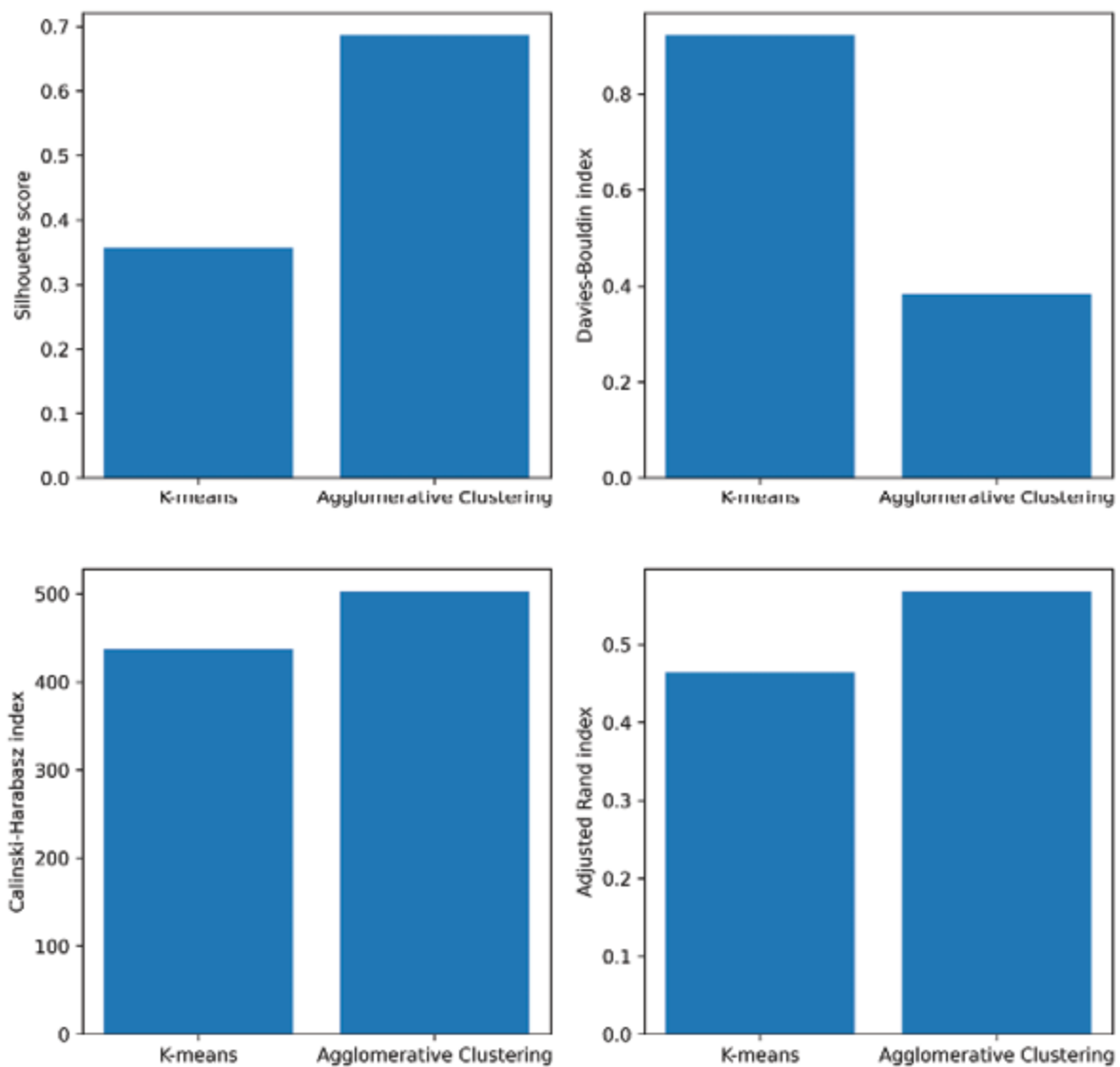


Figure 8.2. *Plot comparing the SI, CI, DI, RI scores of different unsupervised algorithms on the iris dataset*

[Figure 8.2](#) and the SI, CI, DI, RI scores show that agglomerative clustering performs better than K-means on the iris dataset according to all four metrics. Agglomerative clustering has a higher SI score, which means that the clusters are more cohesive and well separated. It also has a lower DI, which means that the clusters are more distinct and less overlapping. In addition, agglomerative clustering has a higher CI score, which means that the clusters have a higher ratio of inter-cluster variance to intra-cluster variance. Finally, agglomerative clustering has a higher RI, which means that the predicted labels are more consistent with the true labels. Therefore, agglomerative clustering is a better model choice for this data.

Conclusion

In this chapter, we explored unsupervised learning and algorithms for uncovering hidden patterns and structures within unlabeled data. We delved into prominent clustering algorithms like K-means, K-prototype, and hierarchical clustering, along with probabilistic approaches like Gaussian mixture models. Additionally, we covered dimensionality reduction techniques like PCA and SVD for simplifying complex datasets. This knowledge lays a foundation for further exploration of unsupervised learning's vast potential in various domains. From customer segmentation and anomaly detection to image compression and recommendation systems, unsupervised learning plays a vital role in unlocking valuable insights from unlabeled data.

We hope that this chapter has helped you understand and apply the concepts and methods of statistical machine learning, and that you are motivated and inspired to learn more and apply these techniques to your own data and

problems.

The next *Chapter 9, Linear Algebra, Nonparametric Statistics, and Time Series Analysis* explores time series data, linear algebra and nonparametric statistics.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[**https://discord.bpbonline.com**](https://discord.bpbonline.com)



CHAPTER 9

Linear Algebra, Nonparametric Statistics, and Time Series Analysis

Introduction

This chapter explores the essential mathematical foundations, statistical techniques, and methods for analyzing time-dependent data. We will cover three interconnected topics: linear algebra, nonparametric statistics, and time series analysis, incorporating survival analysis. The journey begins with linear algebra, where we will unravel key concepts such as linear functions, vectors, and matrices, providing a solid framework for understanding complex data structures. Nonparametric statistics will enable us to analyze data without the restrictive assumptions of parametric models. We will explore techniques like rank-based tests and kernel density estimation, which offer flexibility in analyzing a wide range of data types.

Time series data, prevalent in diverse areas such as stock prices, weather patterns, and heart rate variability, will be examined with a focus on trend and seasonality analysis. In

the realm of survival analysis, where life events such as disease progression, customer churn, or equipment failure are unpredictable, we will delve into the analysis of time-to-event data. We will demystify techniques such as Kaplan-Meier estimators, making survival analysis accessible and understandable. Throughout the chapter, each concept will be illustrated with practical examples and real-world applications, providing a hands-on guide for implementation.

Structure

In this chapter, we will discuss the following topics:

- Linear algebra
- Nonparametric statistics
- Survival analysis
- Time series analysis

Objectives

This chapter provides the reader with the necessary tools, the ability to gain insight, the understanding of the theory and the ways to implement linear algebra, nonparametric statistics and time series analysis techniques with Python. By the last page, you will be armed with the knowledge to tackle complex data challenges and interpret results with clarity about these topics.

Linear algebra

Linear algebra is a branch of mathematics that focuses on the study of vectors, vector spaces and linear transformations. It deals with linear equations, linear functions and their representations through matrices and determinants.

Let us understand vectors, linear function and matrices in

linear algebra.

Following is the explanation of vectors:

- **Vectors:** Vectors are a fundamental concept in linear algebra as they represent quantities that have both magnitude and direction. Examples of such quantities include velocity, force and displacement. In statistics, vectors organize data points. Each data point can be represented as a vector, where each component corresponds to a specific feature or variable.

Tutorial 9.1: To create a 2D vector with NumPy and display, is as follows:

```
1. import numpy as np
2. # Create a 2D vector
3. v = np.array([3, 4])
4. # Access individual components
5. x, y = v
6. # Calculate magnitude (Euclidean norm) of the vector
7. magnitude = np.linalg.norm(v)
8. print(f"Vector v: {v}")
9. print(f"Components: x = {x}, y = {y}")
10. print(f"Magnitude: {magnitude:.2f}")
```

Output:

```
1. Vector v: [3 4]
2. Components: x = 3, y = 4
3. Magnitude: 5.00
```

- **Linear function:** A linear function is represented by the equation $f(x) = ax + b$, where a and b are constants. They model relationships between variables. For example, linear regression shows how a dependent variable changes linearly with respect to an independent variable.

Tutorial 9.2: To create a simple linear function, $f(x) =$

$2x + 3$ and plot it, is as follows:

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. # Define a linear function:  $f(x) = 2x + 3$ 
4. def linear_function(x):
5.     return 2 * x + 3
6. # Generate x values
7. x_values = np.linspace(-5, 5, 100)
8. # Calculate corresponding y values
9. y_values = linear_function(x_values)
10. # Plot the linear function
11. plt.plot(x_values, y_values, label="f(x) = 2x + 3")
12. plt.xlabel("x")
13. plt.ylabel("f(x)")
14. plt.title("Linear Function")
15. plt.grid(True)
16. plt.legend()
17. plt.savefig("linearfunction.jpg",dpi=600,bbox_inches
    ='tight')
18. plt.show()
```

Output:

It plots the $f(x) = 2x + 3$ as shown in [Figure 9.1](#):

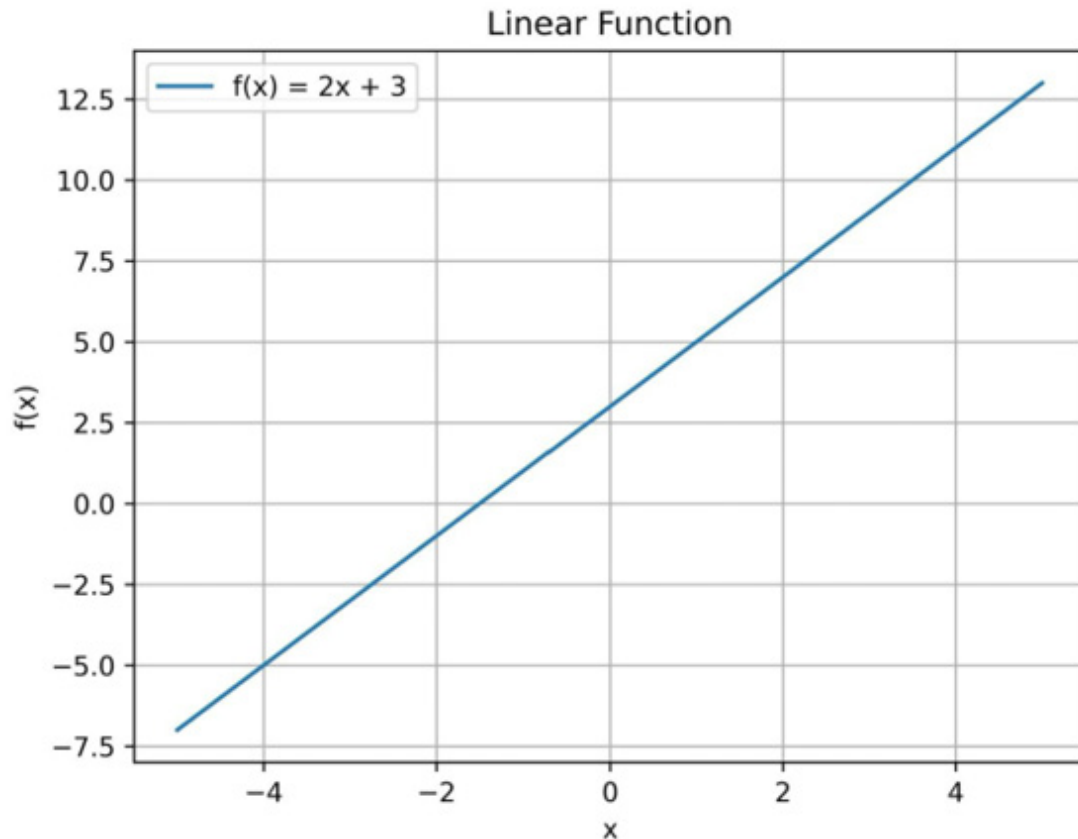


Figure 9.1: Plot of a linear function

- **Matrices:** Matrices are rectangular arrays of numbers that are commonly used to represent systems of linear equations and transformations. In statistics, matrices are used to organize data, where rows correspond to observations and columns represent variables. For example, a dataset with height, weight, and age can be represented as a matrix.

Tutorial 9.3: To create a matrix (rectangular array) of numbers with NumPy and transpose it, as follows:

1. `import numpy as np`
2. `# Create a 2x3 matrix`
3. `A = np.array([[1, 2, 3],`
4. `[4, 5, 6]])`
5. `# Access individual elements`
6. `element_23 = A[1, 2]`

```
7. # Transpose the matrix
8. A_transposed = A.T
9. print(f"Matrix A:\n{A}")
10. print(f"Element at row 2, column 3: {element_23}")
11. print(f"Transposed matrix A:\n{A_transposed}")
```

Output:

```
1. Matrix A:
2. [[1 2 3]
3.  [4 5 6]]
4. Element at row 2, column 3: 6
5. Transposed matrix A:
6. [[1 4]
7.  [2 5]
8.  [3 6]]
```

Linear algebra models and analyses relationships between variables, aiding our comprehension of how changes in one variable affect another. Its further application include cryptography to create solid encryption techniques, regression analysis, dimensionality reduction and solving systems of linear equations. We discussed this earlier in [Chapter 7, Statistical Machine Learning](#) on linear regression. For example, imagine we want to predict a person's weight based on their height. We collect data from several individuals and record their heights (in inches) and weights (in pounds). Linear regression allows us to create a straight line (a linear model) that best fits the data points (height and weight). Using this method, we can predict someone's weight based on their height using the linear equation. The use and implementation of linear algebra in statistics is shown in the following tutorials:

Tutorial 9.4: To illustrate the use of linear algebra, solve a linear system of equations using the linear algebra submodule of SciPy, is as follows:

```

1. import numpy as np
2. # Import the linear algebra submodule of SciPy and assign it the alias "la"
3. import scipy.linalg as la
4. A = np.array([[1, 2], [3, 4]])
5. b = np.array([3, 17])
6. # Solving a linear system of equations
7. x = la.solve(A, b)
8. print(f"Solution x: {x}")
9. print(f"Check if A @ x equals b: {np.allclose(A @ x, b)}")

```

Output:

```

1. Solution x: [11. -4.]
2. Check if A @ x equals b: True

```

Tutorial 9.5: To illustrate the use of linear algebra in statistics to compare performance, solving vs. inverting for linear systems, using SciPy, is as follows:

```

1. import numpy as np
2. import scipy.linalg as la
3. A1 = np.random.random((1000, 1000))
4. b1 = np.random.random(1000)
5. # Uses %timeit magic command to measure the execution time of la.solve(A1, b1) and la.solve solves linear equations
6. solve_time = %timeit -o la.solve(A1, b1)
7. # Measures the time for solving by first inverting A1 using la.inv(A1) and then multiplying the inverse with b1.
8. inv_time = %timeit -o la.inv(A1) @ b1
9. # Prints the best execution time for la.solve method in milliseconds
10. print(f"Solve time: {solve_time.best:.2f} ms")
11. # Prints the best execution time for the inversion method in milliseconds

```

```
12. print(f'Inversion time: {inv_time.best:.2f} ms')
```

Output:

1. 31.3 ms $\pm$ 4.05 ms per loop (mean $\pm$ std. dev. of 7 runs, 10 loops each)
2. 112 ms $\pm$ 4.51 ms per loop (mean $\pm$ std. dev. of 7 runs, 10 loops each)
3. Solve time: 0.03 ms
4. Inversion time: 0.11 ms

Tutorial 9.6: To illustrate the use of linear algebra in statistics to perform basic matrix properties, using the linear algebra submodule of SciPy, is as follows:

1. `import numpy as np`
2. `import scipy.linalg as la`
3. `# Create a complex matrix C`
4. `C = np.array([[1, 2 + 3j], [3 - 2j, 4]])`
5. `# Print the conjugate of C (element-wise complex conjugate)`
6. `print(f'Conjugate of C:\n{C.conjugate()}')`
7. `# Print the trace of C (sum of diagonal elements)`
8. `print(f'Trace of C: {np.diag(C).sum()}')`
9. `# Print the matrix rank of C (number of linearly independent rows/columns)`
10. `print(f'Matrix rank of C: {np.linalg.matrix_rank(C)}')`
11. `# Print the Frobenius norm of C (square root of sum of squared elements)`
12. `print(f'Frobenius norm of C: {la.norm(C, None)}')`
13. `# Print the largest singular value of C (largest eigenvalue of C*C.conjugate())`
14. `print(f'Largest singular value of C: {la.norm(C, 2)}')`
15. `# Print the smallest singular value of C (smallest eigenvalue of C*C.conjugate())`
16. `print(f'Smallest singular value of C: {la.norm(C, -2)}')`

Output:

1. Conjugate of C:
2. `[[1.-0.j 2.-3.j]`
3. `[3.+2.j 4.-0.j]`
4. Trace of C: `(5+0j)`
5. Matrix rank of C: 2
6. Frobenius norm of C: 6.557438524302
7. Largest singular value of C: 6.389028023601217
8. Smallest singular value of C: 1.4765909770949925

Tutorial 9.7: To illustrate the use of linear algebra in statistics to compute the least squares solution in a square matrix, using the linear algebra submodule of SciPy, is as follows:

1. `import numpy as np`
2. `import scipy.linalg as la`
3. `# Define a square matrix A1 and vector b1`
4. `A1 = np.array([[1, 2], [2, 4]])`
5. `b1 = np.array([3, 17])`
6. `# Attempt to solve the system of equations A1x = b1 using la.solve`
7. `try:`
8. `x = la.solve(A1, b1)`
9. `print(f"Solution using la.solve: {x}") # Print solution if successful`
10. `except la.LinAlgError as e: # Catch potential error if matrix is singular`
11. `print(f"Error using la.solve: {e}") # Print error message`
12. `# # Compute least-squares solution`
13. `x, residuals, rank, s = la.lstsq(A1, b1)`
14. `print(f"Least-squares solution x: {x}")`

Output:

1. Error using la.solve: Matrix is singular.
2. Least-squares solution x: [1.48 2.96]

Tutorial 9.8: To illustrate the use of linear algebra in statistics to compute the least squares solution of a random matrix, using the linear algebra submodule of SciPy, is as follows:

1. `import numpy as np`
2. `import scipy.linalg as la`
3. `import matplotlib.pyplot as plt`
4. `A2 = np.random.random((10, 3))`
5. `b2 = np.random.random(10)`
6. *#Computing least square from random matrix*
7. `x, residuals, rank, s = la.lstsq(A2, b2)`
8. `print(f"Least-squares solution for random A2: {x}")`

Output:

1. Least-squares solution for random A2: [0.34430232 0.54211796 0.18343947]

Tutorial 9.9: To illustrate the implementation of linear regression to predict car prices based on historical data, is as follows:

1. `import numpy as np`
2. `from scipy import linalg`
3. *# Sample data: car prices (in thousands of dollars) and features*
4. `prices = np.array([20, 25, 30, 35, 40])`
5. `features = np.array([[2000, 150],`
6. `[2500, 180],`
7. `[2800, 200],`
8. `[3200, 220],`
9. `[3500, 240]])`
10. *# Fit a linear regression model*

```
11. coefficients, residuals, rank, singular_values = linalg.lstsq(features, prices)
12. # Predict price for a new car with features [3000, 170]
13. new_features = np.array([3000, 170])
14. # Calculate predicted price using the dot product of the new features and their corresponding coefficients
15. predicted_price = np.dot(new_features, coefficients)
16. print(f"Predicted price: ${predicted_price:.2f}k")
```

Output:

```
1. Predicted price: $41.60k
```

Nonparametric statistics

Nonparametric statistics is a branch of statistics that does not rely on specific assumptions about the underlying probability distribution. Unlike parametric statistics, which assume that data follow a particular distribution (such as the normal distribution), nonparametric methods are more flexible and work well with different types of data. Nonparametric statistics make inferences without assuming a particular distribution. They often use ordinal data (based on rankings) rather than numerical values. As mentioned unlike parametric methods, nonparametric statistics do not estimate specific parameters (such as mean or variance) but focus on the overall distribution.

Let us understand nonparametric statistics and its use through an example of clinical trial rating, as follows:

- **Clinical trial rating:** Imagine that a researcher is conducting a clinical trial to evaluate the effectiveness of a new pain medication. Participants are asked to rate their treatment experience on a scale of one to five (where one is *very poor* and five is *excellent*). The data collected consist of ordinal ratings, not continuous numerical values. These ratings are inherently

nonparametric because they do not follow a specific distribution.

- To analyze the treatment's impact, the researcher can apply nonparametric statistical tests like the Wilcoxon signed-rank test. Wilcoxon signed-rank test is a statistical method used to compare paired data, specifically when you want to assess whether there is a significant difference between two related groups. It compares the median ratings before and after treatment and does not assume a normal distribution and is suitable for paired data.
- **Hypotheses:**
 - **Null hypothesis (H_0):** The median rating before treatment is equal to the median rating after treatment.
 - **Alternative hypothesis (H_1):** The median rating differs before and after treatment.
- If the p-value from the test is small (typically less than 0.05), we reject the null hypothesis, indicating a significant difference in treatment experience.

This example shows that nonparametric methods allow us to make valid statistical inferences without relying on specific distributional assumptions. They are particularly useful when dealing with ordinal data or situations where parametric assumptions may not hold.

Tutorial 9.10: To illustrate the use of nonparametric statistics to compare treatment ratings (ordinal data). We collect treatment ratings (ordinal data) before and after a new drug. We want to know if the drug improves the patient's experience, as follows:

1. `import numpy as np`
2. `from scipy.stats import wilcoxon`
3. *# Example data (ratings on a scale of 1 to 5)*

```

4. before_treatment = [3, 4, 2, 3, 4]
5. after_treatment = [4, 5, 3, 4, 5]
6. # Null Hypothesis ( $H_0$ ): The median treatment rating before the new drug is equal to the median rating after the drug.
7. # Alternative Hypothesis ( $H_1$ ): The median rating differs before and after the drug.
8. # Perform Wilcoxon signed-rank test
9. statistic, p_value = wilcoxon(before_treatment, after_treatment)
10. if p_value < 0.05:
11.     print("P-value:", p_value)
12.     print("P-value is less than 0.05, so reject the null hypothesis, we can confidently say that the new drug led to better treatment experience.")
13. else:
14.     print("P-value:", p_value)
15.     print("No significant change")
16.     print("P value is greater than or equal to 0.05, so we cannot reject the null hypothesis and therefore cannot conclude that the drug had a significant effect.")

```

Output:

```

1. P-value: 0.0625
2. No significant change
3. P value is greater than or equal to 0.05, so we cannot reject the null hypothesis and therefore cannot conclude that the drug had a significant effect.

```

Nonparametric statistics relies on statistical methods that do not assume a specific distribution for the data, making them versatile for a wide range of applications where traditional parametric assumptions may not hold. In this

section, we will explore some key nonparametric methods, including **rank-based tests**, **goodness-of-fit tests**, and **independence tests**. Rank-based tests, such as the **Kruskal-Wallis test**, allow for comparisons across groups without relying on parametric distributions. **Goodness-of-fit tests**, like the chi-square test, assess how well observed data align with expected distributions, while **independence tests**, such as Spearman's rank correlation or Fisher's exact test, evaluate relationships between variables without assuming linearity or normality. Additionally, resampling techniques like **bootstrapping** provide robust estimates of confidence intervals and other statistics, bypassing the need for parametric assumptions. These nonparametric methods are essential tools for data analysis when distributional assumptions are difficult to justify. Let us explore some key nonparametric methods:

Rank-based tests

Rank-based tests compare rankings or orders of data points between groups. It includes Mann-Whitney U test (Wilcoxon rank-sum test) and Wilcoxon signed-rank test. The Mann-Whitney U test compares medians between two independent groups (e.g., treatment vs. control group). It determines if their distributions differ significantly and is useful when assumptions of normality are violated. Wilcoxon signed-rank test compares paired samples (e.g., before and after treatment), as in *Tutorial 9.10*. It tests if the median difference is zero and is robust to non-gaussian data.

Goodness-of-fit tests

Goodness-of-fit tests assess whether observed data fits a specific distribution. It includes chi-squared goodness-of-fit test. This test checks if observed frequencies match expected frequencies in different categories. Suppose you are a data analyst working for a shop owner who claims that

an equal number of customers visit the shop each weekday. To test this hypothesis, you record the number of customers that come into the shop during a given week, as follows:

Days	Monday	Tuesday	Wednesday	Thursday	Friday
Number of Customers	50	60	40	47	53

Table 9.1: *Number of customers per week days*

Using this data, we determine whether the observed distribution of customers across weekdays matches the expected distribution (equal number of customers each day).

Tutorial 9.11: To implement chi-square goodness of fit test to see if the observed distribution of customers across weekdays matches the expected distribution (equal number of customers each day), is as follows:

1. `import scipy.stats as stats`
2. *# Create two arrays to hold the observed and expected number of customers for each day*
3. `expected = [50, 50, 50, 50, 50]`
4. `observed = [50, 60, 40, 47, 53]`
5. *# Perform Chi-Square Goodness of Fit Test using chisquare function*
6. *# Null Hypothesis (H_0): The variable follows the hypothesized distribution (equal number of customers each day).*
7. *# Alternative Hypothesis (H_1): The variable does not follow the hypothesized distribution.*
8. *# Chisquare function takes two arrays: f_{obs} (observed counts) and f_{exp} (expected counts).*
9. *# By default, it assumes that each category is equally likely.*
10. `result = stats.chisquare(f_obs=observed, f_exp=expected)`

```
11. print("Chi-Square Statistic:", round(result.statistic, 3))
12. print("p-value:", round(result.pvalue, 3))
```

Output:

1. Chi-Square Statistic: 4.36
2. p-value: 0.359

The chi-square test statistic is calculated as 4.36, and the corresponding p-value is 0.359. Since the p-value is not less than 0.05 (our significance level), we fail to reject the null hypothesis. This means we do not have sufficient evidence to say that the true distribution of customers is different from the distribution claimed by the shop owner.

Independence tests

Independence tests determine if two categorical variables are independent. It includes chi-squared test of independence and Kendall's tau or Spearman's rank correlation. Chi-squared test of independence examines association between variables in a contingency table, as discussed in earlier in [Chapter 6, Hypothesis Testing and Significance Tests](#). Kendall's tau or Spearman's rank correlation assess correlation between ranked variables.

Suppose two basketball coaches rank 12 players from worst to best. The rankings assigned by each coach are as follows:

Players	Coach #1 Rank	Coach #2 Rank
A	1	2
B	2	1
C	3	3
D	4	5
E	5	4
F	6	6

G	7	8
H	8	7
I	9	9
J	10	11
K	11	10
L	12	12

Table 9.2: *Rankings assigned by each coach*

Using this we can calculate Kendall's Tau, let us calculate Kendall's Tau to assess the correlation between the two coaches' rankings. A positive Tau indicates a positive association, while a negative tau indicates a negative association. The closer Tau is to 1 or -1, the stronger the association. A Tau of zero indicates no association.

Tutorial 9.12: To calculate Kendall's Tau to assess the correlation between the two coaches' rankings, is as follows:

```

1. import scipy.stats as stats
2. # Coach #1 rankings
3. coach1_ranks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
4. # Coach #2 rankings
5. coach2_ranks = [2, 1, 3, 5, 4, 6, 8, 7, 9, 11, 10, 12]
6. # Calculate concordant and discordant pairs
7. concordant = 0
8. discordant = 0
9. n = len(coach1_ranks)
10. # Iterate through all pairs of players (i, j) where i < j
11. for i in range(n):
12.     for j in range(i + 1, n):
13.         # Check if both coaches ranked player i higher than
           player j (concordant pair)

```

```

14.      # or both coaches ranked player i lower than player
       j (also concordant pair)
15.      if (coach1_ranks[i] < coach1_ranks[j] and coach2_r
       ranks[i] < coach2_ranks[j]) or \
16.      (coach1_ranks[i] > coach1_ranks[j] and coach2_r
       ranks[i] > coach2_ranks[j]):
17.          concordant += 1
18.      # Otherwise, it's a discordant pair
19.      elif (coach1_ranks[i] < coach1_ranks[j] and coach2
       _ranks[i] > coach2_ranks[j]) or \
20.      (coach1_ranks[i] > coach1_ranks[j] and coach2_
       ranks[i] < coach2_ranks[j]):
21.          discordant += 1
22. # Calculate Kendall's Tau
23. tau = (concordant - discordant) / (concordant + discorda
       nt)
24. print("Kendall's Tau:", round(tau, 3))

```

Output:

```
1. Kendall's Tau: 0.879
```

Kendall's Tau of 0.879 indicates a strong positive association between the two ranked variables. In other words, the rankings assigned by the two coaches are closely related, and their preferences align significantly.

Kruskal-Wallis test

Kruskal-Wallis test is nonparametric alternative to one-way ANOVA. It allows to compare medians across multiple independent groups and generalizes the Mann-Whitney test. Suppose researchers want to determine if three different fertilizers lead to different levels of plant growth. They randomly select 30 different plants and split them into three groups of 10, applying a different fertilizer to each group. After one month, they measure the height of each plant.

Tutorial 9.13: To implement the Kruskal-Wallis test to compare median heights across multiple groups, is as follows:

```
1. from scipy import stats
2. # Create three arrays to hold the plant measurements for each of the three groups
3. group1 = [7, 14, 14, 13, 12, 9, 6, 14, 12, 8]
4. group2 = [15, 17, 13, 15, 15, 13, 9, 12, 10, 8]
5. group3 = [6, 8, 8, 9, 5, 14, 13, 8, 10, 9]
6. # Perform Kruskal-Wallis Test
7. # Null hypothesis ( $H_0$ ): The median is equal across all groups.
8. # Alternative hypothesis ( $H_a$ ): The median is not equal across all groups
9. result = stats.kruskal(group1, group2, group3)
10. print("Kruskal-Wallis Test Statistic:", round(result.statistic, 3))
11. print("p-value:", round(result.pvalue, 3))
```

Output:

1. Kruskal-Wallis Test Statistic: 6.288
2. p-value: 0.043

Here, p-value is less than our chosen significance level (e.g., 0.05), so we reject the null hypothesis. We conclude that the type of fertilizer used leads to statistically significant differences in plant growth.

Bootstrapping

Bootstrapping is a resampling technique to estimate parameters or confidence intervals. Like bootstrapping the mean or median from a sample. Bootstrapping is a resampling technique that generates simulated samples by repeatedly drawing from the original dataset. Each simulated sample is the same size as the original sample. By

creating these simulated samples, we can explore the variability of sample statistics and make inferences about the population. It is especially useful when population distribution is unknown or does not follow a standard form. Sample sizes are small. You want to estimate parameters (e.g., mean, median) or construct confidence intervals.

For example, imagine we have a dataset of exam scores (sampled from an unknown population). We resample the exam scores with replacement to create bootstrap samples. We want to estimate the mean exam score and create a bootstrapped confidence interval. The bootstrapped mean provides an estimate of the population mean. The confidence interval captures the uncertainty around this estimate.

Tutorial 9.14: To implement nonparametric statistical method bootstrapping to bootstrap the mean or median from a sample, is as follows:

```
1. import numpy as np
2. # Example dataset (exam scores)
3. scores = np.array([78, 85, 92, 88, 95, 80, 91, 84, 89, 87]
    )
4. # Number of bootstrap iterations
5. # The bootstrapping process is repeated 10,000 times (1
    0,000 iterations is somewhat arbitrary).
6. # Allowing us to explore the variability of the statistic (m
    ean in this case). And construct confidence intervals.
7. n_iterations = 10_000
8. # Initialize an array to store bootstrapped means
9. bootstrapped_means = np.empty(n_iterations)
10. # Perform bootstrapping
11. for i in range(n_iterations):
12.     bootstrap_sample = np.random.choice(scores, size=le
        n(scores), replace=True)
```

```
13. bootstrapped_means[i] = np.mean(bootstrap_sample)
14. # Calculate the bootstrap means of all bootstrapped sam
    ples from the main exam score data set
15. print(f"Bootstrapped Mean: {np.mean(bootstrapped_mea
    ns):.2f}")
16. # Calculate the 95% confidence interval
17. lower_bound = np.percentile(bootstrapped_means, 2.5)
18. upper_bound = np.percentile(bootstrapped_means, 97.5)
19. print(f"95% Confidence Interval: [{lower_bound:.2f}, {up
    per_bound:.2f}]")
```

Output:

1. Bootstrapped Mean: 86.89
2. 95% Confidence Interval: [83.80, 90.00]

This means that we expect the average exam score in the entire population (from which our sample was drawn) to be around 86.89. We are 95% confident that the true population mean exam score falls within this interval (83.80, 89.90).

The nonparametric methods include **Kernel Density Estimation (KDE)** which is nonparametric way to estimate probability density functions (probability distribution for a random, continuous variable) and is useful for visualizing data distributions. The survival analysis is also a nonparametric method because it focuses on estimating survival probabilities without making strong assumptions about the underlying distribution of event times. Kaplan-Meier estimator is a non-parametric method used to estimate the survival function.

Survival analysis

Survival analysis is a statistical method used to analyze the amount of time it takes for an event of interest to occur (helping to understand the time it takes for an event to

occur). It is also known as time-to-event analysis or duration analysis. Common applications include studying time to death (in medical research), disease recurrence, or other significant events. But not limited to medicine, it can be used in various fields such as finance, engineering and social sciences. For example, imagine a clinical trial for lung cancer patients. Researchers want to study the time until death (survival time) for patients receiving different treatments. Other examples include analyzing time until finding a new job after unemployment, mechanical system failure, bankruptcy of a company, pregnancy & recovery from a disease.

Kaplan-Meier estimator is one of the most widely used and simplest methods of survival analysis. It handles censored data, where some observations are partially observed (e.g., lost to follow-up). Kaplan-Meier estimation includes the following:

- Sort the data by time
- Calculate the proportion of surviving patients at each time point
- Multiply the proportions to get the cumulative survival probability
- Plot the survival curve

For example, imagine that video game players are competing in a battle video game tournament. The goal is to use survival analysis to see which player can stay **alive** (not killed) the longest.

In the context of survival analysis, data censoring is often encountered concept. Sometimes we do not observe the event for the entire study period, which is when censoring comes into play. Censored data is now; the organizer may have to end the game early. In this case, some player may still be alive when the game end whistle blows. We know they survived at least that long, but we do not know exactly how much longer they would have lasted. This is censored

data in survival analysis. Censoring has type right and left. Right-censored data occurs when we know an event has not happened yet, but we do not know exactly when it will happen in the future. Here censored data can have type right-centered and left centered like in above video game competition. Players who were alive in the game when the whistle blew are right-censored. We know that they survived at least that long (until the whistle blew), but their true survival time (how long they would have survived if the game had continued) is unknown. Left censored data is the opposite of right-censored data. It occurs when we know that an event has already happened, but we do not know exactly when it happened in the past.

Tutorial 9.15: To implement the Kaplan-Meier method to estimate the survival function (survival analysis) of a video game player in a battling video game competition, is as follows:

```
1. from lifelines import KaplanMeierFitter
2. import numpy as np
3. import matplotlib.pyplot as plt
4. # Let's create a sample dataset
5. # durations represents the time of the event (e.g., time u  
ntil student is "alive" in game (not tagged))
6. # event_observed is a boolean array that denotes if the e  
vent was observed (True) or censored (False)
7. durations = [24, 18, 30, 12, 36, 15, 8, 42, 21, 6,
8.             10, 27, 33, 5, 19, 45, 28, 9, 39, 14,
9.             22, 7, 48, 31, 17, 20, 40, 25, 3, 37]
10. event_observed = [0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0,
11.                  1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1]
12. # Create an instance of KaplanMeierFitter
13. kmf = KaplanMeierFitter()
14. # Fit the data into the model
```

```
15. kmf.fit(durations, event_observed)
16. # Plot the survival function
17. kmf.plot_survival_function()
18. # Customize plot (optional)
19. plt.xlabel('Time')
20. plt.ylabel('Survival Probability')
21. plt.title('Kaplan-Meier Survival Curve')
22. plt.grid(True)
23. # Save the plot
24. plt.savefig('kaplan_meier_survival.png', dpi=600, bbox_inches='tight')
25. plt.show()
```

Output:

Figure 9.2 and *Figure 9.3* show the probability of survival appears to decrease over time with a steeper decline observed in the time period near 10 to near 40 points. This suggests that patients are more likely to experience the event (possibly death) as time progresses after surgery. The KM_estimate in *Figure 9.2* is survival curve line, this line represents the Kaplan-Meier survival curve, which is estimated survival probability over time. And shaded area is the **Confidence Interval (CI)**. The narrower the CI, the more precise our estimate of the survival curve. If the CI widens at certain points, it indicates greater uncertainty in the survival estimate at those time intervals.

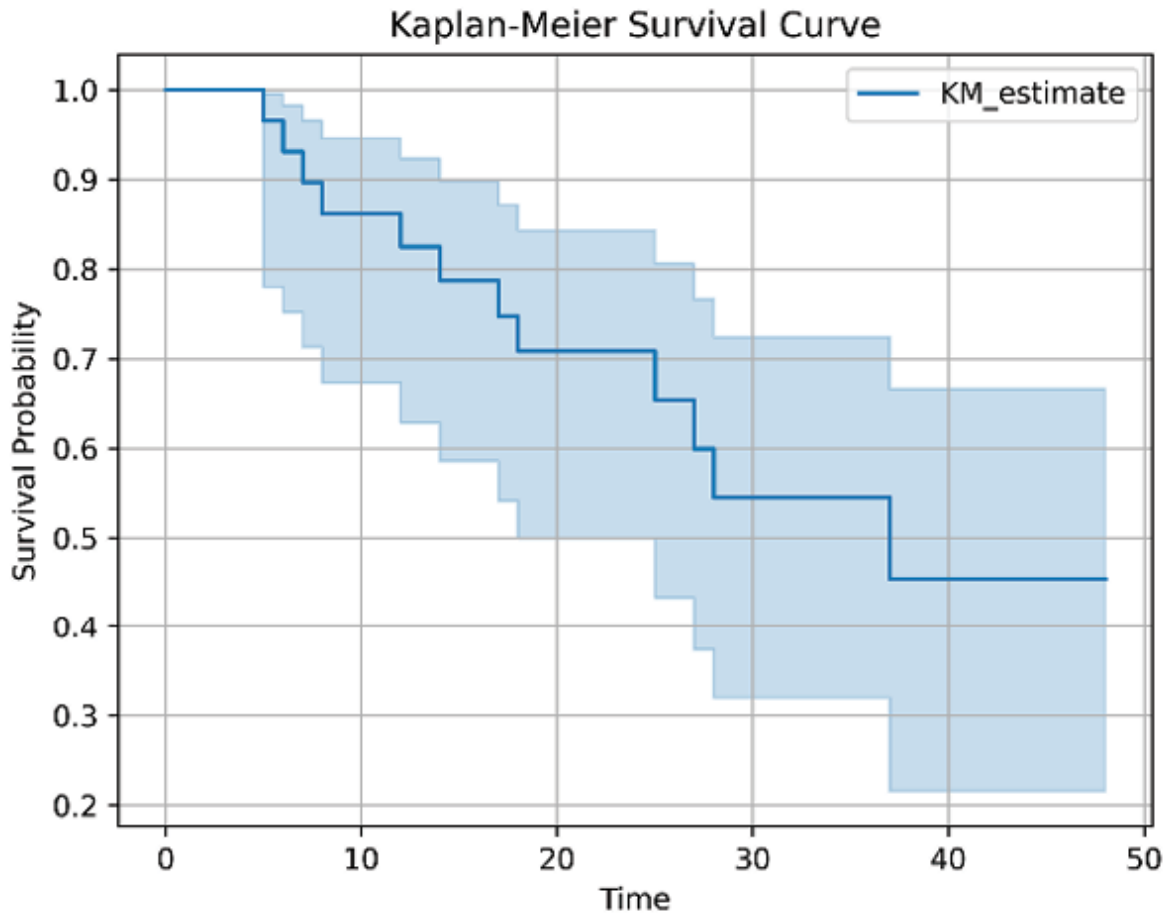


Figure 9.2: Kaplan-Meier curve showing change in probability of survival over time

Let us see another example, suppose we want to estimate the lifespan of patients (time until death) with certain conditions using a sample dataset of 30 patients with their IDs, time of observation (in months) and event status (alive or death). Let us say we are studying patients with heart failure. We will follow them for two years to see if they have a heart attack during that time.

Following is our data set:

- **Patient A:** Has a heart attack after six months (event observed).
- **Patient B:** Still alive after two years (right censored).
- **Patient C:** Drops out of the study after one year (right

censored).

In this case, the way censoring works is as follows:

- **Patient A:** We know the exact time of the event (heart attack).
- **Patient B:** Their data are right-censored because we did not observe the event (heart attack) during the study.
- **Patient C:** Also, right-censored because he dropped out before the end of the study.

Tutorial 9.16: To implement Kaplan-Meier method to estimate survival function (survival analysis) of the patients with a certain condition over time, is as follows:

```
1. import matplotlib.pyplot as plt
2. import pandas as pd
3. # Import Kaplan Meier Fitter from the lifelines library
4. from lifelines import KaplanMeierFitter
5. # Create sample healthcare data (change names as needed)
6. data = pd.DataFrame({
7.     # IDs from 1 to 10
8.     "PatientID": range(1, 31),
9.     # Time is how long a patient was followed up from the
    # start of the study,
10.    # until the end of the study or the occurrence of the event.
11.    "Time": [24, 18, 30, 12, 36, 15, 8, 42, 21, 6,
12.            10, 27, 33, 5, 19, 45, 28, 9, 39, 14,
13.            22, 7, 48, 31, 17, 20, 40, 25, 3, 37],
14.    # Event indicates the event status of patient at the end of observation ,
15.    # whether patient was dead or alive at the end of study period
16.    "Event": ['Alive', 'Death', 'Alive', 'Death', 'Alive', 'Alive'
```

```

    , 'Death', 'Alive', 'Alive', 'Death',
17.     'Alive', 'Death', 'Alive', 'Death', 'Alive', 'Alive', 'D
    eath', 'Alive', 'Alive', 'Death',
18.     'Alive', 'Death', 'Alive', 'Alive', 'Death', 'Alive', 'Al
    ive', 'Death', 'Alive', 'Death']
19. })
20. # Convert Event to boolean (Event indicates occurrence
    of death)
21. data["Event"] = data["Event"] == "Death"
22. # Create Kaplan-
    Meier object (focus on event occurrence)
23. kmf = KaplanMeierFitter()
24. kmf.fit(data["Time"], event_observed=data["Event"])
25. # Estimate the survival probability at different points
26. time_points = range(0, max(data["Time"]) + 1)
27. survival_probability = kmf.survival_function_at_times(ti
    me_points).values
28. # Plot the Kaplan-Meier curve
29. plt.step(time_points, survival_probability, where='post')
30. plt.xlabel('Time (months)')
31. plt.ylabel('Survival Probability')
32. plt.title('Kaplan-Meier Curve for Patient Survival')
33. plt.grid(True)
34. plt.savefig('Survival_Analysis2.png', dpi=600, bbox_inch
    es='tight')
35. plt.show()

```

Output:

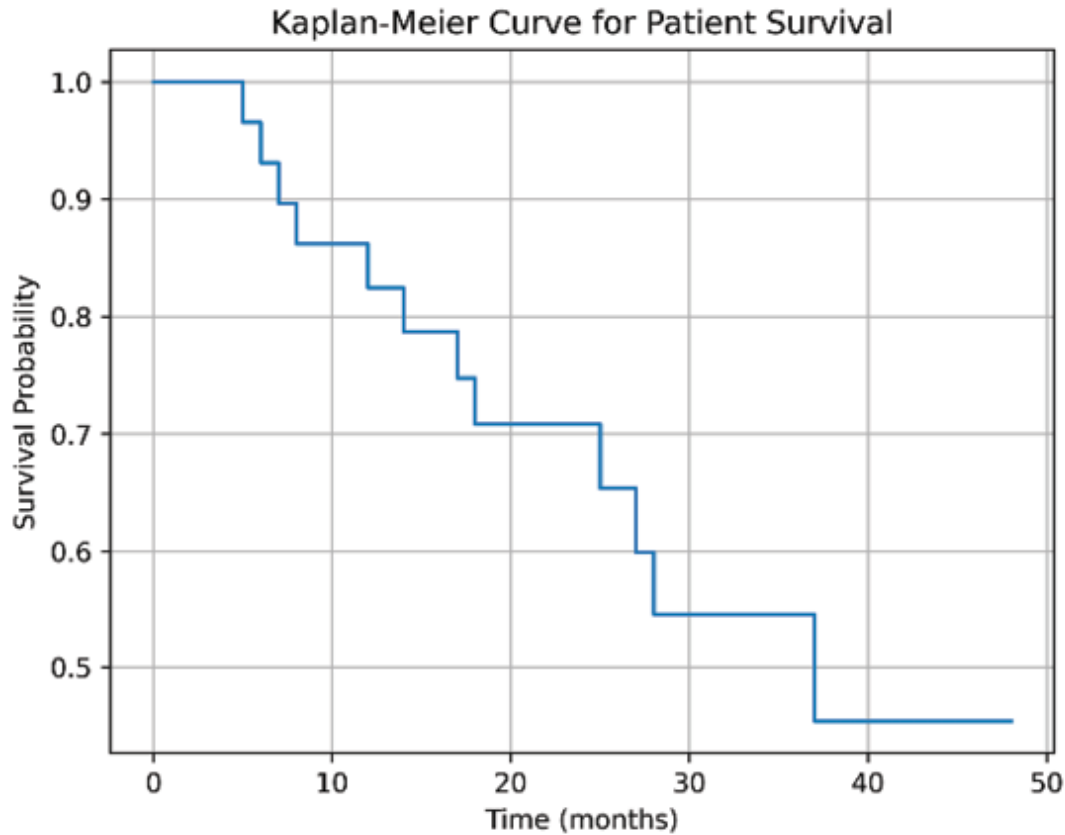


Figure 9.3: Kaplan-Meier curve showing change in probability of survival over time

Following is an example on survival analysis project:

Analyzes and demonstrates patient survival after surgery on a fictitious dataset of patients who have undergone a specific type of surgery. The goal is to understand the factors that affect patient survival time after surgery. Specifically, to analyze the questions. What is the overall survival rate of patients after surgery? How does survival vary with patient age? Is there a significant difference in survival between men and women?

The data includes the following columns:

Columns	Description
patient_id	Unique identifier for each patient
surgery_date	Date of the surgery

event	Indicates whether the event of interest (death) occurred (1) or not (0) during the follow-up period (censored)
survival_time	Time (in days) from surgery to the event (if it occurred) or the end of the follow-up period (if censored).

Table 9.3: *Surgery patient dataset column details*

Tutorial 9.17: To implement Kaplan-Meier survival curve analysis of overall patient survival after surgery, is as follows:

```

1. import pandas as pd
2. from lifelines import KaplanMeierFitter
3. # Create sample data for 30 patients
4. sample_data = {
5.     'patient_id': list(range(101, 131)), # Patient IDs from
        101 to 130
6.     'surgery_date': [
7.         '2020-01-01', '2020-02-15', '2020-03-05', '2020-04-
        10', '2020-05-20',
8.         '2020-06-10', '2020-07-25', '2020-08-15', '2020-09-
        05', '2020-10-20',
9.         '2020-11-10', '2020-12-05', '2021-01-15', '2021-02-
        20', '2021-03-10',
10.        '2021-04-05', '2021-05-20', '2021-06-10', '2021-07-
        25', '2021-08-15',
11.        '2021-09-05', '2021-10-20', '2021-11-10', '2021-12-
        05', '2022-01-15',
12.        '2022-02-20', '2022-03-10', '2022-04-05', '2022-05-
        20', '2022-06-10'],
13.     # 1 for death and 0 for censored
14.     'event': [1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,
        0, 1,
15.              1, 0, 1, 0, 1, 1, 0, 1, 0, 1],
16.     # Survival time in days

```

```

17.     'survival_time': [365, 730, 180, 540, 270, 300, 600, 15
18.                        0, 450, 240,
19.                        330, 720, 210, 480, 270, 660, 150, 390, 21
20.                        0, 570,
21.                        240, 330, 720, 180, 420, 240, 600, 120, 36
22.                        0, 210],
23.     # Gender 0 (Male) and 1 (Female)
24.     'gender': [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
25.               1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
26.     # Age Group 0 - 40 Years is 1 and 41+ Years is 2
27.     'age_group': [1, 2, 1, 2, 1, 2, 2, 1, 2, 1, 1, 2, 1,
28.                  2, 1, 2, 2, 1, 2, 2,
29.                  1, 2, 1, 2, 1, 1, 2, 1, 2, 1]
30. }
31. # Create the dataframe
32. data = pd.DataFrame(sample_data)
33. # Initialize the Kaplan-Meier estimator
34. kmf = KaplanMeierFitter()
35. # Fit the survival data
36. kmf.fit(data['survival_time'], event_observed=data['event'],
37.          label='Overall Survival Analysis')
38. # Plot the survival curve
39. kmf.plot()
40. plt.xlabel("Time (days)")
41. plt.ylabel("Survival Probability")
42. plt.title("Kaplan-Meier Curve for Patient Survival")
43. plt.savefig('example_overall_analysis.png', dpi=600, bbox_inches='tight')
44. plt.show()

```

Output:

In [Figure 9.4](#), survival curve (line) shows the decline in the probability of survival over time, with a steep drop from 100 to 400 days. The widening of the CI, which are the shaded area, indicate greater uncertainty in the survival estimate at those time intervals:

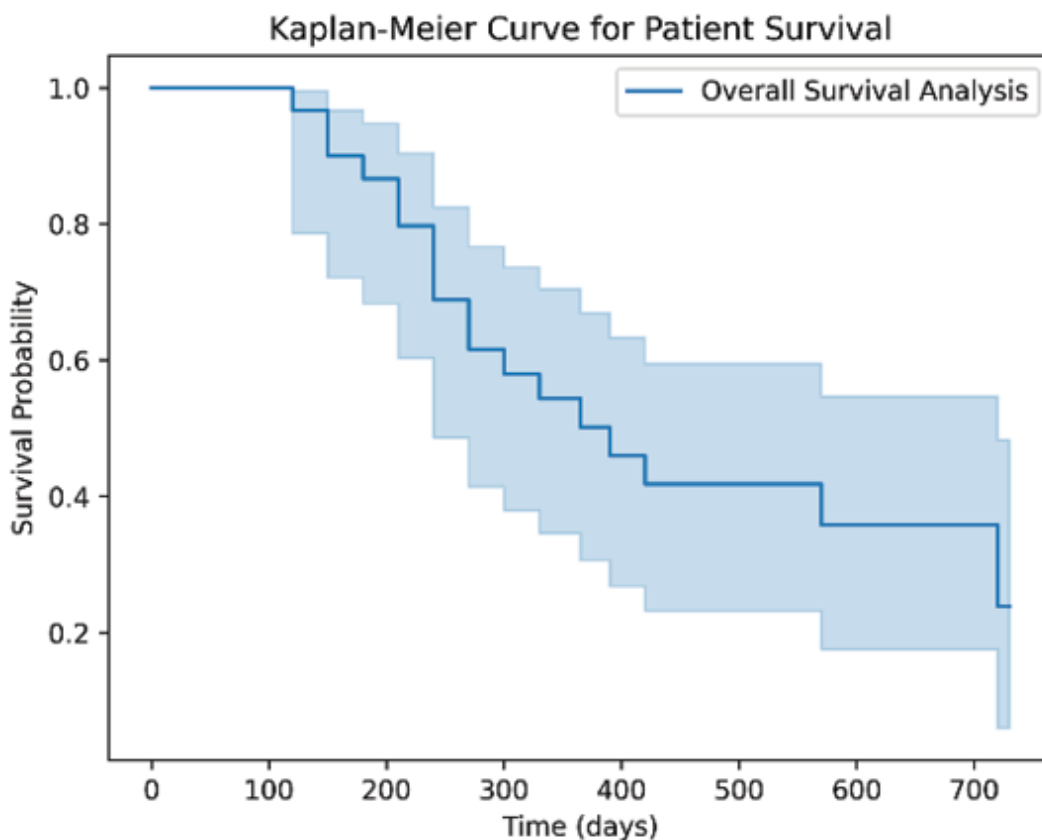


Figure 9.4: Kaplan-Meier curve of overall post-surgery survival

Tutorial 9.18: To continue *Tutorial 9.17*, estimate Kaplan-Meier survival curve analysis for the two age groups after surgery, as follows:

1. # Separate data by gender groups
2. age_group_1 = data[data['age_group'] == 1]
3. # Fit survival data for Gender 1 (Male) age group
4. kmf_age_1 = KaplanMeierFitter()
5. kmf_age_1.fit(age_group_1['survival_time'],
6. event_observed=age_group_1['event'], label='A

```

    ge Group 0 - 40 Years')
7. # Fit survival data for Gender 2 (Female) age group
8. age_group_2 = data[data['age_group'] == 2]
9. kmf_age_2 = KaplanMeierFitter()
10. kmf_age_2.fit(age_group_2['survival_time'],
11.               event_observed=age_group_2['event'], label='A
    ge Group 41+ Years')
12. # Plot the survival curve for both age groups
13. kmf_age_1.plot()
14. kmf_age_2.plot()
15. plt.xlabel("Time (days)")
16. plt.ylabel("Survival Probability")
17. plt.title("Survival Curve by Age Groups")
18. plt.savefig('example_analysis_age_group.png', dpi=600,
    bbox_inches='tight')
19. plt.show()

```

Output:

Figure 9.5 survival curve shows age group 41+ years has lower survival probability then age group 0 to 40:

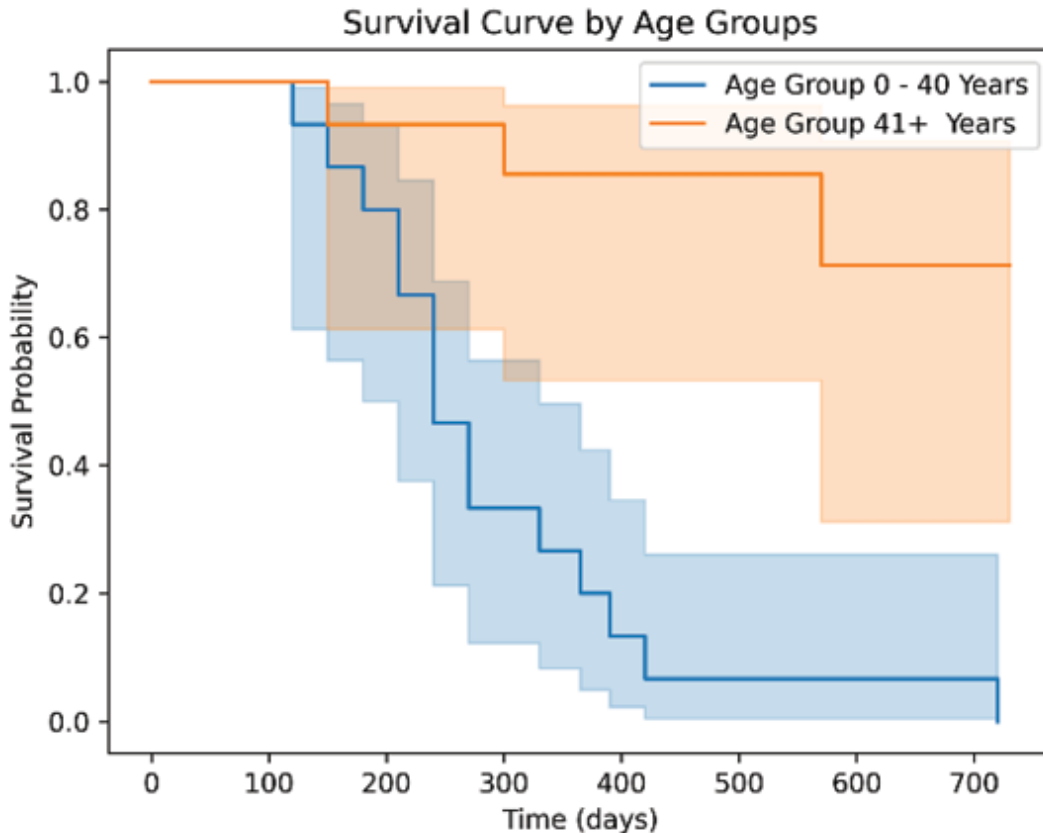


Figure 9.5: Kaplan-Meier curve of post-surgery survival by age group

Tutorial 9.19: To continue *Tutorial 9.17*, estimate Kaplan-Meier survival curve analysis for the two gender groups after surgery, is as follows:

1. # Separate data by gender groups
2. `gender_group_0 = data[data['gender'] == 0]`
3. # Fit survival data for Gender 0 (Male) group
4. `kmf_gender_0 = KaplanMeierFitter()`
5. `kmf_gender_0.fit(gender_group_0['survival_time'],`
6. `event_observed=gender_group_0['event'], label='Gender 0 (Male)')`
7. # Fit survival data for Gender 1 (Female) group
8. `gender_group_1 = data[data['gender'] == 1]`
9. `kmf_gender_1 = KaplanMeierFitter()`
10. `kmf_gender_1.fit(gender_group_1['survival_time'],`

```

11.     event_observed=gender_group_1['event'], label='Gender 1 (Female)')
12. # Plot the survival curve for both age groups
13. kmf_gender_0.plot()
14. kmf_gender_1.plot()
15. plt.xlabel("Time (days)")
16. plt.ylabel("Survival Probability")
17. plt.title("Survival Curve by Gender")
18. plt.savefig('example_analysis_gender_group.png', dpi=600, bbox_inches='tight')
19. plt.show()

```

Output:

Figure 9.6 survival curve shows female has lower survival probability than male:

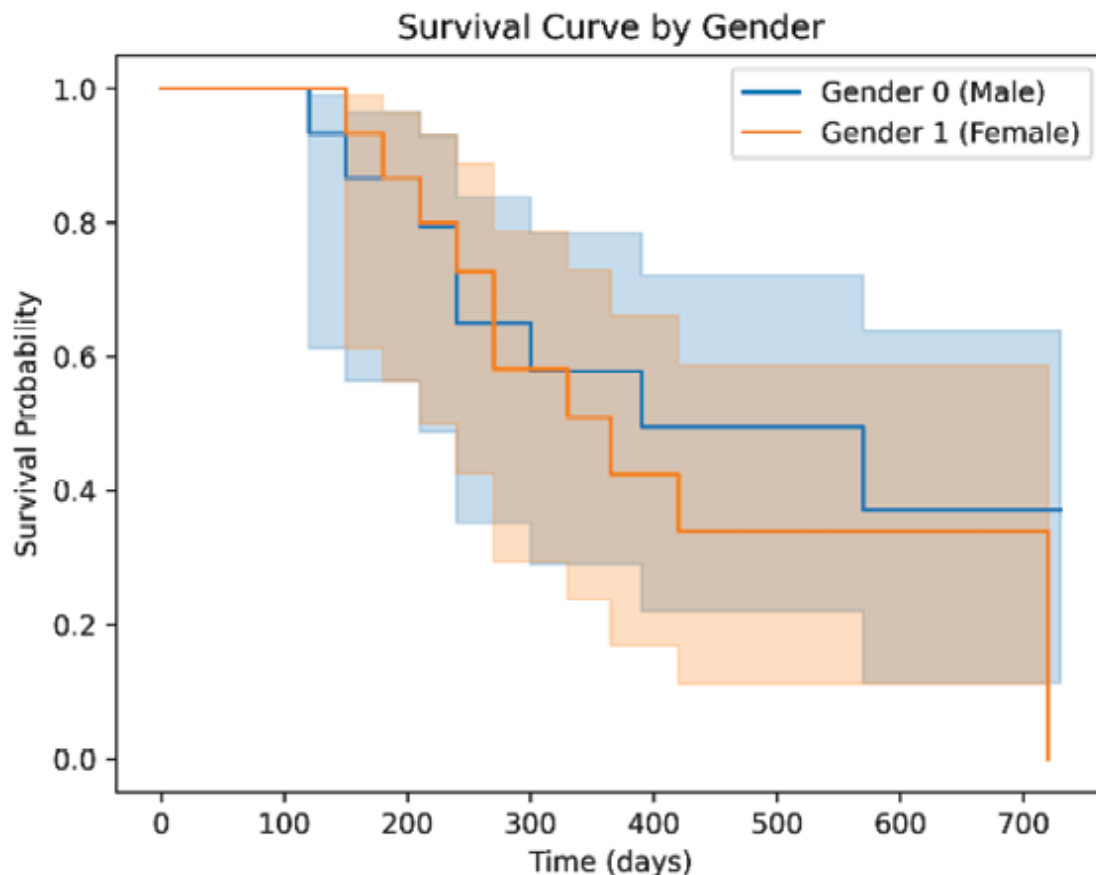


Figure 9.6: Kaplan-Meier curve showing of post-surgery survival by gender

Time series analysis

Time series analysis is a powerful statistical technique used to analyze data collected over time. It helps identify patterns, trends and seasonality in data. Imagine a sequence of data points, such as daily temperatures or monthly sales figures, ordered by time. Time series analysis allows you to make sense of these sequences and potentially predict future values. The data used in time series analysis consists of measurements taken at consistent time intervals. This can be daily, hourly, monthly, or even yearly, depending on the phenomenon being studied. Then the goal is to extract meaningful information from the data. This includes the following techniques:

- **Identifying trends:** Are the values increasing, decreasing, or remaining constant over time?
- **Seasonality:** Are there predictable patterns within a specific time frame, like seasonal fluctuations in sales data?
- **Stationarity:** Does the data have a constant mean and variance over time, or is it constantly changing?

Once you understand the patterns in the data, you can use time series analysis to predict future values. This is critical for applications as diverse as predicting sales trends, stock prices, or weather patterns. For example, to analyze a store's sales data. Imagine you are a retail store manager and you have daily sales data for the past year. Time series analysis can help you do the following:

- **Identify trends:** Are your overall sales increasing or decreasing over the year? Are there significant upward or downward trends?
- **Seasonality:** Do sales show a weekly or monthly pattern? Perhaps sales are higher during holidays or

certain seasons.

- **Forecasting:** Based on the trends and seasonality you identify, you can forecast sales for upcoming periods. This can help you manage inventory, make staffing decisions, and plan marketing campaigns.

By understanding these aspects of your sales data, you can make data-driven decisions to optimize your business strategies. *Tutorial 9.17*, *Tutorial 9.18*, *Tutorial 9.19* show the time series analysis of sales data for trend analysis, seasonality, basic forecasting.

Tutorial 9.20: To implement time series analysis of sales data for trend analysis, is as follows:

```
1. import pandas as pd
2. import matplotlib.pyplot as plt
3. # Sample sales data
4. data = pd.DataFrame({
5.     'date': pd.to_datetime(['2023-01-01', '2023-01-
6.         02', '2023-01-03', '2023-01-04', '2023-01-05',
7.         '2023-01-06', '2023-01-07', '2023-01-
8.         08', '2023-01-09', '2023-01-10',
9.         '2023-02-01', '2023-02-02', '2023-02-
10.        03', '2023-02-04', '2023-02-05',
11.        '2023-02-06', '2023-02-07', '2023-02-
12.        08', '2023-02-09', '2023-02-10',
13.        '2023-03-01', '2023-03-02', '2023-03-
14.        03', '2023-03-04', '2023-03-05',
15.        '2023-03-06', '2023-03-07', '2023-03-
16.        08', '2023-03-09', '2023-03-10',
17.        '2023-04-01', '2023-04-02', '2023-04-
18.        03', '2023-04-04', '2023-04-05',
19.        '2023-04-06', '2023-04-07', '2023-04-
20.        08', '2023-04-09', '2023-04-10']
21. })
```

```
14.     'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
15.               140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
16.               150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
17.               160, 150, 120, 110, 100, 130, 105, 140, 125, 15
18.               0]
19. })
20. # Set the 'date' column as the index
21. data.set_index('date', inplace=True)
22. # Plot the time series data
23. data['sales'].plot(figsize=(12, 6))
24. plt.xlabel('Date')
25. plt.ylabel('Sales')
26. plt.title('Sales Over Time')
27. plt.savefig('trendanalysis.png', dpi=600, bbox_inches='tight')
28. plt.show()
```

Output:

Figure 9.7 shows overall sales increasing over the year, with upward trends:

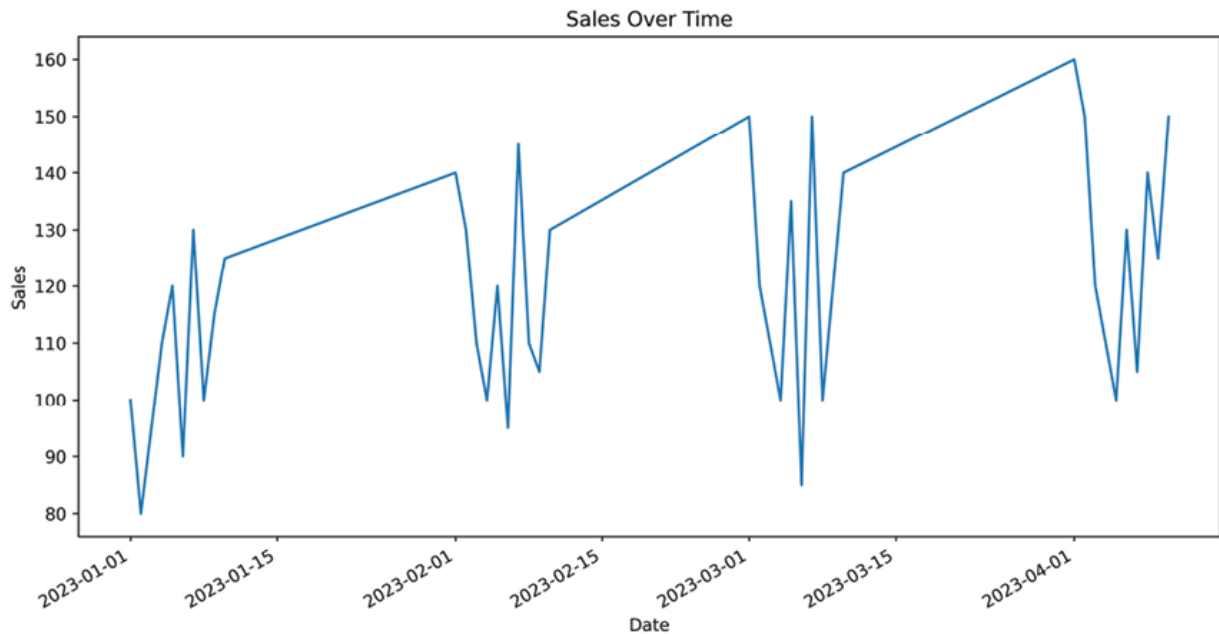


Figure 9.7: Time series analysis to view sales trends throughout the year

Tutorial 9.21: To implement time series analysis of sales data over season or month, to see if season, holidays or festivals affect sales, is as follows:

```
1. import pandas as pd
2. import matplotlib.pyplot as plt
3. # Sample sales data
4. data = pd.DataFrame({
5.     'date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05',
6.                             '2023-01-06', '2023-01-07', '2023-01-08', '2023-01-09', '2023-01-10',
7.                             '2023-02-01', '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',
8.                             '2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09', '2023-02-10',
9.                             '2023-03-01', '2023-03-02', '2023-03-03', '2023-03-04', '2023-03-05',
10.                            '2023-03-06', '2023-03-07', '2023-03-08', '2023-03-09', '2023-03-10',
```

```

11.         '2023-04-01', '2023-04-02', '2023-04-
12.         03', '2023-04-04', '2023-04-05',
13.         '2023-04-06', '2023-04-07', '2023-04-
14.         08', '2023-04-09', '2023-04-10'
15.     ]),
16.     'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
17.              140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
18.              150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
19.              160, 150, 120, 110, 100, 130, 105, 140, 125, 15
20.              0]
21. })
22. # Set the 'date' column as the index
23. data.set_index('date', inplace=True)
24. # Resample data by month (or other relevant period) and calculate mean sales
25. monthly_sales = data.resample('M')['sales'].mean()
26. monthly_sales.plot(figsize=(10, 6))
27. plt.xlabel('Month')
28. plt.ylabel('Average Sales')
29. plt.title('Monthly Average Sales')
30. plt.savefig('seasonality.png', dpi=600, bbox_inches='tight')
31. plt.show()

```

Output:

Figure 9.8 shows overall sales increasing over the year with upward trends:

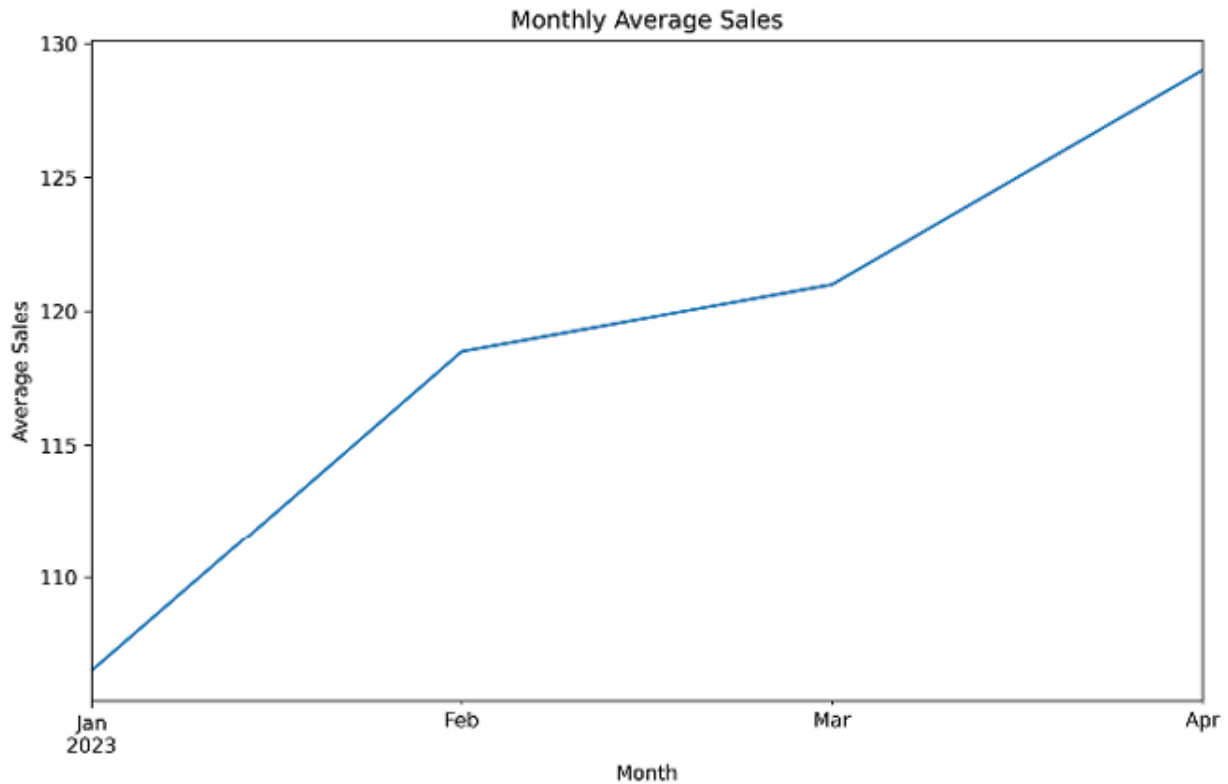


Figure 9.8: Time series analysis of sales by month

Tutorial 9.22: To implement time series analysis of sales data for basic forecasting, is as follows:

1. `import pandas as pd`
2. `import matplotlib.pyplot as plt`
3. `# Sample sales data`
4. `data = pd.DataFrame({`
5. `'date': pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04', '2023-01-05',`
6. `'2023-01-06', '2023-01-07', '2023-01-08', '2023-01-09', '2023-01-10',`
7. `'2023-02-01', '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',`
8. `'2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09', '2023-02-10',`
9. `'2023-03-01', '2023-03-02', '2023-03-`

```

    03', '2023-03-04', '2023-03-05',
10.         '2023-03-06', '2023-03-07', '2023-03-
    08', '2023-03-09', '2023-03-10',
11.         '2023-04-01', '2023-04-02', '2023-04-
    03', '2023-04-04', '2023-04-05',
12.         '2023-04-06', '2023-04-07', '2023-04-
    08', '2023-04-09', '2023-04-10'
13.     ]),
14.     'sales': [100, 80, 95, 110, 120, 90, 130, 100, 115, 125,
15.               140, 130, 110, 100, 120, 95, 145, 110, 105, 130,
16.               150, 120, 110, 100, 135, 85, 150, 100, 120, 140,
17.               160, 150, 120, 110, 100, 130, 105, 140, 125, 15
    0]
18. })
19. # Set the 'date' column as the index
20. data.set_index('date', inplace=True)
21. # Calculate a simple moving average with a window of 7
    days
22. data['rolling_avg_7'] = data['sales'].rolling(window=7).m
    ean()
23. data[['sales', 'rolling_avg_7']].plot(figsize=(12, 6))
24. plt.xlabel('Date')
25. plt.ylabel('Sales')
26. plt.title('Sales with 7-Day Moving Average')
27. plt.savefig('basicforecasting.png', dpi=600, bbox_inches
    ='tight')
28. plt.show()

```

Output:

In [Figure 9.9](#), the solid gray represents the daily sales data. The dashed dark gray line represents the rolling average of sales over a seven-day window. The dashed dark gray line (rolling average) above the solid gray sales line indicates an upward trend in sales over the seven-day period. This indicates that recent sales are higher than the average. The opposite is a downward trend. As you can see, changes in the slope of the rolling average (i.e. sudden spikes or declines)

reveal shifts in sales patterns.

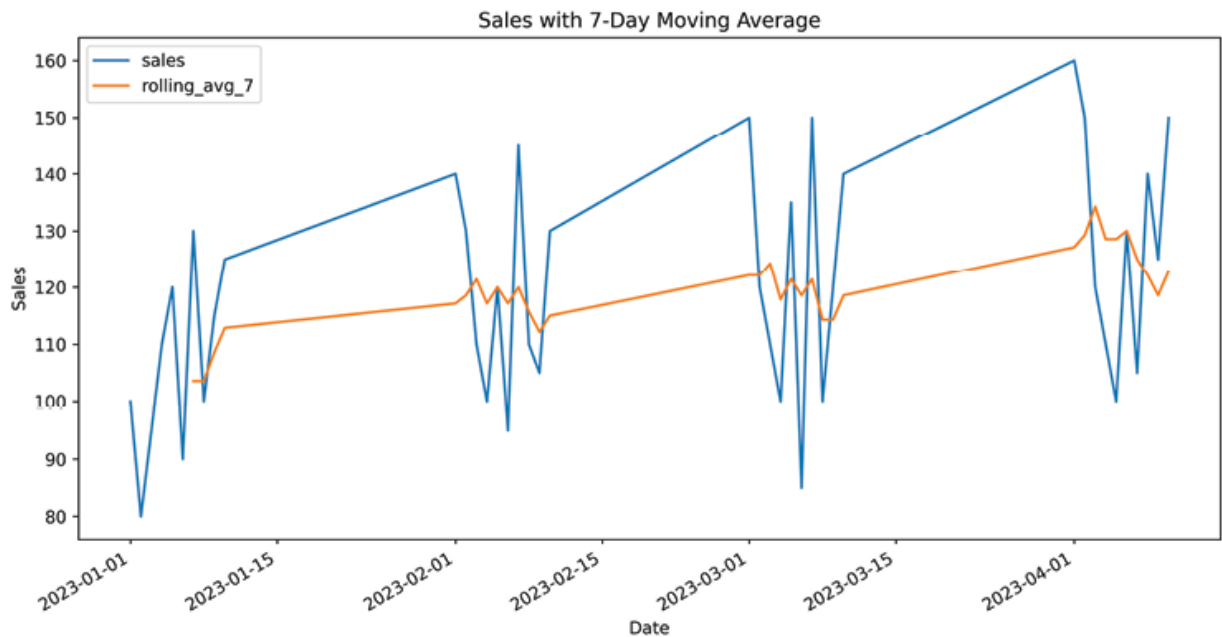


Figure 9.9: Time series analysis of monthly sales to assess the impact of seasons, holidays, and festivals

Conclusion

Finally, this chapter served as an engaging exploration of powerful data analysis techniques like linear algebra, nonparametric statistics, time series analysis and survival analysis. We experienced the elegance of linear algebra, the foundation for maneuvering complex data structures. We embraced the liberating power of nonparametric statistics, which allows us to analyze data without stringent assumptions. We ventured into the realm of time series analysis, revealing the hidden patterns in sequential data. Finally, we delved into survival analysis, a meticulous technique for understanding the time frames associated with the occurrence of events. This chapter, however, serves only as a stepping stone, providing you with the basic knowledge to embark on a deeper exploration. The path to data mastery requires ongoing learning and

experimentation.

Following are some suggested next steps to keep you moving forward: deepen your understanding through practice by tackling real-world problems, master software, packages, and tools and embrace learning. [Chapter 10](#), *Generative AI and Prompt Engineering* ventures into the cutting-edge realm of GPT-4, exploring the exciting potential of prompt engineering for statistics and data science. We will look at how this revolutionary language model can be used to streamline data analysis workflows and unlock new insights from your data.

CHAPTER 10

Generative AI and Prompt Engineering

Introduction

Generative **Artificial Intelligence (AI)** has emerged as one of the most influential and beloved technologies in recent years, particularly since the widespread accessibility of models like ChatGPT to the general public. This powerful technology generates diverse content based on the input it receives, commonly referred to as, prompts. As generative AI continues to evolve, it finds applications across various fields, driving innovation and refinement.

Researchers are actively exploring its capabilities, and there is a growing sense that generative AI is inching closer to achieving **Artificial General Intelligence (AGI)**. AGI represents the holy grail of AI, a system that can understand, learn, and perform tasks across a wide range of domains akin to human intelligence. The pivotal moment in this journey was the introduction of Transformers, a groundbreaking architecture that revolutionized natural language processing. Generative AI, powered by Transformers, has significantly impacted people's lives,

from chatbots and language translation to creative writing and content generation.

In this chapter, we will look into the intricacies of prompt engineering—the art of crafting effective inputs to coax desired outputs from generative models. We will explore techniques, best practices, and real-world examples, equipping readers with a deeper understanding of this fascinating field.

Structure

In this chapter, we will discuss the following topics:

- Generative AI
- Large language model
- Prompt engineering and types of prompts
- Open-ended prompts vs. specific prompts
- Zero-shot, one-shot, and few-shot learning
- Using LLM and generative AI models
- Best practices for building effective prompts
- Industry-specific use cases

Objectives

By the end of this chapter, you would have learned the concept of generative AI, prompt engineering techniques, ways to access generative AI, and many examples of writing prompts.

Generative AI

Generative AI is an artificially intelligent computer program that has a remarkable ability to create new content, and the content is sometimes fresh and original artifacts. It can generate audio, images, text, video, code, and more. It produces new things based on what it has

learned from existing examples.

Now, let us look at how generative AI is built. They leverage powerful foundation models trained on massive datasets and then fine-tuned with complex algorithms for specific creative tasks. Generative AI is based on four major components: the foundation model, training data, fine-tuning, complex mathematics, and computation. Let us look at them in detail as follows:

- Foundation models are the building blocks. Generative AI often relies on foundation models, such as **Large Language Models (LLMs)**. These models are trained on large amounts of text data, learning patterns, context, and grammar.
- Training data is a large reference database of existing examples. Generative AIs learn from training data, which includes everything from books and articles to social media posts, reports, news articles, dissertations, etc. The more diverse the data, the better they become at generating content.
- After initial training, the models undergo fine-tuning. Fine-tuning customizes them for specific tasks. For example, GPT-4 can be fine-tuned to generate conversational responses or to write poetry.
- Building these models involves complex mathematics and requires massive computing power. However, at their core, they are essentially predictive algorithms.

Understanding generative AI

This generative AI takes in the prompt. You provide a prompt (a question, phrase, or topic). Based on the input prompt, AI uses its learned patterns from training data to generate an answer. It does not just regurgitate existing content; it creates something new. The two main approaches used by generative AI are **Generative**

Adversarial Networks (GANs) and autoregressive models:

- **GANs:** Imagine two AI models competing against each other. One, the generator, tries to generate realistic data (images, text, etc.), while the other, the discriminator, tries to distinguish the generated data from real data. Through this continuous competition, the generator learns to produce increasingly realistic output.
- **Autoregressive models:** These models analyze sequences of data, such as sentences or image pixels. They predict the next element in the sequence based on the previous ones. This builds a probabilistic understanding of how the data is structured, allowing the model to generate entirely new sequences that adhere to the learned patterns.

Beyond the foundational models such as GANs and autoregressive models, generative AI also relies on several key mechanisms that enable it to process and generate sophisticated outputs. Behind the scenes, generative AI performs embedding and uses attention mechanism. These two critical components are described as follows:

- **Embedding:** Complex data such as text or images are converted into numerical representations. Each word or pixel is assigned a vector containing its characteristics and relationships to other elements. This allows the model to efficiently process and manipulate the data.
- **Attention mechanisms:** In text-based models, attention allows the AI to focus on specific parts of the input sequence when generating output. Imagine reading a sentence; you pay more attention to relevant words for comprehension. Similarly, the model prioritizes critical elements within the input prompt to create a coherent response.

While understanding generative AI is crucial, it is equally important to keep the human in the loop. Human validation and control are essential to ensure the reliability and ethical use of AI systems. Even though generative AI can produce impressive results, it is not perfect. Human involvement remains essential for validation and control. Validation is when AI-generated content requires human evaluation to ensure accuracy, factuality, and lack of bias. Control is when humans define the training data and prompts that guide the AI's direction and output style.

Large language model

Large Language Model (LLM) is a kind of AI program that excels in understanding and generating human language. It carries out certain functionalities based on trained data, and it consists of multiple building blocks which consists of technologies like deep learning, transformers, and many more.

Following is a brief description of the three aspects:

- **Function:** LLMs can recognize, summarize, translate, predict, and generate text content. They are like super-powered language processors.
- **Training:** They are trained on massive amounts of text data, which is why they are called LLMs. This data can come from books, articles, code, and even conversations.
- **Building blocks:** LLMs are built on a special type of machine learning called deep learning, and more specifically on a neural network architecture called a transformer model.

Now, let us look at how LLMs work. It is the same as generative AI, they take input, encode it and decode it to answer the input. The LLM receives text input, like a sentence or a question. Encoding is a transformer model

within the LLM that analyzes the input, recognizing patterns and relationships between words. Finally, decoding is based on the encoded information; the LLM predicts the most likely output, which could be a translation, a continuation of the sentence, or an answer to a question.

Prompt engineering and types of prompts

Prompt engineering is writing, refining, and optimizing prompts to achieve flawless human-AI interaction. It also entails keeping an updated prompt library and continuously monitoring those prompts. It is like being a teacher for the AI, guiding its learning process to ensure it provides the most accurate and helpful responses. For example: Imagine you are teaching a child to identify animals. You show them a picture of a dog and say, *this is a dog*. The child learns from this and starts recognizing dogs. This is similar to how AI learns from prompts. Now, suppose the child sees a wolf and calls it a dog. This is where refinement comes in. You correct the child by saying, no, that is not a dog; it is a wolf. Similarly, in prompt engineering, we refine and optimize the prompts based on the AI's responses to make the interaction more accurate. Monitoring is like keeping an eye on the child's learning progress. If the child starts calling all four-legged animals' dogs, you know there is a problem. Similarly, prompt engineers continuously monitor the AI's responses to ensure it is learning correctly. Maintaining an up-to-date prompt library is like updating the child's knowledge as they grow. As the child gets older, you might start teaching them about different breeds of dogs.

Similarly, prompt engineers update the AI's prompts as it learns and grows, ensuring it can handle more complex tasks and inquiries. Prompt design is both an art and a

science. Experimenting, iterating, and refining your approach to unlock the full potential of AI-generated responses across applications is a secret to prompting. Whether you are a seasoned developer or a curious beginner, understanding prompt types is crucial for generating insightful and relevant responses. Now, let us understand the types of prompts.

Open-ended prompts versus specific prompts

Open-ended prompts are broad and flexible, giving the AI system room to generate diverse content. They allow for creativity and exploration, encourage imaginative responses without strict constraints. For example, *write a short story about a mysterious mountain* is an open-ended prompt where the AI system is free to use creativity as there are no constraints.

On the other hand, **specific prompts** provide clear instructions and focus on a specific task or topic. They guide the AI to a specific result. They are useful when you need precise answers or targeted information. For example, *summarize the key findings of the research paper titled Climate Change Impact on Arctic Ecosystems*. The choice of open-ended and specific prompts depends on the desired outcome and objective of the task. However, clear and specific prompts provide more accurate and relevant content. [Table 10.1](#) provides the domain in which each of the above prompt types will be useful, along with the examples of each:

Types	Useful domain	Examples
Open-Ended	Creative writing	• Write a short story about an unlikely friendship between a human and an AI in a futuristic city. • Imagine a world where gravity works differently. Describe the daily life of someone living in this world.
	Brainstorming	• Generate ideas for a new sci-fi movie plot involving time travel. • List five innovative uses for drones beyond photography and surveillance.
	Exploration and imagination	• Describe an alien species with unique physical features and cultural practices. • Write a poem inspired by the colors of a sunset over a tranquil lake.
	Character development	• Create a detailed backstory for a rogue archaeologist who hunts ancient artifacts. • Introduce a quirky sidekick character who communicates only through riddles.
	Philosophical reflection	• Explore the concept of free will versus determinism in a thought-provoking essay. • Discuss the ethical implications of AI achieving consciousness.

Types	Useful domain	Examples
Specific Prompts	Summarization	• Provide a concise summary of the American Civil War in three sentences. • Summarize the key findings from the World health statistics 2023 report.
	Technical writing	• Write step-by-step instructions for setting up a home network router. • Create a user manual for a smartphone camera app, including screenshots.
	Comparisons and contrasts	• Compare and contrast the advantages of electric cars versus traditional gasoline cars. • Analyze the differences between classical music and contemporary pop music.
	Problem-solving	• Outline a Python code snippet to calculate the Fibonacci sequence. • Suggest strategies for reducing plastic waste in a coastal city.
	Persuasive writing	• Compose an argumentative essay advocating for stricter regulations on social media privacy. • Write a letter to the editor supporting the implementation of renewable energy policies.

Table 10.1: Prompts types, with their use and examples

Zero-shot, one-shot, and few-shot learning

Prompting techniques play a key role in shaping the behavior of LLMs. They allow prompts to be designed and optimized for better results. These techniques are essential for eliciting specific responses from generative AI models or LLMs. Zero-shot, one-shot, and few-shot prompting are common prompting techniques. Besides that, the chain of thought, self-consistency, generated knowledge prompting, and retrieval augmented generation are additional strategies.

Zero-shot

Used when no labeled data is available for a specific task. It is useful, as it enables models to generalize beyond their training data by learning from related information. For example, recognizing new classes without prior examples, for example, identifying exotic animals based on textual descriptions). Now, let us look at a few more examples as follows:

Example 1:

- **Prompt:** Translate the following English sentence to French: The sun is shining.
- **Technique:** Zero-shot prompting allows the model to perform a task without specific training. The model can translate English to French even though the exact sentence was not seen during training.

Example 2:

- **Prompt:** Summarize the key points from the article about climate change.
- **Technique:** Zero-shot summarization. The model generates a summary without being explicitly trained on the specific article.

One-shot

It is used to deal with limited labeled data and is ideal for scenarios where many labeled examples are scarce. For example, training models with only one example per class, for example, recognition of rare species or ancient scripts. In one-shot learning, a model is expected to understand and generate a response or task (such as writing poem) based on a single prompt without needing additional examples or instructions. Now, let us look at a few examples as follows:

Example 1:

- **Prompt:** Write a short poem about the moon.
- **Technique:** A single input prompt is given to generate content.

Example 2:

- **Prompt:** Describe a serene lakeside scene.
- **Technique:** Model is given one-shot description (i.e, a vivid scene) in the given prompt.

Few-shot

The purpose of few-shot learning is that it can learn from very few labeled samples. Hence, it is useful to bridge the gap between one-shot and traditional supervised learning. For example, it addresses tasks such as medical diagnosis with minimal patient data or personalized recommendations. Now, let us look at a few examples:

Example 1:

- **Prompt:** Continue the story: Once upon a time, in a forgotten forest
- **Technique:** Few-shot prompting allows the model to build on a partial narrative.

Example 2:

- **Prompt:** List three benefits of meditation.
- **Technique :** Few-shot information retrieval. The model provides relevant points based on limited context.

Chain-of-thought

Chain-of-Thought (CoT) encourages models to maintain coherent thought processes across multiple responses. It is useful for generating longer, contextually connected outputs. For example, crafting multi-turn dialogues or essay-like responses. Now, let us look at a few examples as follows:

Example 1:

- **Prompt:** Write a paragraph about the changing seasons.
- **Technique:** Chain of thought involves generating coherent content by building upon previous sentences. Here, writing about the change in the season involves keeping the past season in mind.

Example 2:

- **Prompt:** Discuss the impact of technology on human relationships.
- **Technique:** Chain of thought essay. The model elaborates on the topic step by step.

Self-consistency

Self-consistency prompting is a technique used to ensure that a model's responses are coherent and consistent with its previous answers. This method plays a crucial role in preventing the generation of contradictory or nonsensical information, especially in tasks that require logical reasoning or factual accuracy. The goal is to make sure that the model's output follows a clear line of thought and maintains internal harmony. For instance, when performing fact-checking or engaging in complex reasoning, it's vital that the model doesn't contradict itself within a single response or across multiple responses. By applying self-consistency prompting, the model is guided to maintain logical coherence, ensuring that all parts of the response are in agreement and that the conclusions drawn are based on accurate and consistent information. This is particularly important in scenarios where accuracy and reliability are key, such as in medical diagnostics, legal assessments, or research. Now, let us look at a few examples as follows:

Example 1:

- **Prompt:** Create a fictional character named Gita and describe her personality.

- **Technique:** Self-consistency will ensure coherence within the generated content.

Example 2:

- **Prompt:** Write a dialogue between two friends discussing their dreams.
- **Technique:** Self-consistent conversation. The model has to maintain character consistency throughout.

Generated knowledge

Generated knowledge prompting encourages models to generate novel information. It is useful for creative writing, brainstorming, or expanding existing knowledge. For example, crafting imaginative stories, inventing fictional worlds, or suggesting innovative ideas. Since this is one of the areas of keen interest for most researchers, efforts are being put to make it better for generating knowledge. Now, let us look at a few examples as follows:

Example 1:

- **Prompt:** Explain the concept of quantum entanglement.
- **Technique:** Generated knowledge provides accurate information.

Example 2:

- **Prompt:** Describe the process of photosynthesis.
- **Technique:** Generated accurate scientific explanation.

Retrieval augmented generation

Retrieval augmented generation (RAG) combines generative capabilities with retrieval-based approaches. It enhances content by pulling relevant information from external sources. For example, improving the quality of responses by incorporating factual details from existing knowledge bases. Now, let us look at a few examples as

follows:

Example 1: Generating friendly ML paper titles

- **Prompt:** Create a concise title for a machine learning paper discussing transfer learning.
- **Technique:** RAG combines an information retrieval component with a text generator model.
- **Process:**
 - RAG retrieves relevant documents related to transfer learning (e.g., research papers, blog posts).
 - These documents are concatenated as context with the original input prompt.
 - The text generator produces the final output.

Example 2: Answering complex questions with external knowledge

- **Prompt:** Explain the concept of quantum entanglement.
- **Technique:** RAG leverages external sources for example, Wikipedia to ensure factual consistency.
- **Process:**
 - RAG retrieves relevant Wikipedia articles on quantum entanglement.
 - The retrieved content is combined with the original prompt.
 - The model generates an accurate explanation.

Using LLM and generative AI models

Using generative AI and LLM has become very simple and easy. Here, we give a quick overview of using them with Python in Jupyter Notebook. Also, by pointing to some of the existing web application and their **Uniform Resource**

Locator (URLs). Now, there are many LLM, but the GPT-4 seems dominant; besides that, *Google's Gemini*, *Meta's Llama 3*, *X's Grok-1.5*, open-source models in hugging face exist, and growth continues. Now let us have a look at the following:

Setting up GPT-4 in Python using the OpenAI API

Follow these steps to setup GPT-4 in Python using the OpenAI API:

1. Create an OpenAI developer account
 - a. Before understanding the technical details, you need to create an account with OpenAI. Follow these steps:
 - i. Go to the API signup page.
 - ii. Sign up with your email address and phone number.
 - iii. Once registered, go to the API keys page.
 - iv. Create a new secret key (make sure to keep it secure).
 - v. Add your debit or credit card details on the **Payment Methods** page.
2. Install required libraries
 - a. To use GPT-4 via the API, you will need to install the OpenAI library. Open your command prompt or terminal and run the following command:
 1. `pip install openai`
3. Securely store your API keys
 - a. Keep your secret API key confidential. One of the easy to avoid hardcoding Open-AI API key directly in the code, is by using **dotenv**.
 - b. Install the **python-dotenv** package:
 1. `pip install python-dotenv`
 - c. Create a **.env** file in the project directory and add an

API key on it:

1. `OPENAI_API_KEY=your_actual_api_key_here`

- d. In your Python script or Jupyter Notebook, load the API key from the `.env` file.

4. Start generating content with GPT-4.

Tutorial 10.1: To use and access GPT-4 using Open-AI API key, install **openai** and **dotenv** (which is to hide API key from Jupyter Notebook) and then follow the code:

```
1. # Import libraries
2. import os
3. from dotenv import load_dotenv
4. import openai
5. from IPython.display import display, Markdown
6. # Load environment variables from .env
7. load_dotenv()
8.
9. # Get the API key
10. openai.api_key = os.getenv("OPENAI_API_KEY")
11. # Create completion using GPT-4
12. completion = openai.ChatCompletion.create(
13.     model="gpt-4",
14.     messages=[
15.         {"role": "user", "content": "What is artificial intellig
16.         ence?"}
17.     ]
18. )
19. # Print the response
20. print(completion.choices[0].message['content'])
```

Most of you might have used the GPT-3, which can be accessed free of cost. GPT-4 can be accessed by paying from <https://chat.openai.com/>. It can also be used with

Microsoft Copilot or notebook (<https://copilot.microsoft.com/>).

Similarly, *Gemini* from *Google* can be accessed using (<https://gemini.google.com/app>). Also, from the hugging face platform, many open-source models can be accessed and used.

Tutorial 10.2: To use open-source model **google/flan-t5-large** from Hugging face for text generation, first install **langchain**, **huggingface_hub**, **transformers**, and then type the following code:

```
1. from langchain import PromptTemplate, HuggingFaceHub, LLMChain
2. import os
3. os.environ['HUGGINGFACEHUB_API_TOKEN'] = 'REPLACE_WITH_HUGGINGFACE_TOKEN_KEY'
4. prompt = PromptTemplate(input_variables=[
5.     'Domain'], template='What are the uses of artificial intelligence in {Domain} ?')
6. chain = LLMChain(llm=HuggingFaceHub(
7.     repo_id='google/flan-t5-large'), prompt=prompt)
8. print(chain.run(Domain='healthcare'))
```

Running above code the **'google/flan-t5-large'** model will give a reply to the question. In this way any model can be accessed and used from HuggingFace platform.

Best practices for building effective prompts

To write an effective prompt, the task for which the prompt has to be written, the context, example, persona, format, and tone of the prompt are important. Of these, the task is a must, and giving context to it is kind of mandatory. Having examples is important. Besides these, having persona, format, and tone are good to have in a prompt.

Also, do not be afraid to ask for creative results to solve technically savvy problems to LLMs. They are creative and can produce poems, stories, and even jokes. They can also process mathematical problems; you can ask them to compute mathematical, logical problems.

For example, in a sentence: **I am a tenth grader student, define what artificial intelligence is to me?** The first part of the sentence, **I am a tenth grader student** is context and defines artificial intelligence is the task. Basically, to build effective prompts for LLMs or generative AI, it is good to know and keep the following in mind:

- **Task:** Task is compulsory to include in the prompt, use an action verb to specify the task. Write, Create, Give, Analyze, Calculate, etc. are the action verbs.
 - For example, in a prompt, **Write a short essay on the impact of climate change.**
 - Here, **write** is the action verb, and the task is to write an essay. The prompt can have any number of tasks like it can have a single task or multiple tasks in one.
 - For example, **Calculate the area of a circle with a radius of 5 units is a single task prompt.**
 - Similarly, **Calculate the area of a circle with a radius of 5 units and the circumference of the same circle** is a multitask prompt.
- **Context:** Providing context includes providing background information, setting the environment, and mentioning what the desired success or outcome looks like. For example, **As a high school student, write a letter to your local government official about improving public transportation in the city.** Here, the context is that the user is a high school student

writing to a government official.

- **Clarity and specificity:** Ensure the prompt is clear and unambiguous. Being specific in your prompt will guide the LLM toward the desired output. For example, instead of saying **Write about a historical event**, you could say **Write a detailed account of the Battle of Kurukshetra**.
- **Iterative refinement:** It is often beneficial to refine the prompts iteratively based on the model's responses. For example, if the model's responses are too broad, the prompt can be made more specific.
- **Exemplars, persona, format and tone matters:**
 - Exemplars provide examples to teach the LLM to answer correctly through example or suggestions. For example, **Write a poem about spring**. For instance, you could talk about blooming flowers, longer days, or the feeling of warmth in the air.
 - Persona prompts make the LLM think of someone you wish to have for the task you are facing. For example, **Imagine you are a tour guide. Describe the main attractions of New Delhi**.
 - Format is how you want your output to be structured, and the tone is the mood or attitude conveyed in the response. For example, **Write a formal email to your professor asking for an appointment. The tone should be respectful and professional**.

Industry-specific use cases

Generative AI and LLMs have found applications in many fields and domains. However, their true power lies in fine-tuning- tailoring these models for specific use cases or industries. By customizing prompts and constraints,

organizations can harness the power of LLMs to address unique challenges in different fields as described here:

- **Engineering:** In engineering, code generation, optimizing design, and solving logical problems are a few examples of using generative AI. For example, software engineers can use prompts to automate repetitive coding tasks, freeing time to solve complex problems, etc. Such as, writing a Python function to calculate the area of a circle. Mathematicians and programmers can use it to solve complex mathematical problems like solving and implementing a quadratic equation of the form $ax^2 + bx + c = 0$. It can also be used to create variations of a design based on specified parameters. An engineer could provide a prompt outlining desired material properties, weight constraints, and functionality for a bridge component. The AI would then generate multiple design options that meet these criteria, accelerating the design exploration process.
- **Healthcare:**
 - **Personalized patient education:** Imagine an AI that creates educational materials tailored to a patient's specific condition and literacy level. A physician could ask the AI to create a video explaining diabetes management in a language appropriate for an elderly patient. This can improve patient understanding and medication adherence.
 - **Drug discovery:** Developing new drugs is a long and expensive process. Generative AI can be directed to design new drug molecules with specific properties to target a particular disease. This can accelerate drug discovery and potentially lead to breakthroughs in treatment.
 - **Mental health virtual assistants:** AI-powered

chatbots can provide basic mental health support and emotional monitoring. Prompts can guide the AI to provide appropriate responses based on a user's input, providing 24/7 support and potentially reducing the burden on human therapists.

- **Education:**

- **Personalized learning materials:** Generative AI can create customized practice problems, quizzes, or educational games based on a student's strengths and weaknesses. A teacher could ask the AI to generate math practice problems of increasing difficulty for a student struggling with basic algebra.
- **Simulate real-world scenarios:** Training future doctors, nurses, or even firefighters require exposure to a variety of situations. Generative AI can be instructed to create realistic simulations of medical emergencies, allowing students to practice decision-making in a safe environment.
- **Create accessible learning materials:** Generative AI can be asked to create alternative formats for educational content, such as converting text into audio descriptions for visually impaired students or generating sign language translations for lectures.

- **Manufacturing:**

- **Create bills of materials (BOMs):** Creating BOMs which list all the components needed for a product, can be a tedious task. Generative AI, prompted by a product design or 3D model, can automatically generate a detailed BOM, improving efficiency and reducing errors.
- **Predictive maintenance:** By analyzing sensor

data and historical maintenance records, generative AI models can be asked to predict when equipment might fail. This enables proactive maintenance, minimizing downtime and lost production.

- **Content creation:**

- **Generate product descriptions:** E-commerce businesses can use generative AI to create unique and informative product descriptions based on product specifications and customer data. A prompt could include details such as product features, target audience, and desired tone of voice.
- **Write marketing copy:** Creating catchy headlines, social media posts, or email marketing content can be time-consuming. Generative AI can be prompted with a product or service and generate multiple creative copy options, allowing marketers to choose the most effective.

Conclusion

The field of generative AI, driven by LLMs, is at the forefront of technological innovation. Its impact is reverberating across multiple domains, simplifying tasks, and enhancing human productivity. From chatbots that engage in natural conversations to content generation that sparks creativity, generative AI has become an indispensable ally. However, this journey is not without its challenges. The occasional hallucination where models produce nonsensical results, the need for alignment with human values, and ethical considerations all demand our attention. These hurdles are stepping stones to progress. Imagine a future where generative AI seamlessly assists us, a friendly collaborator that creates personalized emails, generates creative writing, and solves complex problems. It

is more than a tool; it is a companion on our digital journey. This chapter serves as a starting point- an invitation to explore further. Go deeper, experiment, and shape the future. Curiosity will be your guide as you navigate this ever-evolving landscape. Generative AI awaits your ingenuity, and together, we will create harmonious technology that serves humanity.

In final *Chapter 11, Data Science in Action: Real-World Statistical Applications*, we explore two key projects. The first applies data science to banking data, revealing insights that inform financial decisions. The second focuses on health data, using statistical analysis to enhance patient care and outcomes. These real-world applications will demonstrate how data science is transforming industries and improving lives.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Real World Statistical Applications

Introduction

As we reach the climax of the book, this final chapter serves as a practical bridge between theoretical knowledge and real-world applications. Throughout this book, we have moved from the basics of statistical concepts to advanced techniques. In this chapter, we want to solidify your understanding by applying the principles you have learned to real-world projects. In this chapter, we will delve into two comprehensive case studies—one focused on banking data and the other on healthcare data. These projects are designed not only to reinforce the concepts covered in earlier chapters but also to challenge you to use your analytical skills to solve complex problems and generate actionable insights. By implementing the statistical methods and data science techniques discussed in this book, you will see how data visualization, exploratory analysis, inferential statistics and machine learning come together to solve real-world problems. This hands-on approach will help you appreciate the power of statistics in data science and prepare you to apply these skills in your future endeavors,

whether in academia or industry. The final chapter puts theory into practice, ensuring that you leave with both the knowledge and the confidence to tackle statistical data science projects on your own.

Structure

In this chapter, we will discuss the following topics:

- Project I: Implementing data science and statistical analysis on banking data
- Project II: Implementing data science and statistical analysis on health data

Objectives

This chapter aims to demonstrate the practical implementation of data science and statistical concepts using real-world synthetic banking and health data generated for this book only, as a case study. By analyzing these datasets, we will illustrate how to derive meaningful insights and make informed decisions based on statistical inference.

Project I: Implementing data science and statistical analysis on banking data

Project I harnesses the power of synthetic banking data to explore and analyze customer behaviors and credit risk profiles in the banking sector. The generated synthetic dataset contains detailed information on customer demographics, account types, transaction details, loan and credit cards, as shown in *Figure 11.1*:

	CustomerID	Age	Gender	Occupation	EducationLevel	MaritalStatus
0	abc7b464-6e65-4303-9e51-eeaf4f3d1ca2	46	Other	Production designer, theatre/television/film	Master	Single
1	41f7f53a-b53f-4213-906c-98e24ec65c6c	37	Male	Radio producer	High School	Widowed

	CustomerID	AccountNumber	AccountType	DateAccountOpened	AccountStatus
0	abc7b464-6e65-4303-9e51-eeaf4f3d1ca2	270749932	Savings	2009-06-08	Active
1	41f7f53a-b53f-4213-906c-98e24ec65c6c	6989827612	Savings	2012-11-25	Active

	TransactionID	AccountNumber	DateofTransaction	TypeofTransaction	Amount	TransactionDescription
0	73018798	270749932	2022-08-13	Deposit	2984.87	And drop less seat strategy eye.
1	41549331	6989827612	2022-12-25	Transfer	5601.25	Strategy instead study job list.

	LoanID	AccountNumber	LoanAmount	LoanStartDate	LoanType	InterestRate	LoanTerm	MonthlyRepaymentAmount
0	6342289	270749932	14984	2010-04-10	Auto	7.15	30	1669
1	46972378	6989827612	23867	2021-02-05	Personal	3.44	30	1749

	CardNumber	AccountNumber	CreditLimit	CardType	ExpiryDate	CVV
0	213131988105207	270749932	3083	MasterCard	2026-05-09	036
1	4810457564935559469	6989827612	8873	MasterCard	2025-12-09	014

Figure 11.1: Structure of the synthetic health data

Then, using the above-mentioned synthetic health data, follow these steps:

1. **Data loading and exploratory analysis:** The initial phase involves loading the synthetic banking data followed by exploratory analysis. We employ descriptive statistics and visualization techniques to understand the underlying patterns and distributions within the data, here we show distribution of customer by ages, and account types.
2. **Statistical testing:** The next step focuses on statistical analysis to explore the relationships between variables. Here we analyze relationship between customer's education level and their chosen account types. This involves assessing whether significant differences exist in the type of accounts held, based on education levels.
3. **Credit card risk analysis:** By leveraging customer data attributes like education level, marital status, account type, loan type, interest rate, and credit limit, we categorize customers into risk groups: high, medium, and low. This segmentation is based on predefined criteria that consider both financial behavior and demographic factors.

4. **Predictive modelling:** The core analytical task involves developing a predictive model to classify customers into risk categories (high, medium, low) for issuing credit cards. This model helps in understanding and predicting customer risk profiles based on their banking and demographics information, to decide if it is right decision to issue the credit card.
5. **Model deployment for user input prediction:** In the final step, the trained model is deployed as a tool that accepts user inputs via the command line for attributes such as education level, marital status, account type, loan type, interest rate, and credit limit. This allows for real-time risk assessment and prediction on potential credit card issuance.

This project not only enhances our understanding of customer behaviors and risk but also aids in strategic decision-making for credit issuance based on robust data-driven insights.

Part 1: Exploratory data analysis

Here, we will examine the data to understand distributions and relationships. The code snippet for the **Exploratory Data Analysis (EDA)** as follows:

```
1. # Load data from CSV files
2. customers = pd.read_csv(
3.     '/workspaces/ImplementingStatisticsWithPython/note
   books/chapter11project/banking/data/customers.csv')
4. accounts = pd.read_csv(
5.     '/workspaces/ImplementingStatisticsWithPython/note
   books/chapter11project/banking/data/accounts.csv')
6. # Plotting the age distribution of customers
7. plt.figure(figsize=(10, 6))
8. plt.hist(customers['Age'], bins=30, density=True, alpha=
   0.5, color='blue')
```

```
9. plt.title('Distribution of Customer Ages')
10. plt.xlabel('Age')
11. plt.ylabel('Frequency')
12. plt.savefig('age_distribution.jpg', dpi=300, bbox_inches=
    'tight')
13. plt.show()
14. # Analyzing account types by plotting distribution in bar
    plot
15. account_types = accounts['AccountType'].value_counts()
16. plt.figure(figsize=(10, 6))
17. plt.bar(account_types.index, account_types.values, alpha
    =0.5, color='blue')
18. plt.title('Distribution of Account Types')
19. plt.xlabel('Account Type')
20. plt.ylabel('Number of Accounts')
21. plt.savefig('account_type_distribution.jpg', dpi=300, bbo
    x_inches='tight')
22. plt.show()
```

Following is the output of *Project I*, part 1:

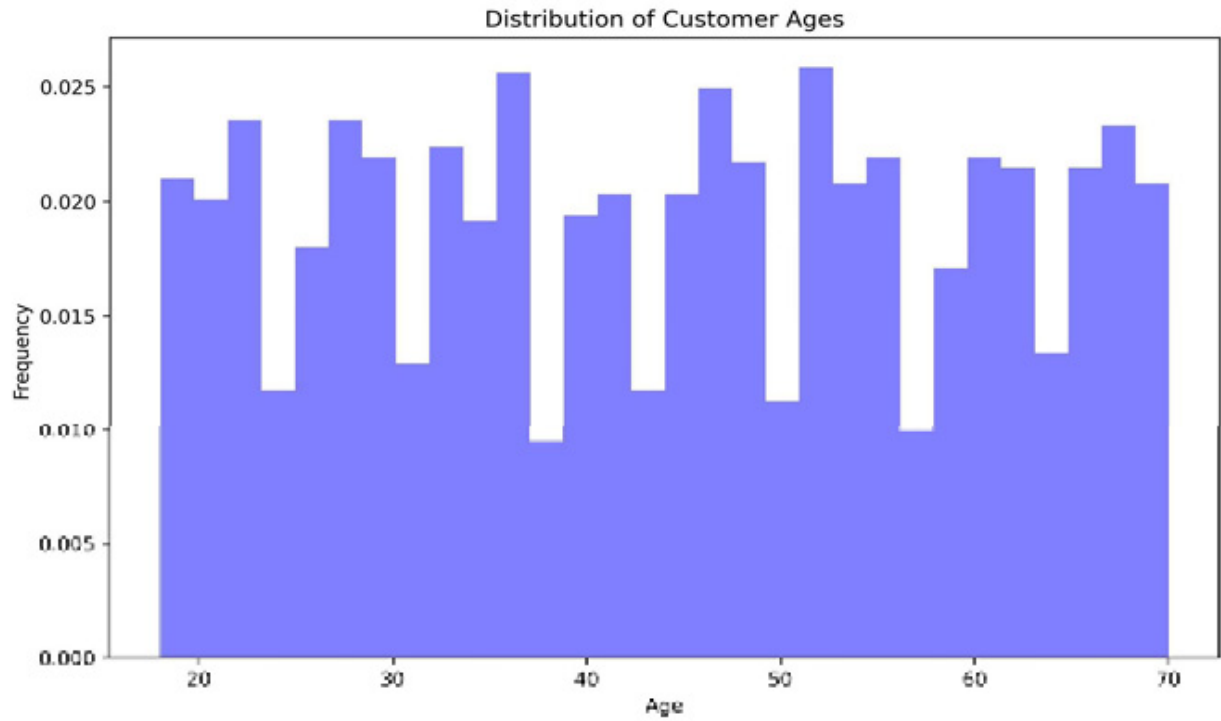


Figure 11.2: Distribution of customer ages

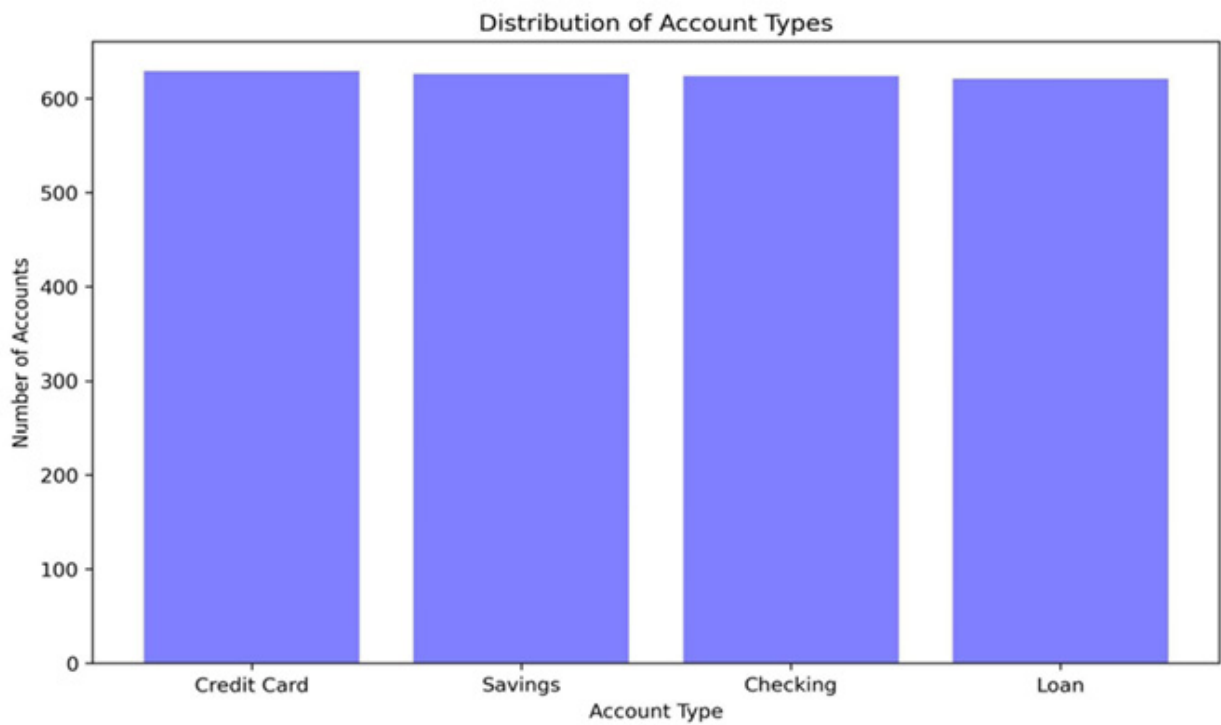


Figure 11.3: Distribution of customers by age in histogram and account type in bar chart

Figure 11.3 shows there is a consistent distribution of customers by account type and by age, ranging from approximately 18 to 70 years old.

Part 2: Statistical testing

We will perform statistical tests to examine differences in account types based on education level. For this, we perform chi square test, compute the p-value and the expected frequencies. Expected frequencies represent the hypothetical counts of observations within each category or combination if there were no association between the variables being studied.

For example, Let us say you are studying the relationship between favorite ice cream flavor (chocolate, vanilla, strawberry) and gender (male, female). If there is no connection between favorite flavor and gender, you would expect an equal distribution of flavors among both males and females. So, the expected frequencies would be roughly equal for each combination of flavor and gender but if the expected frequency distribution is not equal. Then you might find that chocolate is much more popular among males than females, and strawberry is more popular among females than males, suggesting that there might be a relationship between ice cream flavor preference and gender. The code snippet for the statistical testing is as follows:

```
1. # Creating a contingency table of account type by educa  
tion level  
2. contingency_table = pd.crosstab(  
3.     accounts['AccountType'], customers['EducationLevel']  
4. )  
4. # Chi-squared test  
5. chi2, p, dof, expected = chi2_contingency(contingency_t  
6.     able)  
6. # Printing results with labels for better readability  
7. print("Chi-squared Test results:")
```

```

8. print(f"P-value: {p}")
9. print("\nExpected Frequencies:")
10. # Printing expected frequencies with proper labels for ea
    sy understanding
11. expected_df = pd.DataFrame(expected,
12.                             index=contingency_table.index,
13.                             columns=contingency_table.columns)
14. print(expected_df)

```

Following is the output of *Project I*, part 2:

```

1. Chi-squared Test results:
2. P-value: 0.4387673577903144
3. Expected Frequencies:
4. EducationLevel  Bachelor  High School  Master  PhD
5. AccountType
6. Checking        154.9152    156.2192  173.1712  167.694
   4
7. Credit Card     143.0352    144.2392  159.8912  154.834
   4
8. Loan            143.2728    144.4788  160.1568  155.0916
9. Savings         152.7768    154.0628  170.7808  165.3796

```

Output shows p-value above the standard significance level of 0.05, so we cannot reject the null hypothesis, indicating no significant association between **Account Type** and **Education Level**. Larger expected frequencies signify higher anticipated counts, suggesting a greater likelihood of those outcomes under the assumption of independence between variables, while smaller values imply lower expected counts.

Part 3: Analyze the credit card risk

Now, we will analyze the credit card risk based on education level, marital status, account type, loan type,

interest rate, credit limit. To do so, we will create a comprehensive dataset that includes various attributes from different **Comma Separated Value (CSV)** files and categorizing credit card risk into high, medium, and low based on several factors.

We use the following conditions and features to analyze what is the risk for issuing a credit card:

- **Interest rate:** High risk if the interest rate are high.
- **Credit limit:** High risk if the credit limit is high.
- **Education level:** Higher educational levels might correlate with lower risk.
- **Marital status:** Married individuals might be considered lower risk compared to single ones.
- **Account type:** Certain account types like loans might carry higher risk than others like savings.
- **Loan amount:** Larger loan amounts could be considered higher risk.

The following code creates a comprehensive dataset by merging useful datasets and applying conditions to categorize risk, and saving the new dataset with a column credit card risk type:

```
1. # Merge customers with accounts
2. customer_accounts = pd.merge(customers, accounts, on
    ='CustomerID', how='inner')
3. # Merge the above result with loans
4. customer_accounts_loans = pd.merge(
5.     customer_accounts, loans, on='AccountNumber', how
    ='inner')
6. # Merge the complete data with credit cards
7. complete_data = pd.merge(customer_accounts_loans,
8.                             credit_cards, on='AccountNumber', how
    ='inner')
9. # Function to categorize credit card risk, using the cond
```


itions

```
10. def categorize_risk(row):
11.     # Base risk score initialization
12.     risk_score = 0
13.     # Credit Limit and Interest Rate Conditions
14.     if row['CreditLimit'] > 7000 or row['InterestRate'] > 7
       :
15.         risk_score += 3
16.     elif 5000 < row['CreditLimit'] <= 7000 or 5 < row['Int
       erestRate'] <= 7:
17.         risk_score += 2
18.     else:
19.         risk_score += 1
20.     # Education Level Condition
21.     if row['EducationLevel'] in ['PhD', 'Master']:
22.         risk_score -= 1 # Lower risk if higher education
23.     elif row['EducationLevel'] in ['High School']:
24.         risk_score += 1 # Higher risk if lower education
25.     # Marital Status Condition
26.     if row['MaritalStatus'] == 'Married':
27.         risk_score -= 1
28.     elif row['MaritalStatus'] in ['Single', 'Divorced', 'Wido
       wed']:
29.         risk_score += 1
30.     # Account Type Condition
31.     if row['AccountType'] in ['Loan', 'Credit Card']:
32.         risk_score += 2
33.     elif row['AccountType'] in ['Savings', 'Checking']:
34.         risk_score -= 1
35.     # Loan Amount Condition
36.     if row['LoanAmount'] > 20000:
```

```

37.     risk_score += 2
38.     elif row['LoanAmount'] <= 5000:
39.         risk_score -= 1
40.     # Categorize risk based on final risk score
41.     if risk_score >= 5:
42.         return 'High'
43.     elif 3 <= risk_score < 5:
44.         return 'Medium'
45.     else:
46.         return 'Low'
47. # Apply the function to determine credit card risk type
48. complete_data['credit_cards_risk_type'] = complete_data
49.     categorize_risk, axis=1)
50. # Select the relevant columns
51. credit_cards_risk = complete_data[['CustomerID', 'Educ
52.     ationLevel', 'MaritalStatus',
53.     'AccountType', 'LoanAmount', 'Inte
54.     restRate', 'CreditLimit', 'credit_cards_risk_type']]

```

Following is the output of *Project I*, Part 3:

	CustomerID	EducationLevel	MaritalStatus	AccountType	LoanAmount	InterestRate	CreditLimit	credit_cards_risk_type
0	34da4208-3b9e-486b-bd63-34381ad846dc	PhD	Widowed	Credit Card	39117	7.63	6880	High
1	08d50394-bcf1-4537-861d-2eb481c938ee	Master	Divorced	Credit Card	22027	5.06	4175	High
2	a187a327-f9db-4fda-ac9c-05559737cdb3	Master	Divorced	Loan	46136	5.23	6347	High
3	1872823f-a112-45b3-84c5-96500b9ee8a7	High School	Widowed	Savings	8390	4.53	1285	Low
4	c1c5156a-ffec-48da-97dc-64899a696a99	High School	Married	Credit Card	6735	1.74	7614	High

Figure 11.4: Data frame with customer bank details and credit card risk

[Figure 11.4](#) is a data frame with a new column credit cards risk type, which indicates the risk level of the customer for issuing credit cards.

Part 4: Predictive modeling

Now that we have a new data frame as shown in [Figure](#)

11.4, in it we will use regression to predict credit limits and classification to identify high-risk accounts with credit cards risk type as the output or target variable and others as inputs or predictors but before we apply logistic regression, we will encode categorical columns into integer numbers. The code snippet for encoding categorical values into numbers and applying predictive modelling as following:

```
1. # Load data : EducationLevel MaritalStatus AccountType Amount LoanType InterestRate CreditLimit
2. credit_cards = pd.read_csv('credit_cards_risk.csv')
3. # Mapping catagorrical variables into numerical values as follow
4. education_levels = {"High School": 0, "Bachelor": 1, "Master": 2, "PhD": 3}
5. marital_status = {"Single": 0, "Married": 1, "Divorced": 2, "Widowed": 3}
6. account_types = {"Checking": 0, "Savings": 1, "Credit Card": 2, "Loan": 3}
7. risk_types = {"Low": 0, "Medium": 1, "High": 2}
8. # Apply the mapping to the respective columns
9. credit_cards['EducationLevel'] = credit_cards['EducationLevel'].map(
10.     education_levels)
11. credit_cards['MaritalStatus'] = credit_cards['MaritalStatus'].map(
12.     marital_status)
13. credit_cards['AccountType'] = credit_cards['AccountType'].map(account_types)
14. credit_cards['credit_cards_risk_type'] = credit_cards['credit_cards_risk_type'].map(
15.     risk_types)
16. # Prepare data for logistic regression
17. X = credit_cards[['EducationLevel', 'MaritalStatus', 'Acc
```

```

ountType',
18.         'LoanAmount', 'InterestRate', 'CreditLimit']]
    # Predictor
19. y = credit_cards['credit_cards_risk_type'] # Response variable
20. # Splitting data into training and testing sets
21. X_train, X_test, y_train, y_test = train_test_split(
22.     X, y, test_size=0.2, random_state=42)
23. # Create a logistic regression model
24. model = LogisticRegression()
25. model.fit(X_train, y_train)
26. # Predictions and evaluation
27. predictions = model.predict(X_test)
28. print(classification_report(y_test, predictions))

```

Following is the output of *Project I*, part 4:

The trained model evaluation matrices scores are as follows:

1.		precision	recall	f1-score	support
2.					
3.	0	0.85	0.34	0.48	116
4.	1	0.52	0.57	0.54	133
5.	2	0.73	0.90	0.81	251
6.					
7.	accuracy			0.68	500
8.	macro avg	0.70	0.60	0.61	500
9.	weighted avg	0.70	0.68	0.66	500

Part 5: Use the predictive model above Part 4. Feed it user input and see predictions

Finally, we will take **EducationLevel**, **MaritalStatus**, **AccountType**, **LoanType**, **InterestRate**, **CreditLimit** as user input to see what **credit_card_risk_type** (high, low,

medium) the trained prediction model predicts. The code snippet to do so is as following:

```
1. # Define mappings for categorical input to integer encoding
2. education_levels = {"High School": 0, "Bachelor": 1, "Master": 2, "PhD": 3}
3. marital_status = {"Single": 0, "Married": 1, "Divorced": 2, "Widowed": 3}
4. account_types = {"Checking": 0, "Savings": 1, "Credit Card": 2, "Loan": 3}
5. risk_type_options = {0: 'Low', 1: 'Medium', 2: 'High'}
6. # Function to get user input and convert to encoded value
7. def get_user_input(prompt, category_dict):
8.     while True:
9.         response = input(prompt)
10.        if response in category_dict:
11.            return category_dict[response]
12.        else:
13.            print("Invalid entry. Please choose one of:",
14.                  list(category_dict.keys()))
15. # Function to get numerical input and validate it
16. def get_numerical_input(prompt):
17.     while True:
18.         try:
19.             value = float(input(prompt))
20.             return value
21.         except ValueError:
22.             print("Invalid entry. Please enter a valid number.")
23. # Collect inputs
24. education_level = get_user_input(
```

```

25.     "Enter Education Level (High School, Bachelor, Master, PhD): ", education_levels)
26. marital_status = get_user_input(
27.     "Enter Marital Status (Single, Married, Divorced, Widowed): ", marital_status)
28. account_type = get_user_input(
29.     "Enter Account Type (Checking, Savings, Credit Card, Loan): ", account_types)
30. loan_amount = get_numerical_input("Enter Loan Amount: ")
31. interest_rate = get_numerical_input("Enter Interest Rate: ")
32. credit_limit = get_numerical_input("Enter Credit Limit: ")
33. # Prepare the input data for prediction
34. input_data = pd.DataFrame({
35.     'EducationLevel': [education_level],
36.     'MaritalStatus': [marital_status],
37.     'AccountType': [account_type],
38.     'LoanAmount': [loan_amount],
39.     'InterestRate': [interest_rate],
40.     'CreditLimit': [credit_limit]
41. })
42. # Predict the risk type
43. prediction = model.predict(input_data)
44. print("Predicted Risk Type:", risk_type_options[prediction[0]])

```

Upon running the above *Project I*, part 5 snippet, you will be asked to provide the necessary input, based on which the predictive model will tell you if the risk is high, medium or low.

Project II: Implementing data science and statistical analysis on health data

Project II explores the extensive capabilities of Python for implementing statistics concepts in the realm of health data analysis. Here we use a synthetic health data set generated for this tutorial. The dataset includes a variety of medical measurements reflecting common data types collected in health informatics. This synthetic dataset simulates realistic health records with 2500 entries containing metrics such **Body Mass Index (BMI)**, glucose level, blood pressure, heart rate, cholesterol, hemoglobin, white blood cell count, and platelets. Each record also contains a unique patient ID and a binary health outcome indicating the presence or absence of a particular condition. This structure supports analyses ranging from basic descriptive statistics to complex machine learning models. The primary goal of this project is to provide hands-on experience and demonstrate how Python can be used to manipulate, analyze, and predict health outcomes based on statistical data. In Project II, we first perform exploratory data analysis to view and better understand the data set. Then we apply statistical analysis to look at the correlation and covariance between features. This is followed by inferential statistics where we compute t-statistics, p-value, and confidence interval for selected features if of interest. Finally, a statistical logistic regression model is trained to classify health outcomes as binary values representing good and bad health, and the results are evaluated.

Part 1: Exploratory data analysis

In part 1 of Project II, we will examine data to understand distributions and relationships. We will visualize the distribution using histograms, box plots, and, then plot relationship between glucose level and cholesterol in a scatter plots. And then view summary of data through

measures of central tendency and variability. The code snippet is as follows:

```
1. # Load the data
2. data = pd.read_csv(
3.     '/workspaces/ImplementingStatisticsWithPython/note
   books/chapter11project/health/data/synthetic_health_data.csv')
4. # Define features for plots
5. features = ['Health_Outcome', 'Body_Mass_Index', 'Glucose_Level', 'Blood_Pressure',
6.             'Heart_Rate', 'Cholesterol', 'Haemoglobin', 'White_Blood_Cell_Count', 'Platelets']
7. # Plot histograms
8. fig, axs = plt.subplots(3, 3, figsize=(15, 10))
9. for ax, feature in zip(axs.flatten(), features):
10.     ax.hist(data[feature], bins=20, color='skyblue', edgecolor='black')
11.     ax.set_title(f'Histogram of {feature}')
12. plt.tight_layout()
13. plt.savefig('health_histograms.png', dpi=300, bbox_inches='tight')
14. plt.show()
```

Following is the output of *Project II*, part 1:

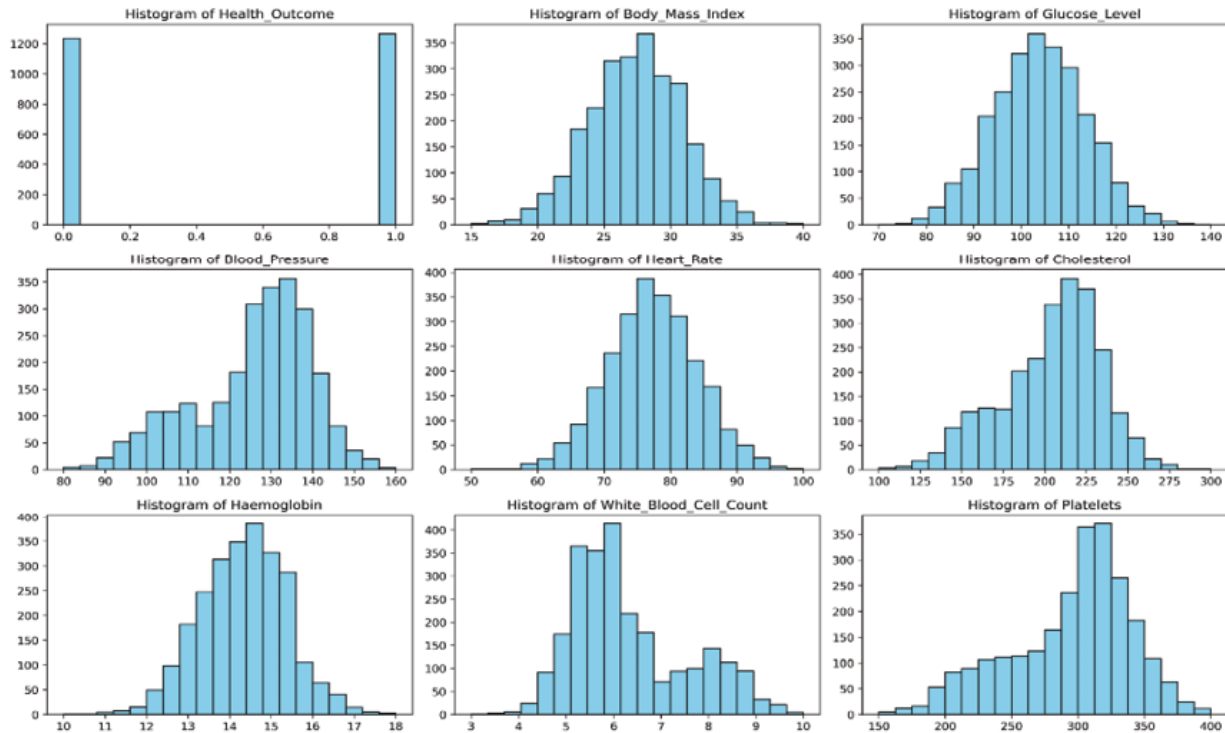


Figure 11.5: Distribution of customers across each feature

To see the distribution in box plot, code snippet is as follows:

1. `# Plot box plots`
2. `plt.figure(figsize=(12, 8))`
3. `sns.boxplot(data=data[features])`
4. `plt.xticks(rotation=45)`
5. `plt.title('Box Plot of Selected Variables')`
6. `plt.savefig('health_boxplot.png', dpi=300, bbox_inches='tight')`
7. `plt.show()`

Figure 11.6 shows the median, quartiles, and outliers, which represent the spread, skewness, and central tendency of the data in the box plot.

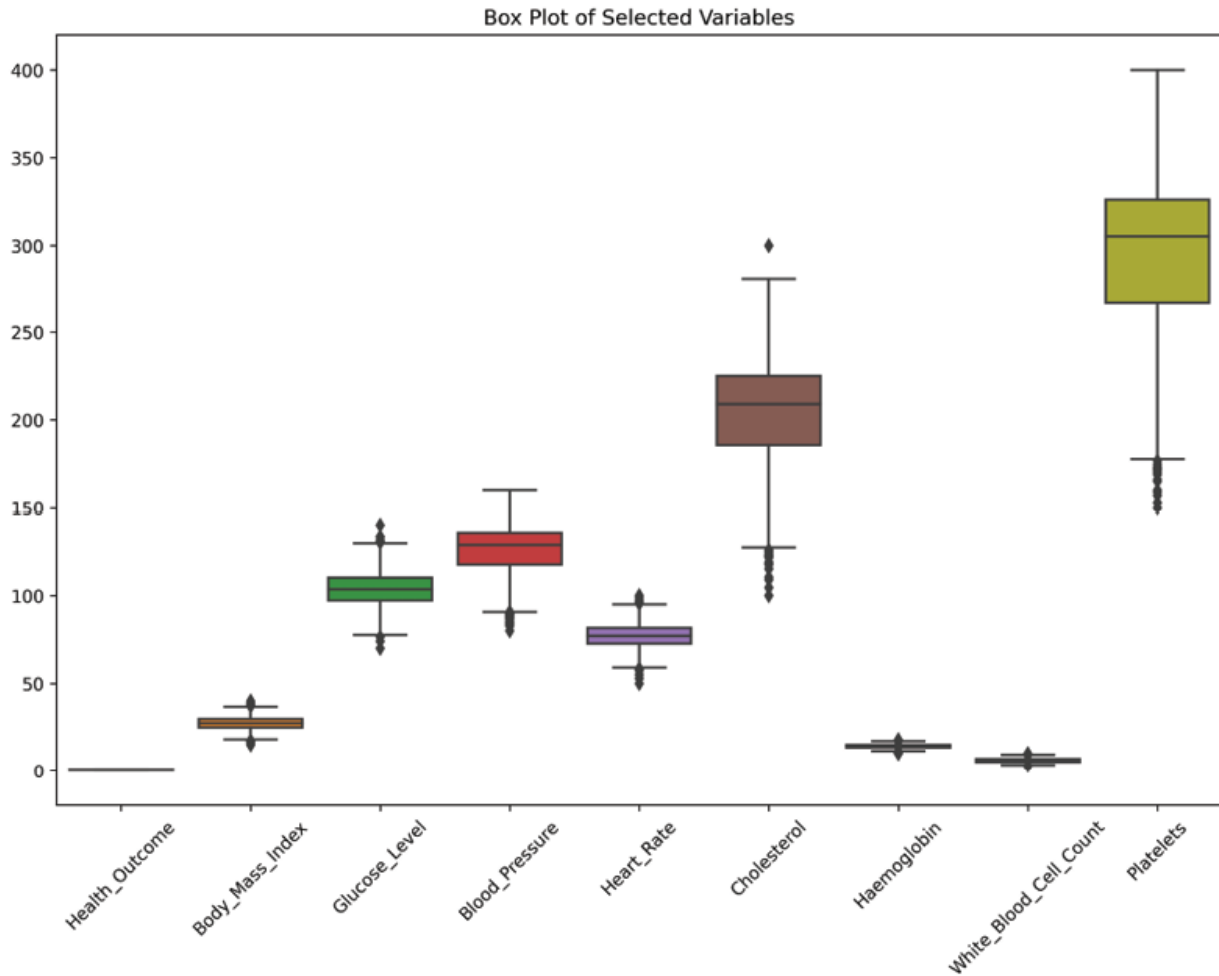


Figure 11.6: Box plot showing spread, skewness, and central tendency across each feature

Then, to see the distribution in scatter plot, code snippet is as follows:

1. *# Scatter plot of two variables*
2. `sns.scatterplot(x='Glucose_Level', y='Cholesterol', data=data)`
3. `plt.title('Scatter Plot of Glucose Level vs Cholesterol')`
4. `plt.savefig('health_scatterplot.png', dpi=300, bbox_inches='tight')`
5. `plt.show()`

Figure 11.7 shows that the majority of patients have glucose levels from 80 to 120 (milligrams per deciliter) and

cholesterol from 125 to 250 (milligrams per deciliter):

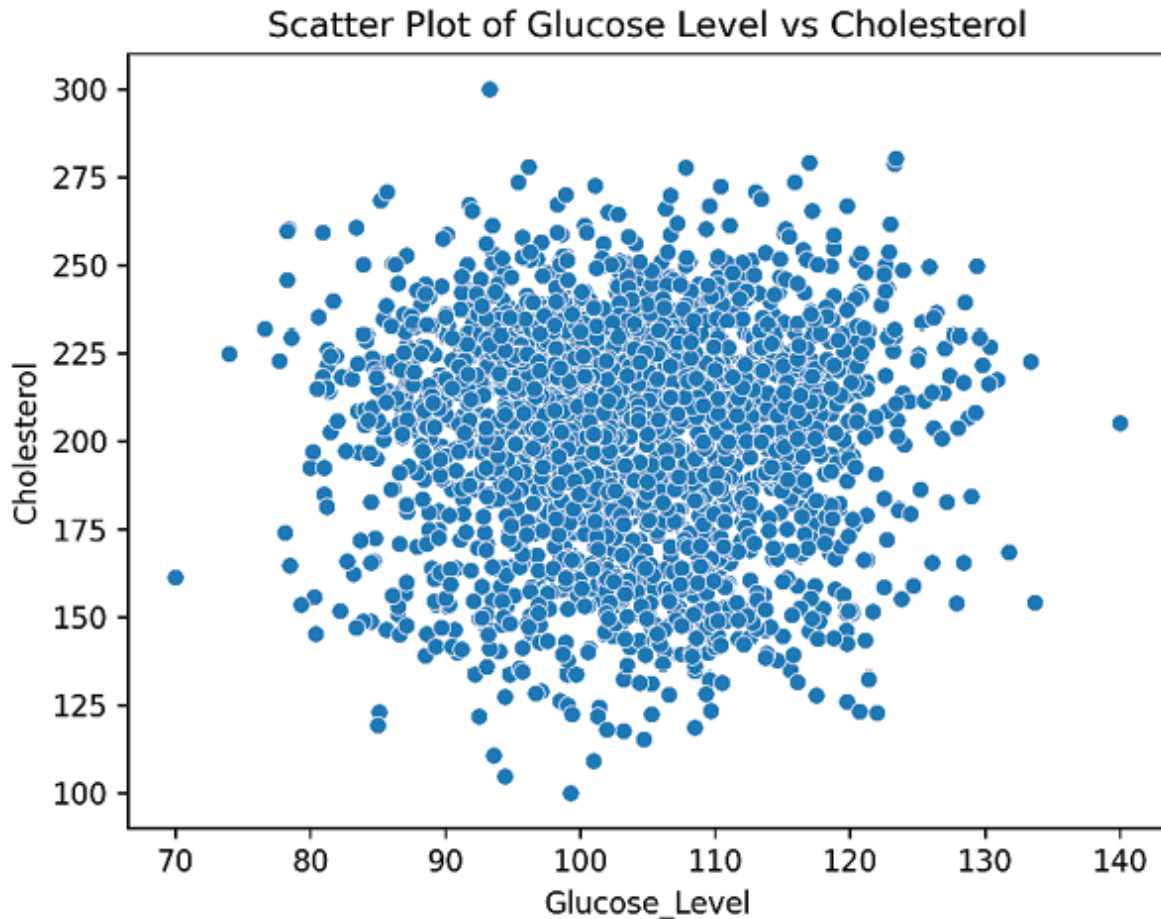


Figure 11.7: Scatter plot to view relationship between cholesterol and glucose level

This following code displays the summary statistics of the features in data:

1. *# Print descriptive statistics for the selected features*
2. `display(data[features].describe())`

Figure 11.8 shows platelets variable has wide range of values, with a minimum of 150 and a maximum of 400. This suggests considerable variation in platelet counts within the dataset, which may be important for understanding potential health outcomes.

	Health_Outcome	Body_Mass_Index	Glucose_Level	Blood_Pressure	Heart_Rate	Cholesterol	Haemoglobin	White_Blood_Cell_Count	Platelets
count	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000
mean	0.506000	27.386360	103.778120	125.760120	77.287960	204.128280	14.297880	6.290240	294.893600
std	0.500064	3.508407	9.687728	14.084897	6.773203	30.023524	1.047151	1.226606	45.792266
min	0.000000	15.000000	70.000000	80.000000	50.000000	100.000000	10.000000	3.000000	150.000000
25%	0.000000	25.000000	97.175000	117.775000	72.700000	185.900000	13.600000	5.400000	266.675000
50%	1.000000	27.500000	103.750000	128.800000	77.300000	209.100000	14.300000	6.000000	305.300000
75%	1.000000	29.800000	110.400000	135.800000	81.800000	225.025000	15.000000	7.000000	326.150000
max	1.000000	40.000000	140.000000	160.000000	100.000000	300.000000	18.000000	10.000000	400.000000

Figure 11.8: Summary statistics of selected features

Part 2: Statistical analysis

Here we will view relationships between variables using covariance and correlation and look out for outliers using z score measure. The code is as follows:

```

1. # Select features for analysis
2. features = ['Health_Outcome', 'Body_Mass_Index', 'Glucose_Level', 'Blood_Pressure',
3.             'Heart_Rate', 'Cholesterol', 'Haemoglobin', 'White_Blood_Cell_Count', 'Platelets']
4. # Analyzing relationships between variables using covariance and correlation.
5. # Correlation matrix
6. correlation_matrix = data[features].corr()
7. plt.figure(figsize=(10, 8))
8. sns.heatmap(correlation_matrix, annot=True)
9. plt.title('Correlation Matrix')
10. plt.show()

```

The above code illustrates correlations between chosen features. A correlation coefficient of +1 denotes high positive correlation, indicating that as one feature increases, the other also increases, and vice versa. Conversely, a coefficient of -1 signifies high negative correlation, suggesting that as one feature increases, the other decreases, and vice versa as following:

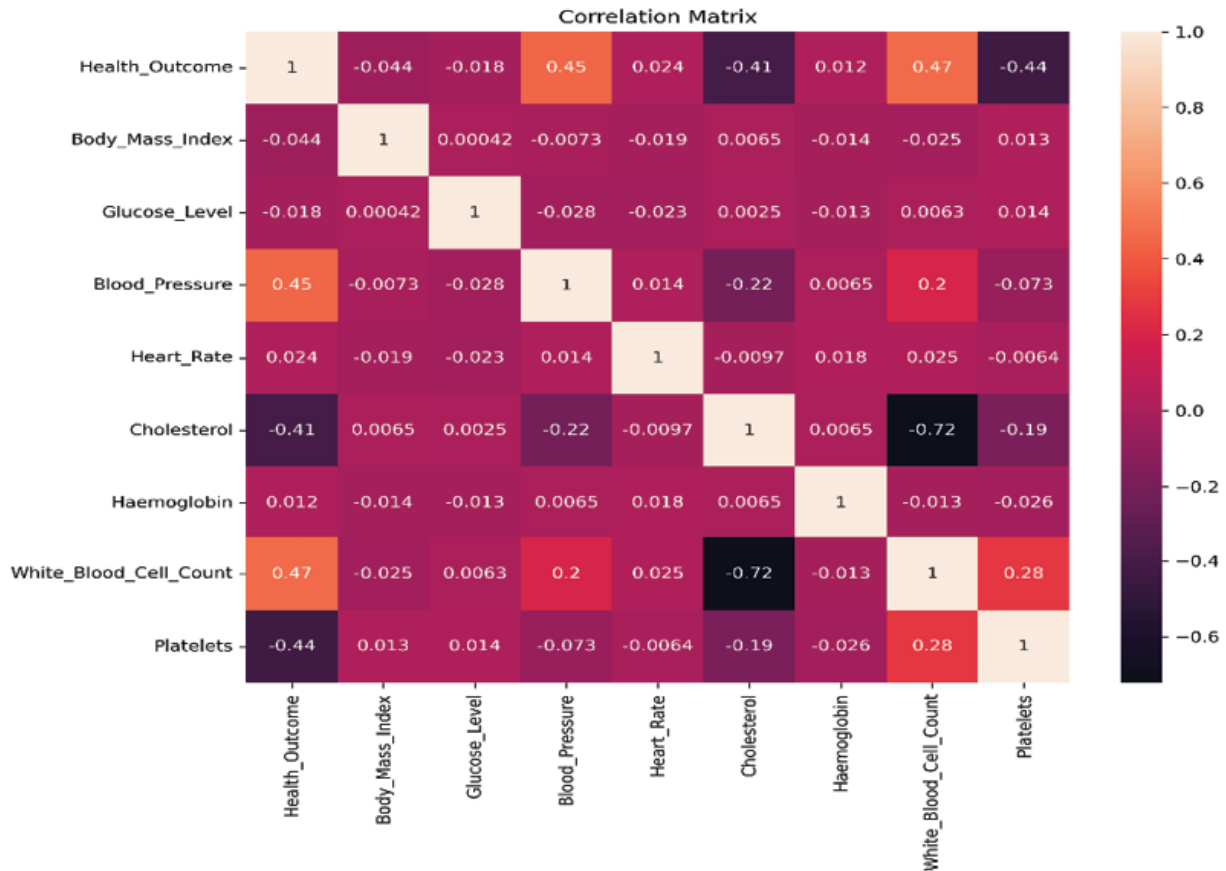


Figure 11.9: Correlation matrix of features, color intensity represents level of correlation

Again, we employ a covariance matrix to observe covariance values. A high positive covariance indicates that both variables move in the same direction as one increases, the other tends to increase and vice versa. Conversely, a high negative covariance implies that both variables move in opposite directions as one increases, the other tends to decrease, and vice versa. The following code illustrates the covariance between features:

```
1. # Covariance matrix
2. covariance_matrix = data[features].cov()
3. print("Covariance Matrix:")
4. display(covariance_matrix)
```

Then, using the following code, we will calculate the z-scores for each element in the dataset, z-score quantifies

how many standard deviations a data point is from the dataset's mean. Here we use the condition `abs_z_scores > 1`. This metric is crucial for identifying outliers, as it provides a standardized way to detect outliers. As the output, it does not detect any outliers:

```
1. # Identifying outliers and understanding their impact.
2. # Z-score for outlier detection
3. z_scores = zscore(data)
4. abs_z_scores = np.abs(z_scores)
5. outliers = (abs_z_scores > 1).all(axis=1)
6. data_outliers = data[outliers]
7. print("Detected Outliers:")
8. print(data_outliers)
```

Part 3: Inferential statistics

Now, we will use statistical methods to infer population characteristics from glucose level data categorized by health outcomes. We will begin by performing a t-test to compare the mean glucose levels between groups with different health outcomes, yielding a t-statistic and p-value to assess the significance of differences. Following this, we will calculate the 95% confidence interval for the overall mean glucose level, providing a range that likely includes the true mean with 95% certainty. These steps help determine the relationship between health outcomes and glucose levels and estimate the mean glucose level's variability as follows:

```
1. # Apply T-test
2. group1 = data[data['Health_Outcome'] == 0]
   ['Glucose_Level']
3. group2 = data[data['Health_Outcome'] == 1]
   ['Glucose_Level']
4. t_stat, p_val = ttest_ind(group1, group2)
```

```

5. print(f"T-statistic: {t_stat}, P-value: {p_val}")
6. # Confidence interval for the mean of a column
7. ci_low, ci_upp = norm.interval(
8.     alpha=0.95, loc=data['Glucose_Level'].mean(), scale=
      data['Glucose_Level'].std())
9. print(
10.     f"95% confidence interval for the mean glucose level:
      ({ci_low}, {ci_upp})")

```

Following is the output of *Project II*, part 3:

A T-statistic of 0.92 indicates a moderate difference between the mean glucose levels of two groups. The P-value of 0.36 indicates that there is a 36% chance of observing such a difference if there were no true difference between the groups and the confidence interval score suggests that we are 95% confident that the true mean glucose level for Group 1 is between 84.79 and 122.77, and similarly for Group 2 as follows:

```

1. T-statistic:          0.9204677863057696,          P-
   value: 0.3574172393450691
2. 95% confidence interval for the mean glucose level:
   (84.79052199831503, 122.76571800168497)

```

Part 4: Statistical machine learning

Finally, we will train a logistic regression model using input features (**Body_Mass_Index**, **Glucose_Level**, **Blood_Pressure**, **Heart_Rate**, **Cholesterol**, **Haemoglobin**, **White_Blood_Cell_Count**, **Platelets**) to predict binary class outcomes (**Health_Outcome**) and evaluates the model's accuracy, displays a confusion matrix for insight into performance, and plots a **Receiver-Operating Characteristic Curve (ROC)** curve to assess its ability to classify instances as follows:

```

1. X = data.drop(['Health_Outcome', 'Patient_ID'], axis=1)

```



```

2. y = data['Health_Outcome']
3. X_train, X_test, y_train, y_test = train_test_split(
4.     X, y, test_size=0.3, random_state=42)
5. model = LogisticRegression()
6. model.fit(X_train, y_train)
7. predictions = model.predict(X_test)
8. # Accuracy and confusion matrix
9. print("Accuracy:", model.score(X_test, y_test))
10. print("Confusion Matrix:")
11. print(confusion_matrix(y_test, predictions))
12. # ROC Curve and AUC
13. probs = model.predict_proba(X_test)[:, 1]
14. fpr, tpr, thresholds = roc_curve(y_test, probs)
15. roc_auc = auc(fpr, tpr)
16. plt.figure(figsize=(8, 6))
17. plt.plot(fpr, tpr, color='darkorange', lw=2,
18.         label=f'ROC curve (area = {roc_auc:.2f})')
19. plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
20. plt.xlabel('False Positive Rate')
21. plt.ylabel('True Positive Rate')
22. plt.title('Receiver Operating Characteristic (ROC) Curve'
23.         )
24. plt.legend(loc="lower right")
25. plt.show()

```

Following is the output of *Project II*, part 4:

As a result, we obtained a trained model with an accuracy of 94.26%, which means that the model correctly predicts the outcome about 94.26% of the time and the receiver operating characteristics curve value of 97% indicates that the model has a high true positive rate and a low false positive rate, which means strong predictive ability as follows:

1. Accuracy: 0.9426666666666667

2. Confusion Matrix:

3. $\begin{bmatrix} 331 & 28 \end{bmatrix}$

4. $\begin{bmatrix} 15 & 376 \end{bmatrix}$

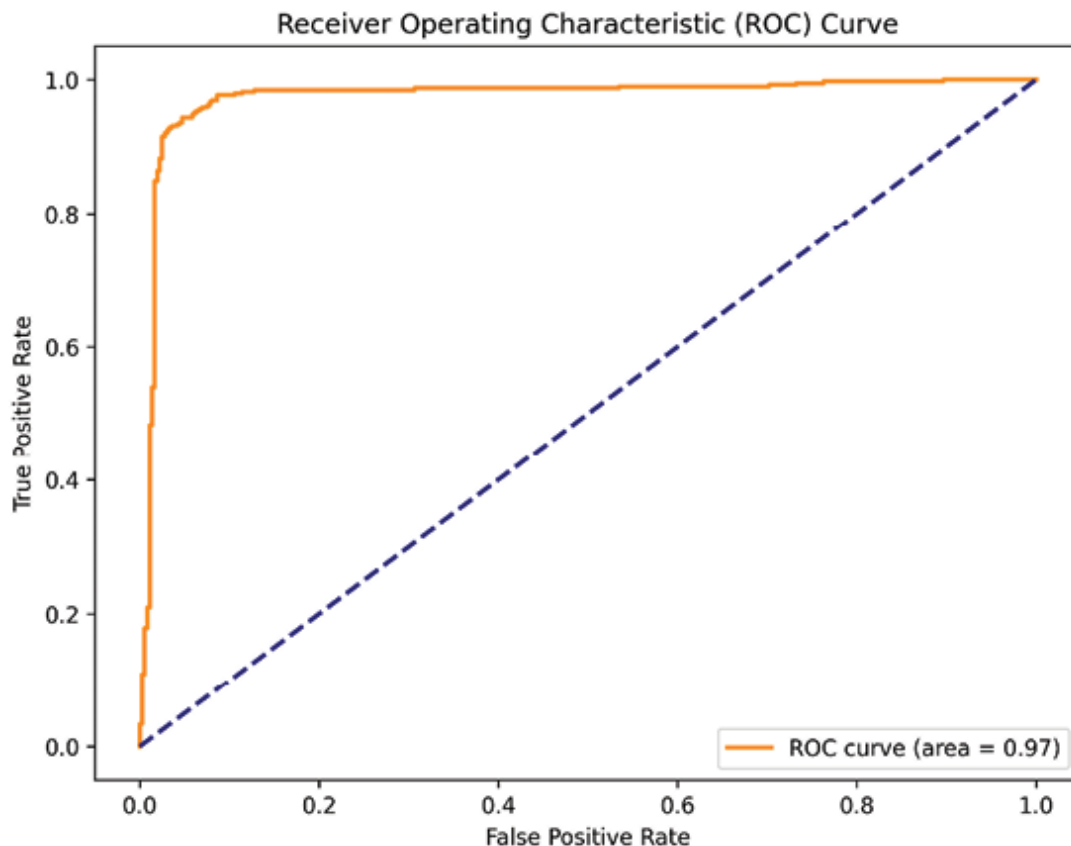


Figure 11.10: Receiver operating characteristic curve of the health outcome prediction model

Conclusion

This chapter provided a hands-on experience in the practical application of data science and statistical analysis in two critical sectors: banking and healthcare. Using synthetic data, the chapter demonstrated how the theories, methods, and techniques covered throughout the book can be skillfully applied to real-world contexts. However, the use of statistics, data science, and Python programming extends

far beyond these examples. In banking, additional applications include fraud detection and risk assessment, customer segmentation, and forecasting. In healthcare, applications extend to predictive modelling for patient outcomes, disease surveillance and public health management, and improving operational efficiency in healthcare systems.

Despite these advances, the real-world use of data requires careful consideration of ethical, privacy, and security issues, which are paramount and must always be carefully addressed. In addition, the success of statistical applications is highly dependent on the quality and granularity of the data, making data quality and management equally critical. With ongoing technological advancements and regulatory changes, there is a constant need to learn and adapt new methodologies and tools. This dynamic nature of data science requires practitioners to remain current and flexible to effectively navigate the evolving landscape.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

- alternative hypothesis [114](#), [200](#)
- Anaconda
 - installation [3](#)
 - launching [3](#)
- anomalies [139-142](#)
- Apriori [267](#)
 - implementing [268](#), [269](#)
- arrays [155](#)
 - 1-Dimensional array [155](#)
 - 2-Dimensional array [156](#)
 - uses [157](#)
- Artificial General Intelligence (AGI) [311](#)
- autoregressive models [313](#)

B

- Bidirectional Encoder Representations from Transformers (BERT) [253](#)
- binary coding [84](#), [85](#)
- binomial distribution [151](#)
- binom.interval function [176](#)
- bivariate analysis [26](#), [27](#)
- bivariate data [26](#), [27](#)
- body mass index (BMI) [96](#), [213](#)
- Bokeh [92](#)
- bootstrapping [289](#), [293](#)

C

- Canonical Correlation Analysis (CCA) [30](#)
- Chain-of-Thought (CoT) [318](#)
- chi-square test [118-120](#), [210](#)
- clinical trial rating [287](#)
- cluster analysis [29](#)
- collection methods [33](#)
- Comma Separated Value (CSV) files [332](#)
- confidence interval [161](#), [172](#), [173](#)
 - estimation for diabetes data [179-183](#)
 - estimation in text [183-185](#)

- for differences [177-179](#)
- for mean [175](#)
- for proportion [176](#), [177](#)
- confidence intervals [169](#), [170](#)
 - types [170](#), [171](#)
- contingency coefficient [124](#)
- continuous data [13](#)
- continuous probability distributions [148](#)
- convolutional neural networks (CNNs) [138](#)
- correlation [117](#), [138](#), [139](#)
 - negative correlation [138](#), [139](#)
 - positive correlation [138](#)
- co-training [251](#)
- covariance [116](#), [117](#), [136-138](#)
- Cramer's V [120-123](#)
- cumulative frequency [106](#)

D

- data [5](#)
 - qualitative data [6-8](#)
 - quantitative data [8](#)
- data aggregation [50](#)
 - mean [50](#), [51](#)
 - median [51](#), [52](#)
 - mode [52](#), [53](#)
 - quantiles [55](#)
 - standard deviation [54](#)
 - variance [53](#), [54](#)
- data binning [72-77](#)
- data cleaning
 - duplicates [42](#), [43](#)
 - imputation [40](#), [41](#)
 - missing values [39](#), [40](#)
 - outliers [43-45](#)
- data encoding [82](#), [83](#)
- data frame
 - standardization [66](#)
- data grouping [77-79](#)
- data manipulation [45](#), [46](#)
- data normalization [58](#), [59](#)
 - NumPy array [59-61](#)
 - pandas data frame [61-64](#)
- data plotting [92](#), [93](#)
 - bar chart [95](#), [96](#)
 - dendrograms [100](#)

- graphs [100](#)
- line plot [93](#)
- pie chart [94](#)
- scatter plot [97](#)
- stacked area chart [99](#)
- violin plot [100](#)
- word cloud [100](#)
- data preparation tasks [35](#)
 - cleaning [39](#)
 - data quality [35-37](#)
- data science and statistical analysis, on banking data
 - credit card risk, analyzing [332-335](#)
 - exploratory data analysis (EDA) [329-331](#)
 - implementing [328](#), [329](#)
 - predictive modeling [335-338](#)
 - statistical testing [331](#), [332](#)
- data science and statistical analysis, on health data
 - exploratory data analysis [339-342](#)
 - implementing [338](#), [339](#)
 - inferential statistics [344](#), [345](#)
 - statistical analysis [342-344](#)
 - statistical machine learning [345](#), [346](#)
- data sources [32](#), [33](#)
- data standardization [58](#), [64](#), [65](#)
 - data frame [66](#)
 - NumPy array [66](#)
- data transformation [58](#), [67-70](#)
- data wrangling [45](#), [46](#)
- decision tree [235-238](#)
- dendrograms [100](#)
- Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [264](#)
- describe() [18](#)
- descriptive statistics [103](#)
- detect_outliers function [142](#)
- discrete data [12](#)
- discrete probability distributions [147](#)
- dtype() [17](#)

E

- Eclat [270](#)
 - implementing [270](#)
- effective prompts
 - best practices [322](#), [323](#)
- Enchant [45](#)
- environment setup [2](#)

Exploratory Data Analysis (EDA) [49](#)
 importance [50](#)
Exploratory Factor Analysis (EFA) [30](#)

F

factor analysis [30](#)
feature scaling [88](#)
few-shot learning [317](#)
First Principal Component (PC1) [32](#)
FP-Growth [273](#)
 implementing [273](#), [274](#)
frequency distribution [106](#)
frequency tables [106](#)

G

Gaussian distribution [150](#)
Gaussian Mixture Models (GMMs) [260](#)
 implementing [261](#)
generated knowledge prompting [319](#)
Generative Adversarial Networks (GANs) [313](#)
generative AI models [320](#)
Generative Artificial Intelligence (AI) [311-313](#)
GitHub Codespaces [3](#)
goodness-of-fit tests [289](#)
Google Collaboratory [3](#)
GPT-4
 setting up in Python, OpenAI API used [320-322](#)
graph-based methods [252](#)
graphs [100](#)
groupby() [22](#)
groupby().sum() [23](#)

H

hash coding [87](#)
head() [21](#)
hierarchical clustering [259](#)
 implementing [260](#)
histograms [96](#)
hypothesis testing [114](#), [187-190](#)
 in diabetes dataset [213-215](#)
 one-sided testing [193](#)
 performing [191-193](#)
 two-sample testing [196](#)
 two-sided testing [194](#), [195](#)

I

- independence tests [289](#), [290](#)
- independent tests [197](#)
- industry-specific use cases, LLMs [324](#)
- info() [20](#)
- integrated development environment (IDE) [2](#)
- Interquartile Range (IQR) [61](#)
- interval data [13](#)
- interval estimate [164-166](#)
- is_numeric_dtype() [19](#)
- is_string_dtype() [19](#)

K

- Kaplan-Meier estimator [295](#)
- Kaplan-Meier survival curve analysis
 - implementing [300-304](#)
- Kendall's Tau [291](#)
- Kernel Density Estimation (KDE) [294](#)
- K-means clustering [257](#), [258](#)
- K modes [259](#)
- K-Nearest Neighbor (KNN) [242](#)
 - implementing [242](#)
- K-prototype clustering [258](#), [259](#)
- Kruskal-Wallis test [289](#), [292](#)
- kurtosis [132](#), [133](#)

L

- label coding [83](#)
- language model [254](#)
- Large Language Model (LLM) [312](#), [314](#), [320](#)
 - industry-specific use cases [324](#), [325](#)
- left skew [128](#)
- leptokurtic distribution [132](#)
- level of measurement [10](#)
 - continuous data [13](#)
 - discrete data [12](#)
 - interval data [13](#)
 - nominal data [10](#)
 - ordinal data [11](#)
 - ratio data [14](#), [15](#)
- linear algebra [280](#)
 - using [283-286](#)
- Linear Discriminant Analysis (LDA) [64](#)
- linear function [281](#)

Linear Mixed-Effects Models (LMMs) [233-235](#)

linear regression [225-231](#)

log10() function [69](#)

logistic regression [231-233](#)

fitting models to dependent data [233](#)

M

machine learning (ML) [222](#), [223](#)

algorithm [223](#)

data [223](#)

fitting models [223](#)

inference [223](#)

prediction [223](#)

statistics [223](#)

supervised learning [224](#)

margin of error [167](#), [168](#)

Masked Language Models (MLM) [253](#)

Matplotlib [5](#), [50](#), [92](#)

matrices [155](#), [282](#)

uses [157](#), [158](#)

mean [50](#), [51](#)

mean deviation [113](#)

measure of association [114-116](#)

chi-square [118-120](#)

contingency coefficient [124-126](#)

correlation [116](#)

covariance [116](#)

Cramer's V [120-124](#)

measure of central tendency [108](#), [109](#)

measure of frequency [104](#)

frequency tables and distribution [106](#)

relative and cumulative frequency [106](#), [107](#)

visualizing [104](#)

measures of shape [126](#)

skewness [126-130](#)

measures of variability or dispersion [110-113](#)

median [51](#), [52](#)

Microsoft Azure Notebooks [3](#)

missing data

data imputation [88-92](#)

model selection and evaluation methods [243](#)

evaluation metrics [243-248](#)

multivariate analysis [28](#), [29](#)

multivariate data [28](#), [29](#)

multivariate regression [29](#)

N

Natural Language Processing (NLP) [142](#), [252](#)

negative skewness [128](#)

NLTK [45](#)

nominal data [10](#)

nonparametric statistics [287](#)

- bootstrapping [293](#), [294](#)

- goodness-of-fit tests [289](#), [290](#)

- independence tests [290-292](#)

- Kruskal-Wallis test [292](#), [293](#)

- rank-based tests [289](#)

- using [288](#), [289](#)

nonparametric test [198](#), [199](#)

normal probability distributions [150](#)

null hypothesis [114](#), [200](#)

NumPy [4](#), [50](#)

NumPy array

- normalization [59-61](#)

- standardization [66](#)

numpy.genfromtxt() [25](#)

numpy.loadtxt() [25](#)

O

one-hot encoding [82](#)

one-shot learning [317](#)

one-way ANOVA [211](#)

open-ended prompts [315](#)

- versus specific prompts [315](#)

ordinal data [11](#)

outliers [139-144](#)

- detecting [88](#)

- treating [88-92](#)

P

paired test [197](#)

pandas [4](#), [50](#)

pandas data frame

- normalization [61-64](#)

parametric test [198](#)

platykurtic distribution [132](#)

Plotly [92](#)

point estimate [162](#), [163](#)

Poisson distribution [153](#)

population and sample [34](#), [35](#)

Principal Component Analysis (PCA) [29-32](#), [64](#), [262](#)

probability [145](#), [146](#)

probability distributions [147](#)

 binomial distribution [151](#), [152](#)

 continuous probability distributions [148](#)

 discrete probability distributions [147](#)

 normal probability distributions [150](#)

 Poisson distribution [153](#), [154](#)

 uniform probability distributions [149](#)

prompt engineering [314](#)

prompt types [315](#)

p-value [173](#), [190](#), [206](#)

 using [174](#)

PySpellChecker [45](#)

Python [4](#)

Q

qualitative data [6](#)

 example [6-8](#)

 versus, quantitative data [17-25](#)

quantile [55-58](#)

quantitative data [8](#)

 example [9](#), [10](#)

R

random forest [238-240](#)

rank-based tests [289](#)

ratio data [14](#), [15](#)

read_csv() [24](#)

read_json() [24](#)

Receiver-Operating Characteristic Curve (ROC) curve [345](#)

relative frequency [106](#)

retrieval augmented generation (RAG) [319](#)

Robust Scaler [61](#)

S

sample [216](#)

sample mean [216](#)

sampling [189](#)

sampling distribution [216-219](#)

sampling techniques [216-218](#)

scatter plot [97](#)

Scikit-learn [50](#)

Scipy [50](#)

- Seaborn [50](#), [92](#)
- Second Principal Component (PC2) [32](#)
- `select_dtypes(include='_____')` [22](#)
- self-consistency prompting [318](#)
- self-supervised learning [248](#)
- self-supervised techniques
 - word embedding [252](#)
- self-training classifier [249](#)
- semi-supervised learning [248](#)
- semi-supervised techniques [249-251](#)
- significance levels [206](#)
- significance testing [187](#), [199-203](#)
 - ANOVA [205](#)
 - chi-square test [206](#)
 - correlation test [206](#)
 - in diabetes dataset [213-215](#)
 - performing [203-205](#)
 - regression test [206](#)
 - t-test [205](#)
- Singular Value Decomposition (SVD) [263](#)
- skewness [126](#)
- Sklearn [5](#)
- specific prompts [315](#)
- stacked area chart [99](#)
- standard deviation [54](#)
- standard error [166](#), [167](#)
- Standard Error of the Mean (SEM) [173](#)
- Standard Scaler [61](#)
- statistical relationships [135](#)
 - correlation [138](#)
 - covariance [136-138](#)
- statistical tests [207](#)
 - chi-square test [210](#), [211](#)
 - one-way ANOVA [211](#), [212](#)
 - t-test [208](#), [209](#)
 - two-way ANOVA [212](#), [213](#)
 - z-test [207](#), [208](#)
- statistics [5](#)
- Statsmodels [50](#)
- supervised learning [224](#)
 - fitting models to independent data [224](#), [225](#)
- Support Vector Machines (SVMs) [240](#)
 - implementing [241](#)
- survival analysis [294-299](#)

T

tail() [21](#)
t-Distributed Stochastic Neighbor Embedding (t-SNE) [265](#)
 implementing [266](#), [267](#)
term frequency-inverse document frequency (TF-IDF) [138](#)
TextBlob [45](#)
time series analysis [304](#), [305](#)
 implementing [305-309](#)
train_test_split() [35](#)
t-test [172](#), [208](#)
two-way ANOVA [212](#)
type() [23](#)

U

uniform probability distributions [149](#)
Uniform Resource Locator (URLs) [320](#)
univariate analysis [25](#), [26](#)
univariate data [25](#), [26](#)
unsupervised learning [256](#), [257](#)
 Apriori [267-269](#)
 DBSCAN [264](#)
 Eclat [270](#)
 evaluation matrices [275-278](#)
 FP-Growth [273](#), [274](#)
 Gaussian Mixture Models (GMMs) [260](#), [261](#)
 hierarchical clustering [259](#), [260](#)
 K-means clustering [257](#), [258](#)
 K-prototype clustering [258](#), [259](#)
 model selection and evaluation [275](#)
 Principal Component Analysis (PCA) [262](#)
 Singular Value Decomposition (SVD) [263](#)
 t-SNE [265-267](#)

V

value_counts() [18](#)
variance [53](#)
vectors [280](#)
Vega-altair [92](#)
violin plot [100](#)

W

Word2Vec [138](#)
word cloud [100](#)
word embeddings [252](#)

implementing [253](#)

Z

zero-shot learning [316](#), [317](#)

z-test [207](#), [208](#)