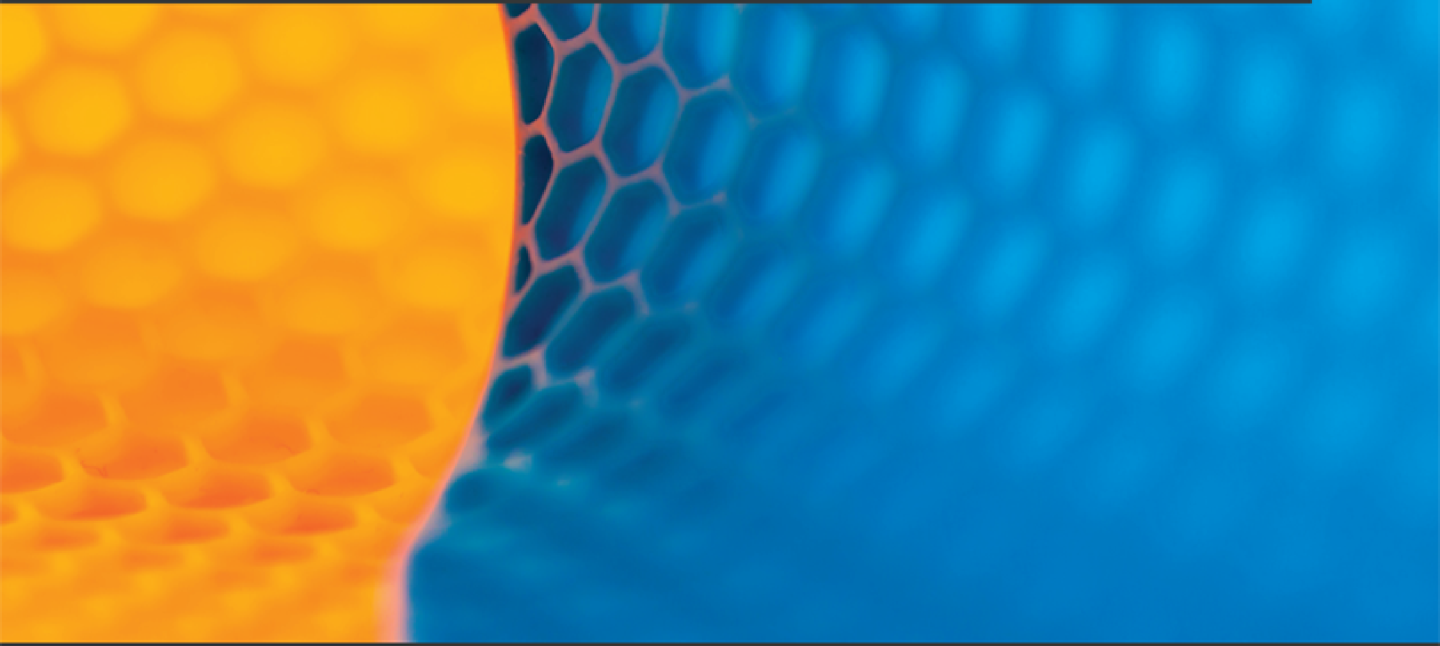
Python Data Analysis

Third Edition

Perform data collection, data processing, wrangling, visualization, and model building using Python



Avinash Navlani | Armando Fandango | Ivan Idris

Python Data Analysis
Third Edition

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Kunal Parikh
Publishing Product Manager: Ali Abidi
Content Development Editor: Joseph Sunil
Senior Editor: Roshan Kumar
Technical Editor: Sonam Pandey
Copy Editor: Safis Editing
Project Coordinator: Aishwarya Mohan
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Roshan Kawale

First published: October 2014
Second edition: March 2017
Third Edition: February 2021

Production reference: 1070121

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78995-524-8

`www.packt.com`

Contributors

About the authors

Avinash Navlani has over 8 years of experience working in data science and AI. Currently, he is working as a senior data scientist, improving products and services for customers by using advanced analytics, deploying big data analytical tools, creating and maintaining models, and onboarding compelling new datasets. Previously, he was a university lecturer, where he trained and educated people in data science subjects such as Python for analytics, data mining, machine learning, database management, and NoSQL. Avinash has been involved in research activities in data science and has been a keynote speaker at many conferences in India.

Armando Fandango creates AI-empowered products by leveraging his expertise in deep learning, machine learning, distributed computing, and computational methods and has provided thought leadership roles as the chief data scientist and director at start-ups and large enterprises. He has advised high-tech AI-based start-ups. Armando has authored books such as *Python Data Analysis - Second Edition* and *Mastering TensorFlow, Packt Publishing*. He has also published research in international journals and conferences.

Ivan Idris has an MSc in experimental physics. His graduation thesis had a strong emphasis on applied computer science. After graduating, he worked for several companies as a Java developer, data warehouse developer, and QA analyst. His main professional interests are business intelligence, big data, and cloud computing. Ivan Idris enjoys writing clean, testable code and interesting technical articles. Ivan Idris is the author of *NumPy 1.5 Beginner's Guide* and *NumPy Cookbook* by Packt Publishing. You can find more information and a blog with a few NumPy examples at ivanidris.net.

About the reviewers

Greg Walters has been involved with computers and computer programming since 1972. He is well versed in Visual Basic, Visual Basic .NET, Python, and SQL and is an accomplished user of MySQL, SQLite, Microsoft SQL Server, Oracle, C++, Delphi, Modula-2, Pascal, C, 80x86 Assembler, COBOL, and Fortran. He is a programming trainer and has trained numerous people on many pieces of computer software, including MySQL, Open Database Connectivity, Quattro Pro, Corel Draw!, Paradox, Microsoft Word, Excel, DOS, Windows 3.11, Windows for Workgroups, Windows 95, Windows NT, Windows 2000, Windows XP, and Linux. He is semi-retired and has written over 100 articles for Full Circle Magazine. He is also a musician and loves to cook. He is open to working as a freelancer on various projects.

Alistair McMaster is currently employed as a Software Engineer and Quantitative Strategist at a major financial services firm. He graduated from the University of Cambridge in 2016 with a B.A. (Hons) in Natural Sciences specializing in Astrophysics. His broader career interests include applications of data science to relationship networks and supporting social causes.

Alistair is an active contributor to pandas and a strong advocate of open-source software. In his spare time, he enjoys distance running, cycling, rock climbing, and walks with his family and friends on weekends.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Section 2: Exploratory Data Analysis and Data Cleaning

Section 3: Deep Dive into Machine Learning

Section 4: NLP, Image Analytics, and Parallel Computing

Preface

Data analysis enables you to generate value from small and big data by discovering new patterns and trends, and Python is one of the most popular tools for analyzing a wide variety of data. With this book, you'll get up and running with using Python for data analysis by exploring the different phases and methodologies used in data analysis, and you'll learn how to use modern libraries from the Python ecosystem to create efficient data pipelines.

Starting with the essential statistical and data analysis fundamentals using Python, you'll perform complex data analysis and modeling, data manipulation, data cleaning, and data visualization using easy-to-follow examples. You'll then learn how to conduct time series analysis and signal processing using ARMA models. As you advance, you'll get to grips with smart processing and data analytics using machine learning algorithms such as regression, classification, **Principal Component Analysis** (**PCA**), and clustering. In the concluding chapters, you'll work on real-world examples to analyze textual and image data using **natural language processing** (**NLP**) and image analytics techniques, respectively. Finally, the book will demonstrate parallel computing using Dask.

By the end of this data analysis book, you'll be equipped with the skills you need to prepare data for analysis and create meaningful data visualizations in order to forecast values from data.

Who this book is for

This book is for data analysts, business analysts, statisticians, and data scientists looking to learn how to use Python for data analysis. Students and academic faculties will also find this book useful for learning and teaching Python data analysis using a hands-on approach. A basic understanding of math and a working knowledge of Python will help you get started with this book.

What this book covers

`Chapter 1`, *Getting Started with Python Libraries*, explains the data analyst process and the successful installation of Python libraries and Anaconda. Also, we will discuss Jupyter Notebook and its advanced features.

`Chapter 2`, *NumPy and Pandas*, introduces NumPy and Pandas. This chapter provides a basic overview of NumPy arrays, Pandas DataFrames, and their associated functions.

`Chapter 3`, *Statistics*, gives a quick overview of descriptive and inferential statistics.

`Chapter 4`, *Linear Algebra*, gives a quick overview of linear algebra and its associated NumPy and SciPy functions.

`Chapter 5`, *Data Visualization*, introduces us to the matplotlib, seaborn, Pandas plotting, and bokeh visualization libraries.

`Chapter 6`, *Retrieving, Processing, and Storing Data*, explains how to read and write various data formats, such as CSV, Excel, JSON, HTML, and Parquet. Also, we will discuss how to acquire data from relational and NoSQL databases.

`Chapter 7`, *Cleaning Messy Data*, explains how to preprocess raw data and perform feature engineering.

`Chapter 8`, *Signal Processing and Time Series*, contains time series and signal processing examples using sales, beer production, and sunspot cycle dataset. In this chapter, we will mostly use NumPy, SciPy, and statsmodels.

`Chapter 9`, *Supervised Learning – Regression Analysis*, explains linear regression and logistic regression in detail with suitable examples using the scikit-learn library.

`Chapter 10`, *Supervised Learning – Classification Techniques*, explains various classification techniques, such as naive Bayes, decision tree, K-nearest neighbors, and SVM. Also, we will discuss model performance evaluation measures.

`Chapter 11`, *Unsupervised Learning – PCA and Clustering*, gives a detailed discussion on dimensionality reduction and clustering techniques. Also, we will evaluate the clustering performance.

`Chapter 12`, *Analyzing Textual Data*, gives a quick overview of text preprocessing, feature engineering, sentiment analysis, and text similarity. This chapter mostly uses the NLTK, SpaCy, and scikit-learn libraries.

`Chapter 13`, *Analyzing Image Data*, gives a quick overview of image processing operations using OpenCV. Also, we will discuss face detection.

`Chapter 14`, *Parallel Computing Using Dask*, explains how to perform data preprocessing and machine learning modeling in parallel using Dask.

To get the most out of this book

The execution of the code examples provided in this book requires the installation of Python 3.5 or newer on Mac OS X, Linux, or Microsoft Windows. In this book, we will frequently use SciPy, NumPy, Pandas, scikit-learn, statsmodels, matplotlib, and seaborn. Chapter 1, Getting Started with Python Libraries, provides instructions for the installation and advanced tips so that you can work smoothly. Also, the process of installing specific and additional libraries is explained in the respective chapters. Installation of Bokeh is explained in Chapter 5, Data Visualization. Similarly, the installation of NLTK and SpaCy is explained in Chapter 12, Analyzing Textual Data.

We can also install any library or package that you want to explore using the `pip` command. We need to run the following command with admin privileges:

```
$ pip install <library name>
```

We can also install it from our Jupyter Notebook with `!` (exclamation mark) before the `pip` command:

```
!pip install <library name>
```

To uninstall a Python library or package installed with `pip`, use the following command:

```
$ pip uninstall <library name>
```

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:
`https://static.packt-cdn.com/downloads/9781789955248_ColorImages.pdf`.

Conventions used

In this book, you will find a number of text styles and conventions used throughout this book. Here, we have shown some examples of these styles. Code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The other convention the `pandas` project insists on is the `import pandas as pd` import statement."

A block of code is set as follows:

```
# Creating an array
import numpy as np

a = np.array([2,4,6,8,10])

print(a)
```

Any command-line input or output is written as follows:

```
$ mkdir
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 `copyright@packt.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 `authors.packtpub.com`.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1

Section 1: Foundation for Data Analysis

The main objective of this section is to build fundamental data analysis skills for the learner. These skills involve the Jupyter Notebook, and basic Python libraries such as NumPy, Pandas, Scipy, and statsmodels. Also, this section focuses on subjective knowledge of statistics and linear algebra to build math capabilities.

This section includes the following chapters:

- `Chapter 1`, *Getting Started with Python Libraries*
- `Chapter 2`, *NumPy and pandas*
- `Chapter 3`, *Statistics*
- `Chapter 4`, *Linear Algebra*

1
Getting Started with Python Libraries

As you already know, Python has become one of the most popular, standard languages and is a complete package for data science-based operations. Python offers numerous libraries, such as NumPy, Pandas, SciPy, Scikit-Learn, Matplotlib, Seaborn, and Plotly. These libraries provide a complete ecosystem for data analysis that is used by data analysts, data scientists, and business analysts. Python also offers other features, such as flexibility, being easy to learn, faster development, a large active community, and the ability to work on complex numeric, scientific, and research applications. All these features make it the first choice for data analysis.

In this chapter, we will focus on various data analysis processes, such as KDD, SEMMA, and CRISP-DM. After this, we will provide a comparison between data analysis and data science, as well as the roles and different skillsets for data analysts and data scientists. Finally, we will shift our focus and start installing various Python libraries, IPython, Jupyter Lab, and Jupyter Notebook. We will also look at various advanced features of Jupyter Notebooks.

In this introductory chapter, we will cover the following topics:

- Understanding data analysis
- The standard process of data analysis
- The KDD process
- SEMMA
- CRISP-DM
- Comparing data analysis and data science
- The skillsets of data analysts and data scientists
- Installing Python 3
- Software used in this book
- Using IPython as a shell

- Using Jupyter Lab
- Using Jupyter Notebooks
- Advanced features of Jupyter Notebooks

Let's get started!

Understanding data analysis

The 21st century is the century of information. We are living in the age of information, which means that almost every aspect of our daily life is generating data. Not only this, but business operations, government operations, and social posts are also generating huge data. This data is accumulating day by day due to data being continually generated from business, government, scientific, engineering, health, social, climate, and environmental activities. In all these domains of decision-making, we need a systematic, generalized, effective, and flexible system for the analytical and scientific process so that we can gain insights into the data that is being generated.

In today's smart world, data analysis offers an effective decision-making process for business and government operations. Data analysis is the activity of inspecting, pre-processing, exploring, describing, and visualizing the given dataset. The main objective of the data analysis process is to discover the required information for decision-making. Data analysis offers multiple approaches, tools, and techniques, all of which can be applied to diverse domains such as business, social science, and fundamental science.

Let's look at some of the core fundamental data analysis libraries of the Python ecosystem:

- **NumPy**: This is a short form of numerical Python. It is the most powerful scientific library available in Python for handling multidimensional arrays, matrices, and methods in order to compute mathematics efficiently.
- **SciPy**: This is also a powerful scientific computing library for performing scientific, mathematical, and engineering operations.
- **Pandas**: This is a data exploration and manipulation library that offers tabular data structures such as DataFrames and various methods for data analysis and manipulation.
- **Scikit-learn**: This stands for "Scientific Toolkit for Machine learning". It is a machine learning library that offers a variety of supervised and unsupervised algorithms, such as regression, classification, dimensionality reduction, cluster analysis, and anomaly detection.

- **Matplotlib**: This is a core data visualization library and is the base library for all other visualization libraries in Python. It offers 2D and 3D plots, graphs, charts, and figures for data exploration. It runs on top of NumPy and SciPy.
- **Seaborn**: This is based on Matplotlib and offers easy to draw, high-level, interactive, and more organized plots.
- **Plotly**: Plotly is a data visualization library. It offers high quality and interactive graphs, such as scatter charts, line charts, bar charts, histograms, boxplots, heatmaps, and subplots.

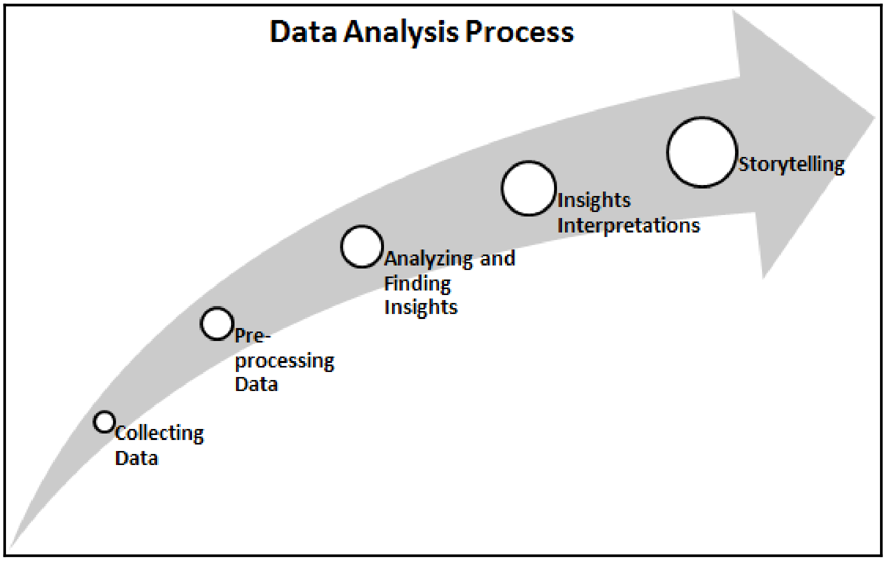
Installation instructions for the required libraries and software will be provided throughout this book when they're needed. In the meantime, let's discuss various data analysis processes, such as the standard process, KDD, SEMMA, and CRISP-DM.

The standard process of data analysis

Data analysis refers to investigating the data, finding meaningful insights from it, and drawing conclusions. The main goal of this process is to collect, filter, clean, transform, explore, describe, visualize, and communicate the insights from this data to discover decision-making information. Generally, the data analysis process is comprised of the following phases:

1. **Collecting Data**: Collect and gather data from several sources.
2. **Preprocessing Data**: Filter, clean, and transform the data into the required format.
3. **Analyzing and Finding Insights**: Explore, describe, and visualize the data and find insights and conclusions.
4. **Insights Interpretations**: Understand the insights and find the impact each variable has on the system.
5. **Storytelling**: Communicate your results in the form of a story so that a layman can understand them.

We can summarize these steps of the data analysis process via the following process diagram:



In this section, we have covered the standard data analysis process, which emphasizes finding interpretable insights and converting them into a user story. In the next section, we will discuss the KDD process.

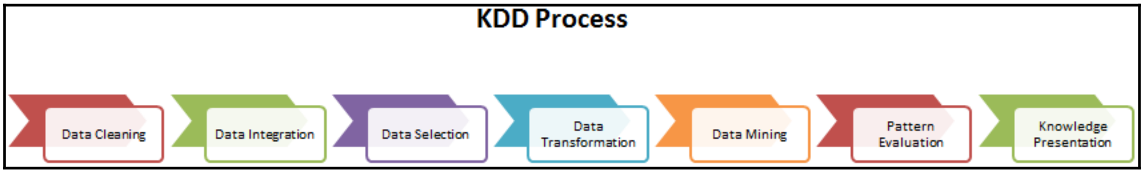
The KDD process

The **KDD** acronym stands for **knowledge discovery from data** or **Knowledge Discovery in Databases**. Many people treat KDD as one synonym for data mining. Data mining is referred to as the knowledge discovery process of interesting patterns. The main objective of KDD is to extract or discover hidden interesting patterns from large databases, data warehouses, and other web and information repositories. The KDD process has seven major phases:

1. **Data Cleaning**: In this first phase, data is preprocessed. Here, noise is removed, missing values are handled, and outliers are detected.
2. **Data Integration**: In this phase, data from different sources is combined and integrated together using data migration and ETL tools.
3. **Data Selection**: In this phase, relevant data for the analysis task is recollected.

4. **Data Transformation**: In this phase, data is engineered in the required appropriate form for analysis.
5. **Data Mining**: In this phase, data mining techniques are used to discover useful and unknown patterns.
6. **Pattern Evaluation**: In this phase, the extracted patterns are evaluated.
7. **Knowledge Presentation**: After pattern evaluation, the extracted knowledge needs to be visualized and presented to business people for decision-making purposes.

The complete KDD process is shown in the following diagram:



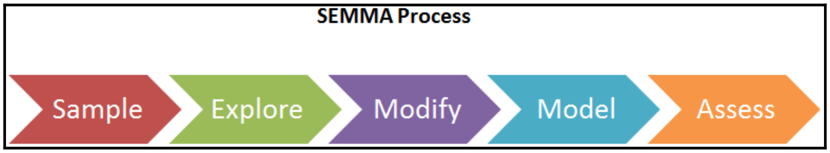
KDD is an iterative process for enhancing data quality, integration, and transformation to get a more improved system. Now, let's discuss the SEMMA process.

SEMMA

The **SEMMA** acronym's full form is **Sample**, **Explore**, **Modify**, **Model**, and **Assess**. This sequential data mining process is developed by SAS. The SEMMA process has five major phases:

1. **Sample**: In this phase, we identify different databases and merge them. After this, we select the data sample that's sufficient for the modeling process.
2. **Explore**: In this phase, we understand the data, discover the relationships among variables, visualize the data, and get initial interpretations.
3. **Modify**: In this phase, data is prepared for modeling. This phase involves dealing with missing values, detecting outliers, transforming features, and creating new additional features.
4. **Model**: In this phase, the main concern is selecting and applying different modeling techniques, such as linear and logistic regression, backpropagation networks, KNN, support vector machines, decision trees, and Random Forest.
5. **Assess**: In this last phase, the predictive models that have been developed are evaluated using performance evaluation measures.

The following diagram shows this process:



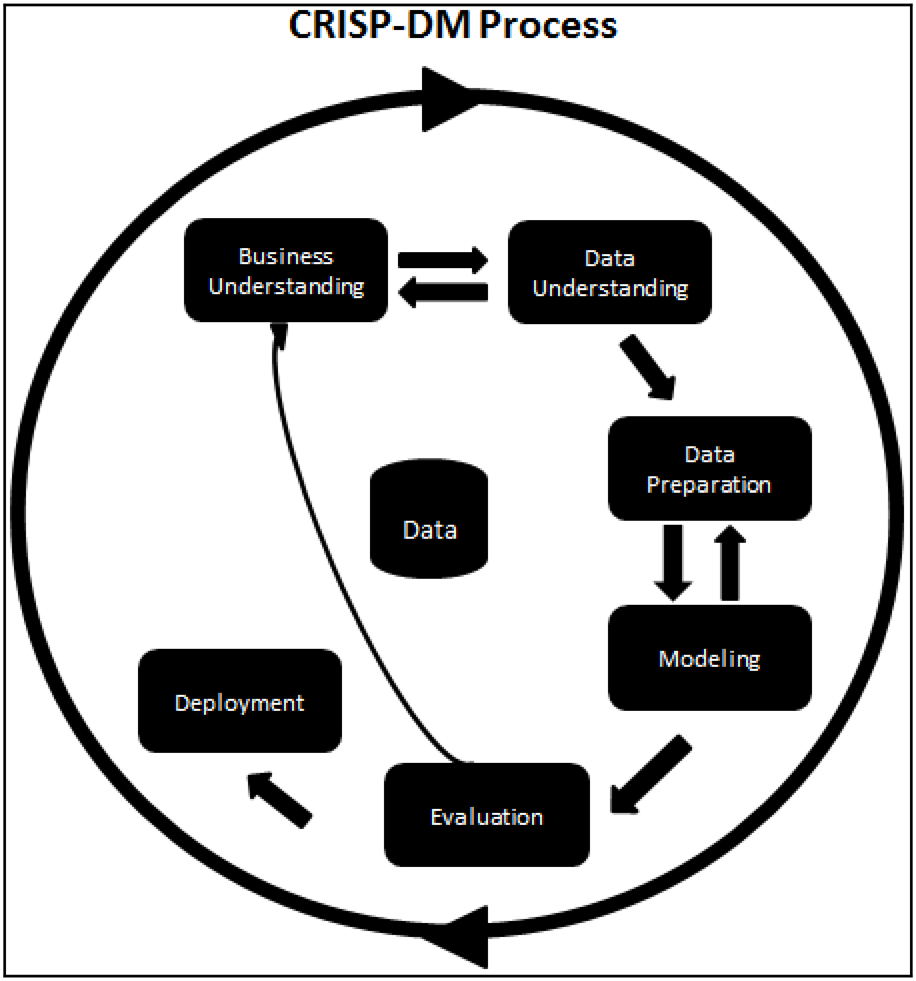
The preceding diagram shows the steps involved in the SEMMA process. SEMMA emphasizes model building and assessment. Now, let's discuss the CRISP-DM process.

CRISP-DM

CRISP-DM's full form is **CRoss-InduStry Process for Data Mining**. CRISP-DM is a well-defined, well-structured, and well-proven process for machine learning, data mining, and business intelligence projects. It is a robust, flexible, cyclic, useful, and practical approach to solving business problems. The process discovers hidden valuable information or patterns from several databases. The CRISP-DM process has six major phases:

1. **Business Understanding**: In this first phase, the main objective is to understand the business scenario and requirements for designing an analytical goal and initial action plan.
2. **Data Understanding:** In this phase, the main objective is to understand the data and its collection process, perform data quality checks, and gain initial insights.
3. **Data Preparation**: In this phase, the main objective is to prepare analytics-ready data. This involves handling missing values, outlier detection and handling, normalizing data, and feature engineering. This phase is the most time-consuming for data scientists/analysts.
4. **Modeling**: This is the most exciting phase of the whole process since this is where you design the model for prediction purposes. First, the analyst needs to decide on the modeling technique and develop models based on data.
5. **Evaluation:** Once the model has been developed, it's time to assess and test the model's performance on validation and test data using model evaluation measures such as MSE, RMSE, R-Square for regression and accuracy, precision, recall, and the F1-measure.
6. **Deployment**: In this final phase, the model that was chosen in the previous step will be deployed to the production environment. This requires a team effort from data scientists, software developers, DevOps experts, and business professionals.

The following diagram shows the full cycle of the CRISP-DM process:



The standard process focuses on discovering insights and making interpretations in the form of a story, while KDD focuses on data-driven pattern discovery and visualizing this. SEMMA majorly focuses on model building tasks, while CRISP-DM focuses on business understanding and deployment. Now that we know about some of the processes surrounding data analysis, let's compare data analysis and data science to find out how they are related, as well as what makes them different from one other.

Comparing data analysis and data science

Data analysis is the process in which data is explored in order to discover patterns that help us make business decisions. It is one of the subdomains of data science. Data analysis methods and tools are widely utilized in several business domains by business analysts, data scientists, and researchers. Its main objective is to improve productivity and profits. Data analysis extracts and queries data from different sources, performs exploratory data analysis, visualizes data, prepares reports, and presents it to the business decision-making authorities.

On the other hand, data science is an interdisciplinary area that uses a scientific approach to extract insights from structured and unstructured data. Data science is a union of all terms, including data analytics, data mining, machine learning, and other related domains. Data science is not only limited to exploratory data analysis and is used for developing models and prediction algorithms such as stock price, weather, disease, fraud forecasts, and recommendations such as movie, book, and music recommendations.

The roles of data analysts and data scientists

A data analyst collects, filters, processes, and applies the required statistical concepts to capture patterns, trends, and insights from data and prepare reports for making decisions. The main objective of the data analyst is to help companies solve business problems using discovered patterns and trends. The data analyst also assesses the quality of the data and handles the issues concerning data acquisition. A data analyst should be proficient in writing SQL queries, finding patterns, using visualization tools, and using reporting tools Microsoft Power BI, IBM Cognos, Tableau, QlikView, Oracle BI, and more.

Data scientists are more technical and mathematical than data analysts. Data scientists are research- and academic-oriented, whereas data analysts are more application-oriented. Data scientists are expected to predict a future event, whereas data analysts extract significant insights out of data. Data scientists develop their own questions, while data analysts find answers to given questions. Finally, data scientists focus on **what is going to happen**, whereas data analysts focus on **what has happened so far**. We can summarize these two roles using the following table:

Features	**Data Scientist**	**Data Analyst**
Background	Predict future events and scenarios based on data	Discover meaningful insights from the data.
Role	Formulate questions that can profit the business	Solve the business questions to make decisions.

Type of data	Work on both structured and unstructured data	Only work on structured data
Programming	Advanced programming	Basic programming
Skillset	Knowledge of statistics, machine learning algorithms, NLP, and deep learning	Knowledge of statistics, SQL, and data visualization
Tools	R, Python, SAS, Hadoop, Spark, TensorFlow, and Keras	Excel, SQL, R, Tableau, and QlikView

Now that we know what defines a data analyst and data scientist, as well as how they are different from each other, let's have a look at the various skills that you would need to become one of them.

The skillsets of data analysts and data scientists

A data analyst is someone who discovers insights from data and creates value out of it. This helps decision-makers understand how the business is performing. Data analysts must acquire the following skills:

- **Exploratory Data Analysis (EDA)**: EDA is an essential skill for data analysts. It helps with inspecting data to discover patterns, test hypotheses, and assure assumptions.
- **Relational Database**: Knowledge of at least one of the relational database tools, such as MySQL or Postgre, is mandatory. SQL is a must for working on relational databases.
- **Visualization and BI Tools**: A picture speaks more than words. Visuals have more of an impact on humans and visuals are a clear and easy option for representing the insights. Visualization and BI tools such as Tableau, QlikView, MS Power BI, and IBM Cognos can help analysts visualize and prepare reports.
- **Spreadsheet**: Knowledge of MS Excel, WPS, Libra, or Google Sheets is mandatory for storing and managing data in tabular form.
- **Storytelling and Presentation Skills**: The art of storytelling is another necessary skill. A data analyst should be an expert in connecting data facts to an idea or an incident and turning it into a story.

On the other hand, the primary job of a data scientist is to solve problems using data. In order to do this, they need to understand the client's requirements, their domain, their problem space, and ensure that they get exactly what they really want. The tasks that data scientists undertake vary from company to company. Some companies use data analysts and offer the title of data scientist just to glorify the job designation. Some combine data analyst tasks with data engineers and offer data scientists designation; others assign them to machine learning-intensive tasks with data visualizations.

The task of the data scientist varies, depending on the company. Some employ data scientists as well-known data analysts and combine their responsibilities with data engineers. Others give them the task of performing intensive data visualization on machines.

A data scientist has to be a jack of all trades and wear multiple hats, including those of a data analyst, statistician, mathematician, programmer, ML, or NLP engineer. Most people are not skilled enough or experts in all these trades. Also, getting skilled enough requires lots of effort and patience. This is why data science cannot be learned in 3 or 6 months. Learning data science is a journey. A data scientist should have a wide variety of skills, such as the following:

- **Mathematics and Statistics**: Most machine learning algorithms are based on mathematics and statistics. Knowledge of mathematics helps data scientists develop custom solutions.
- **Databases**: Knowledge of SQL allows data scientists to interact with the database and collect the data for prediction and recommendation.
- **Machine Learning**: Knowledge of supervised machine learning techniques such as regression analysis, classification techniques, and unsupervised machine learning techniques such as cluster analysis, outlier detection, and dimensionality reduction.
- **Programming Skills**: Knowledge of programming helps data scientists automate their suggested solutions. Knowledge of Python and R is recommended.
- **Storytelling and Presentation skills**: Communicating the results in the form of storytelling via PowerPoint presentations.
- **Big Data Technology**: Knowledge of big data platforms such as Hadoop and Spark helps data scientists develop big data solutions for large-scale enterprises.
- **Deep Learning Tools**: Deep learning tools such as Tensorflow and Keras are utilized in NLP and image analytics.

Apart from these skillsets, knowledge of web scraping packages/tools for extracting data from diverse sources and web application frameworks such as Flask or Django for designing prototype solutions is also obtained. It is all about the skillset for data science professionals.

Now that we have covered the basics of data analysis and data science, let's dive into the basic setup needed to get started with data analysis. In the next section, we'll learn how to install Python.

Installing Python 3

The installer file for installing Python 3 can easily be downloaded from the official website (`https://www.python.org/downloads/`) for Windows, Linux, and Mac 32-bit or 64-bit systems. The installer can be installed by double-clicking on it. This installer also has an IDE named "IDLE" that can be used for development. We will dive deeper into each of the operating systems in the next few sections.

Python installation and setup on Windows

This book is based on the latest Python 3 version. All the code that will be used in this book is written in Python 3, so we need to install Python 3 before we can start coding. Python is an open source, distributed, and freely available language. It is also licensed for commercial use. There are many implementations of Python, including commercial implementations and distributions. In this book, we will focus on the standard Python implementation, which is guaranteed to be compatible with NumPy.

> You can download Python 3.9.x from the Python official website: `https://www.python.org/downloads/`. Here, you can find installation files for Windows, Linux, Mac OS X, and other OS platforms. You can find instructions for installing and using Python for various operating systems at `https://docs.python.org/3.7/using/index.html`.

You need to have Python 3.5.x or above installed on your system. The sunset date for Python 2.7 was moved from 2015 to 2020, but at the time of writing, Python 2.7 will not be supported and maintained by the Python community.

At the time of writing this book, we had Python 3.8.3 installed as a prerequisite on our Windows 10 virtual machine: `https://www.python.org/ftp/python/3.8.3/python-3.8.3.exe`.

Python installation and setup on Linux

Installing Python on Linux is significantly easier compared to the other OSes. To install the foundational libraries, run the following command-line instruction:

```
$ pip3 install numpy scipy pandas matplotlib jupyter notebook
```

It may be essential to run the `sudo` command before the preceding command if you don't have sufficient rights on the machine that you are using.

Python installation and setup on Mac OS X with a GUI installer

Python can be installed via the installation file from the Python official website. The installer file can be downloaded from its official web page (`https://www.python.org/downloads/mac-osx/`) for macOS. This installer also has an IDE named "IDLE" that can be used for development.

Python installation and setup on Mac OS X with brew

For Mac systems, you can use the Homebrew package manager to install Python. It will make it easier to install the required applications for developers, researchers, and scientists. The `brew install` command is used to install another application, such as installing python3 or any other Python package, such as NLTK or SpaCy.

To install the most recent version of Python, you need to execute the following command in a Terminal:

```
$ brew install python3
```

After installation, you can confirm the version of Python you've installed by running the following command:

```
$ python3 --version
Python 3.7.4
```

You can also open the Python Shell from the command line by running the following command:

```
$ python3
```

Now that we know how to install Python on our system, let's dive into the actual tools that we will need to start data analysis.

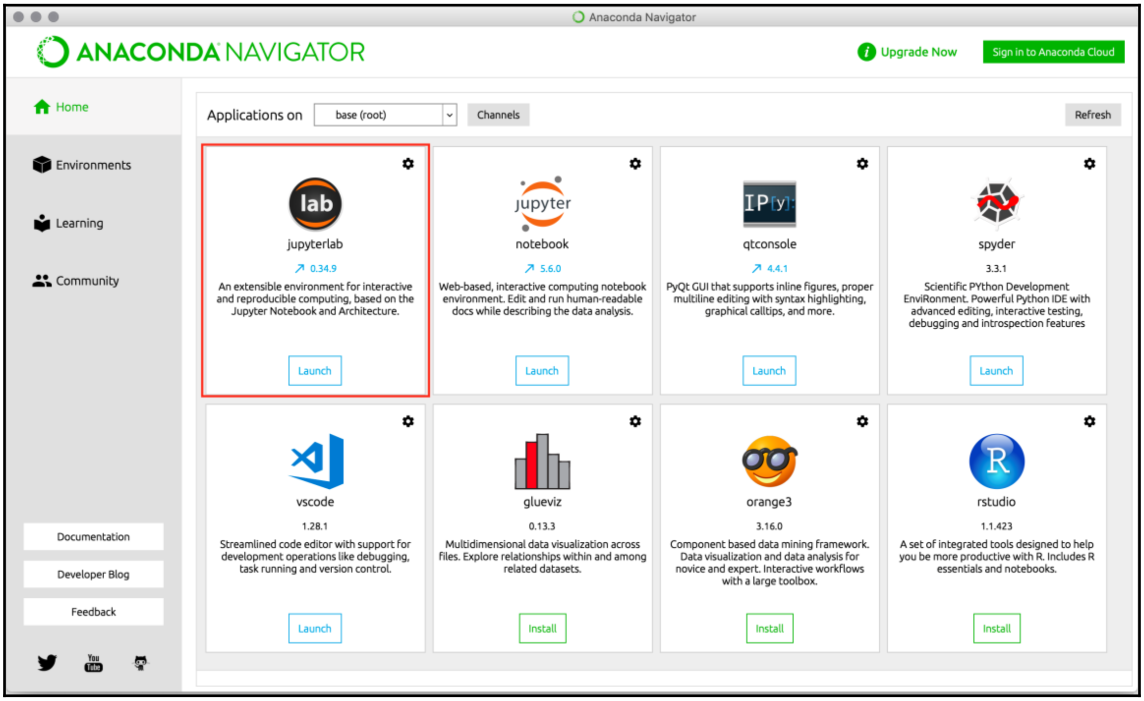
Software used in this book

Let's discuss the software that will be used in this book. In this book, we are going to use Anaconda IDE to analyze data. Before installing it, let's understand what Anaconda is.

A Python program can easily run on any system that has it installed. We can write a program on a Notepad and run it on the command prompt. We can also write and run Python programs on different IDEs, such as Jupyter Notebook, Spyder, and PyCharm. Anaconda is a freely available open source package containing various data manipulation IDEs and several packages such as NumPy, SciPy, Pandas, Scikit-learn, and so on for data analysis purposes. Anaconda can easily be downloaded and installed, as follows:

1. Download the installer from `https://www.anaconda.com/distribution/`.
2. Select the operating system that you are using.
3. From the Python 3.7 section, select the 32-bit or 64-bit installer option and start downloading.
4. Run the installer by double-clicking on it.
5. Once the installation is complete, check your program in the **Start** menu or search for Anaconda in the **Start** menu.

Anaconda also has an Anaconda Navigator, which is a desktop GUI application that can be used to launch applications such as Jupyter Notebook, Spyder, Rstudio, Visual Studio Code, and JupyterLab:



An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

Streamlined code editor with support for development operations like debugging, task running and version control.

Multidimensional data visualization across files. Explore relationships within and among related datasets.

Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

Now, let's look at IPython, a shell-based computing environment for data analysis.

Using IPython as a shell

IPython is an interactive shell that is equivalent to an interactive computing environment such as Matlab or Mathematica. This interactive shell was created for the purpose of quick experimentation. It is a very useful tool for data professionals that are performing small experiments.

IPython shell offers the following features:

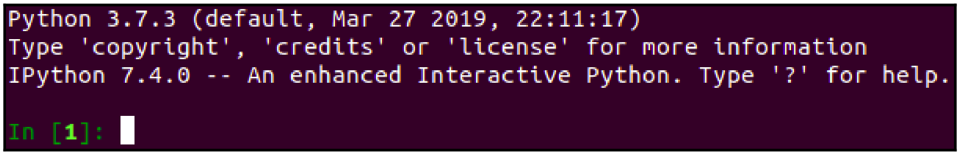
- Easy access to system commands.
- Easy editing of inline commands.
- Tab completion, which helps you find commands and speed up your task.

- Command History, which helps you view previously used commands.
- Easily execute external Python scripts.
- Easy debugging with the Python debugger.

Now, let's execute some commands on IPython. To start IPython, use the following command on the command line:

```
$ ipython3
```

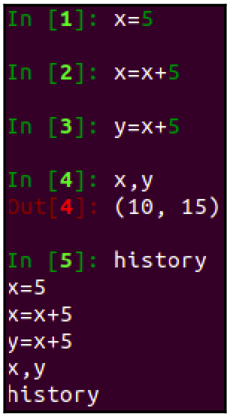
When you run the preceding command, the following window will appear:



```
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 
```

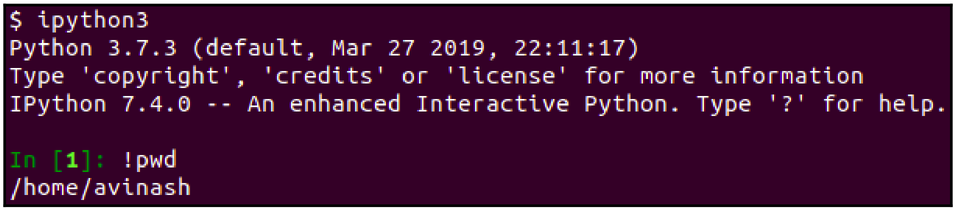
Now, let's understand and execute some commands that the IPython shell provides:

- **History Commands:** The `history` command used to check the list of previously used commands. The following screenshot shows how to use the `history` command in IPython:



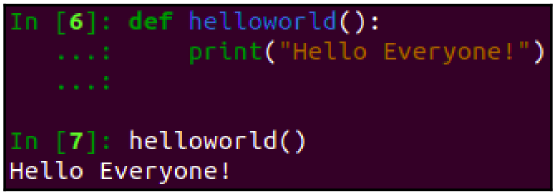
```
In [1]: x=5

In [2]: x=x+5

In [3]: y=x+5

In [4]: x,y
Out[4]: (10, 15)

In [5]: history
x=5
x=x+5
y=x+5
x,y
history
```

- **System Commands:** We can also run system commands from IPython using the exclamation sign (`!`). Here, the input command after the exclamation sign is considered a system command. For example, `!date` will display the current date of the system, while `!pwd` will show the current working directory:



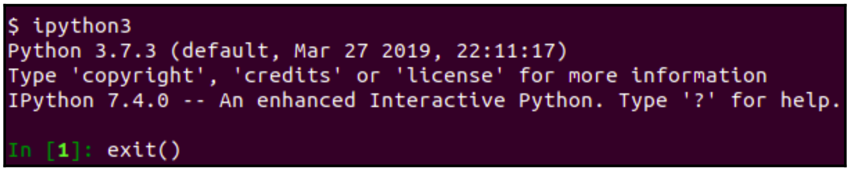
```
$ ipython3
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: !pwd
/home/avinash
```

- **Writing Function:** We can write functions as we would write them in any IDE, such as Jupyter Notebook, Python IDLE, PyCharm, or Spyder. Let's look at an example of a function:



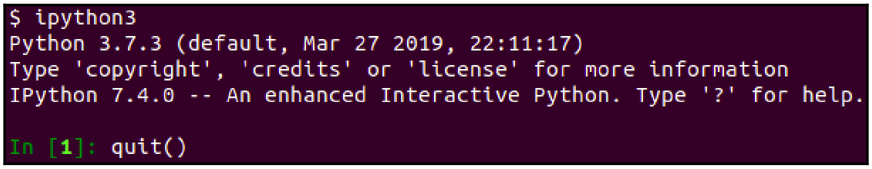
```
In [6]: def helloworld():
   ...:     print("Hello Everyone!")
   ...:

In [7]: helloworld()
Hello Everyone!
```

- **Quit Ipython Shell:** You can exit or quit the IPython shell using `quit()` or `exit()` or *CTRL* + *D*:



```
$ ipython3
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: exit()
```

You can also quit the IPython shell using the `quit()` command:



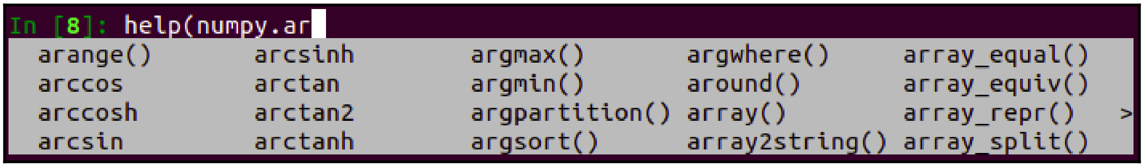
```
$ ipython3
Python 3.7.3 (default, Mar 27 2019, 22:11:17)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: quit()
```

In this subsection, we have looked at a few basic commands we can use on the IPython shell. Now, let's discuss how we can use the `help` command in the IPython shell.

Reading manual pages

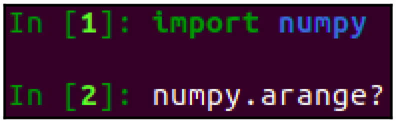
In the IPython shell, we can open a list of available commands using the `help` command. It is not compulsory to write the full name of the function. You can just type in a few initial characters and then press the *tab* button, and it will find the word you are looking for. For example, let's use the `arrange()` function. There are two ways we can find help about functions:

- **Use the help function:** Let's type `help` and write a few initial characters of the function. After that, press the *tab* key, select a function using the arrow keys, and press the *Enter* key:



```
In [8]: help(numpy.ar
  arange()      arcsinh       argmax()        argwhere()      array_equal()
  arccos        arctan        argmin()        around()        array_equiv()
  arccosh       arctan2       argpartition()  array()         array_repr()    >
  arcsin        arctanh       argsort()       array2string()  array_split()
```

- **Use a question mark:** We can also use a question mark after the name of the function. The following screenshot shows an example of this:



```
In [1]: import numpy

In [2]: numpy.arange?
```

In this subsection, we looked at the help and question mark support that's provided for module functions. We can also get help from library documentation. Let's discuss how to get documentation for data analysis in Python libraries.

Where to find help and references to Python data analysis libraries

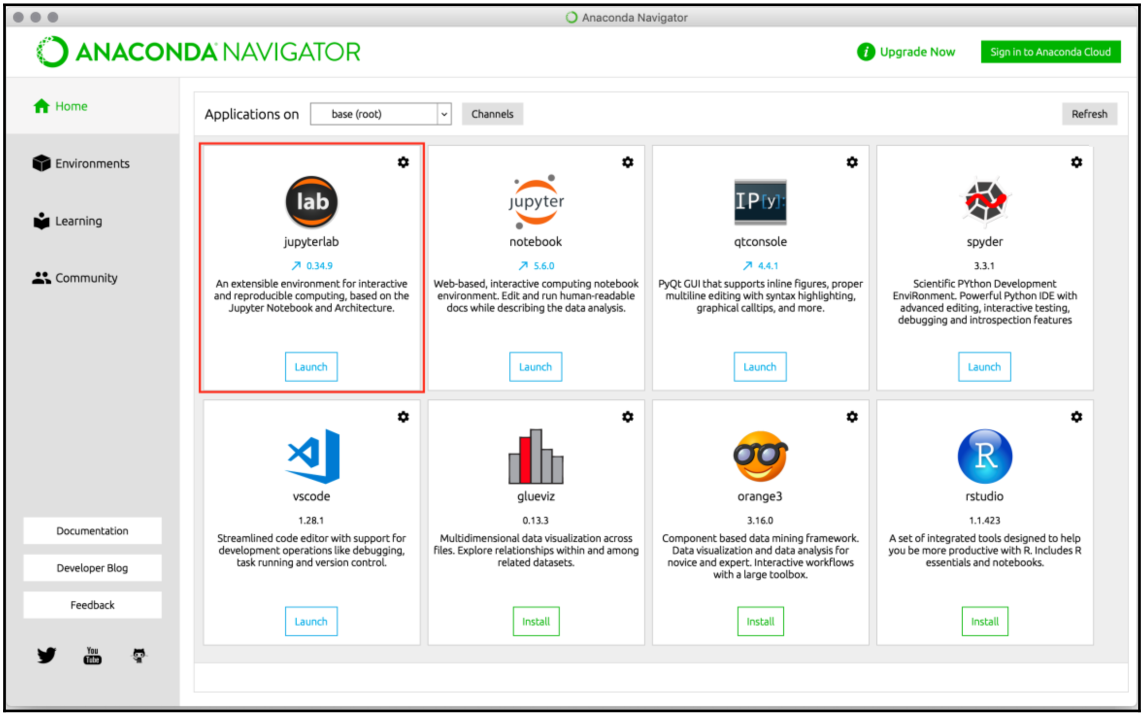
The following table lists the documentation websites for the Python data analysis libraries we have discussed in this chapter:

Packages/Software	**Description**
NumPy	`https://numpy.org/doc/`
SciPy	`https://docs.scipy.org/doc/`
Pandas	`https://pandas.pydata.org/docs/`
Matplotlib	`https://matplotlib.org/3.2.1/contents.html`
Seaborn	`https://seaborn.pydata.org/`
Scikit-learn	`https://scikit-learn.org/stable/`
Anaconda	`https://www.anaconda.com/distribution/`

You can also find answers to various Python programming questions related to NumPy, SciPy, Pandas, Matplotlib, Seaborn, and Scikit-learn on the StackOverflow platform. You can also raise issues related to the aforementioned libraries on GitHub.

Using JupyterLab

JupyterLab is a next-generation web-based user interface. It offers a combination of data analysis and machine learning product development tools such as a Text Editor, Notebooks, Code Consoles, and Terminals. It's a flexible and powerful tool that should be a part of any data analyst's toolkit:



jupyterlab
0.34.9
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.

notebook
5.6.0
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.

qtconsole
4.4.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.

spyder
3.3.1
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features

vscode
1.28.1
Streamlined code editor with support for development operations like debugging, task running and version control.

glueviz
0.13.3
Multidimensional data visualization across files. Explore relationships within and among related datasets.

orange3
3.16.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.

rstudio
1.1.423
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

You can install JupyterLab using `conda`, `pip`, or `pipenv`.

To install using `conda`, we can use the following command:

```
$ conda install -c conda-forge jupyterlab
```

To install using `pip`, we can use the following command:

```
$ pip install jupyterlab
```

To install using `pipenv`, we can use the following command:

```
$ pipenv install jupyterlab
```

In this section, we have learned how to install Jupyter Lab. In the next section, we will focus on Jupyter Notebooks.

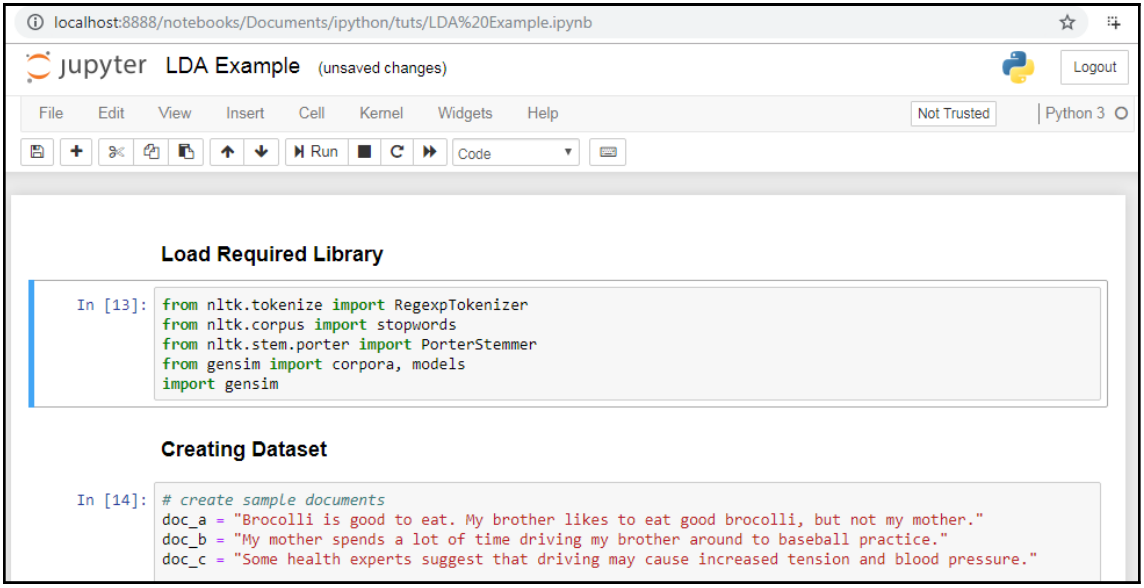
Using Jupyter Notebooks

Jupyter Notebook is a web application that's used to create data analysis notebooks that contain code, text, figures, links, mathematical equations, and charts. Recently, the community introduced the next generation of web-based Jupyter Notebooks, called JupyterLab. You can take a look at these notebook collections at the following links:

- `https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks`
- `https://nbviewer.jupyter.org/`

Often, these notebooks are used as educational tools or to demonstrate Python software. We can import or export notebooks either from plain Python code or from the special notebook format. The notebooks can be run locally, or we can make them available online by running a dedicated notebook server. Certain cloud computing solutions, such as Wakari, PiCloud, and Google Colaboratory, allow you to run notebooks in the cloud.

"Jupyter" is an acronym that stands for Julia, Python, and R. Initially, the developers implemented it for these three languages, but now, it is used for various other languages, including C, C++, Scala, Perl, Go, PySpark, and Haskell:



Load Required Library

```
In [13]: from nltk.tokenize import RegexpTokenizer
         from nltk.corpus import stopwords
         from nltk.stem.porter import PorterStemmer
         from gensim import corpora, models
         import gensim
```

Creating Dataset

```
In [14]: # create sample documents
         doc_a = "Brocolli is good to eat. My brother likes to eat good brocolli, but not my mother."
         doc_b = "My mother spends a lot of time driving my brother around to baseball practice."
         doc_c = "Some health experts suggest that driving may cause increased tension and blood pressure."
```

Jupyter Notebook offers the following features:

- It has the ability to edit code in the browser with proper indentation.
- It has the ability to execute code from the browser.
- It has the ability to display output in the browser.
- It can render graphs, images, and videos in cell output.
- It has the ability to export code in PDF, HTML, Python file, and LaTex format.

We can also use both Python 2 and 3 in Jupyter Notebooks by running the following commands in the Anaconda prompt:

```
# For Python 2.7
conda create -n py27 python=2.7 ipykernel

# For Python 3.5
conda create -n py35 python=3.5 ipykernel
```

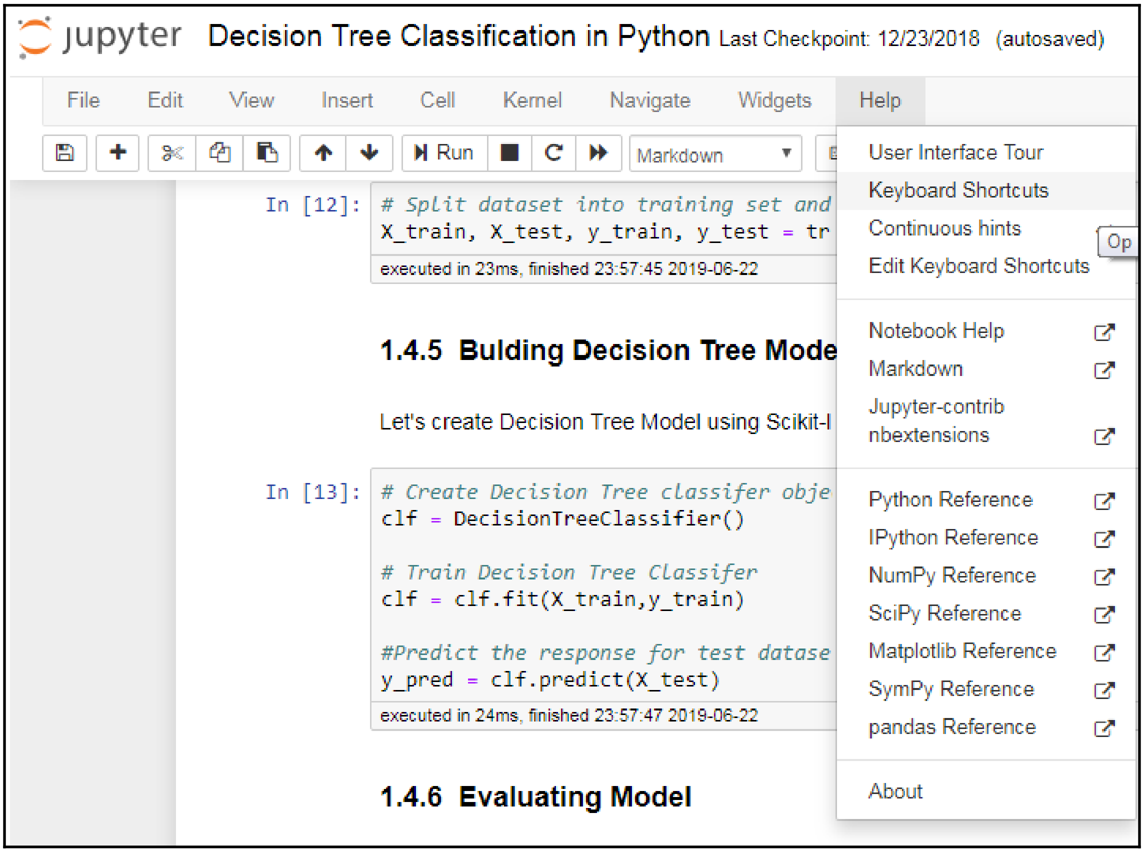
Now that we now about various tools and libraries and also have installed Python, let's move on to some of the advanced features in the most commonly used tool, Jupyter Notebooks.

Advanced features of Jupyter Notebooks

Jupyter Notebook offers various advanced features, such as keyboard shortcuts, installing other kernels, executing shell commands, and using various extensions for faster data analysis operations. Let's get started and understand these features one by one.

Keyboard shortcuts

Users can find all the shortcut commands that can be used inside Jupyter Notebook by selecting the **Keyboard Shortcuts** option in the **Help** menu or by using the *Cmd* + *Shift* + *P* shortcut key. This will make the quick select bar appear, which contains all the shortcuts commands, along with a brief description of each. It is easy to use the bar and users can use it when they forget something:



In [12]:

```
# Split dataset into training set and
X_train, X_test, y_train, y_test = tr
```

executed in 23ms, finished 23:57:45 2019-06-22

1.4.5 Bulding Decision Tree Mode

Let's create Decision Tree Model using Scikit-l

In [13]:

```
# Create Decision Tree classifer obje
clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test datase
y_pred = clf.predict(X_test)
```

executed in 24ms, finished 23:57:47 2019-06-22

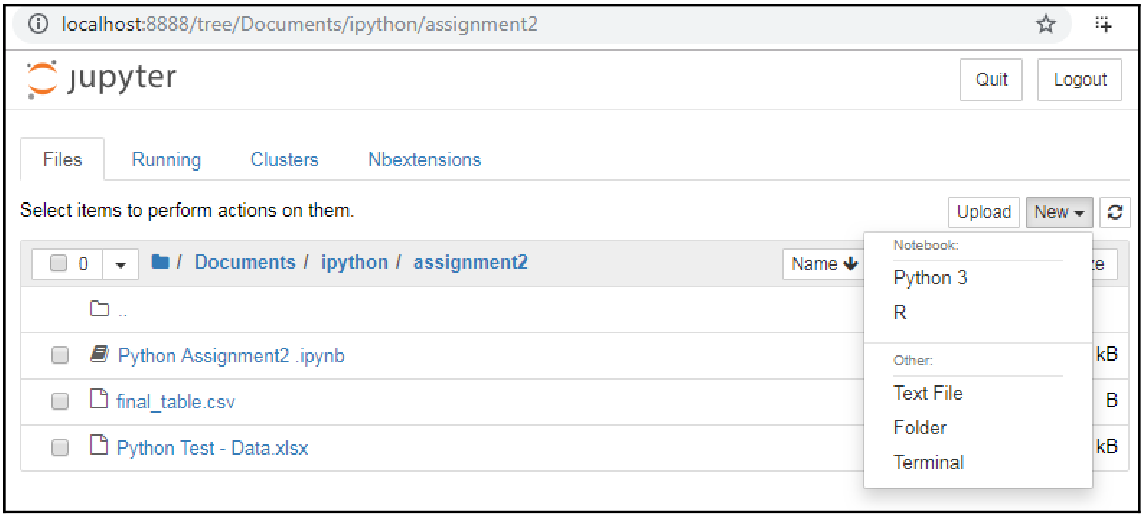
1.4.6 Evaluating Model

Installing other kernels

Jupyter has the ability to run multiple kernels for different languages. It is very easy to set up an environment for a particular language in Anaconda. For example, an R kernel can be set by using the following command in Anaconda:

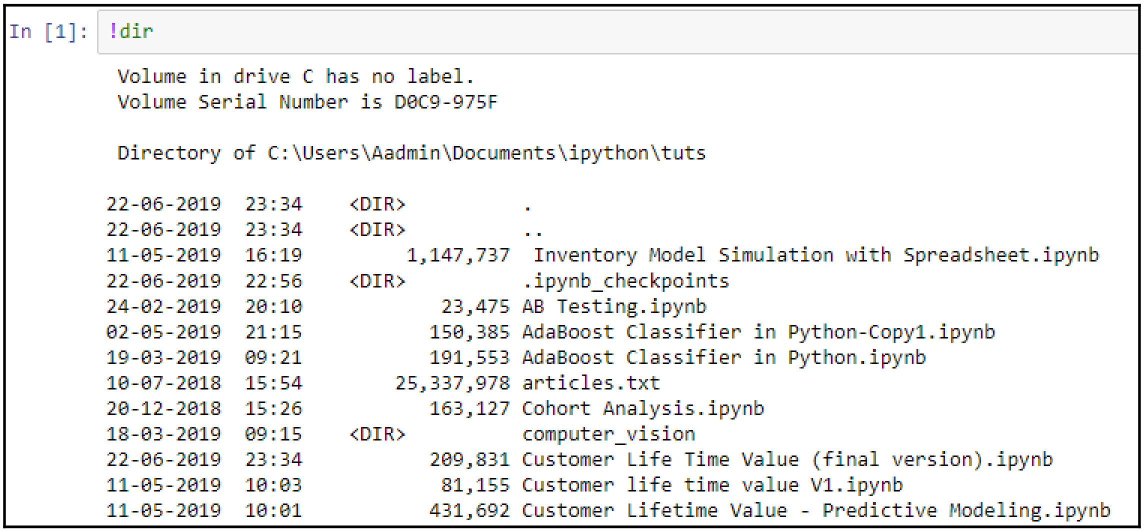
```
$ conda install -c r r-essentials
```

The R kernel should then appear, as shown in the following screenshot:



Running shell commands

In Jupyter Notebook, users can run shell commands for Unix and Windows. The shell offers a communication interface for talking with the computer. The user needs to put `!` (an exclamation sign) before running any command:



```
In [1]: !dir

         Volume in drive C has no label.
         Volume Serial Number is D0C9-975F

         Directory of C:\Users\Aadmin\Documents\ipython\tuts

        22-06-2019  23:34    <DIR>          .
        22-06-2019  23:34    <DIR>          ..
        11-05-2019  16:19         1,147,737  Inventory Model Simulation with Spreadsheet.ipynb
        22-06-2019  22:56    <DIR>          .ipynb_checkpoints
        24-02-2019  20:10            23,475 AB Testing.ipynb
        02-05-2019  21:15           150,385 AdaBoost Classifier in Python-Copy1.ipynb
        19-03-2019  09:21           191,553 AdaBoost Classifier in Python.ipynb
        10-07-2018  15:54        25,337,978 articles.txt
        20-12-2018  15:26           163,127 Cohort Analysis.ipynb
        18-03-2019  09:15    <DIR>          computer_vision
        22-06-2019  23:34           209,831 Customer Life Time Value (final version).ipynb
        11-05-2019  10:03            81,155 Customer life time value V1.ipynb
        11-05-2019  10:01           431,692 Customer Lifetime Value - Predictive Modeling.ipynb
```

Extensions for Notebook

Notebook extensions (or nbextensions) add more features compared to basic Jupyter Notebooks. These extensions improve the user's experience and interface. Users can easily select any of the extensions by selecting the **NBextensions** tab.

To install `nbextension` in Jupyter Notebook using `conda`, run the following command:

```
conda install -c conda-forge jupyter_nbextensions_configurator
```

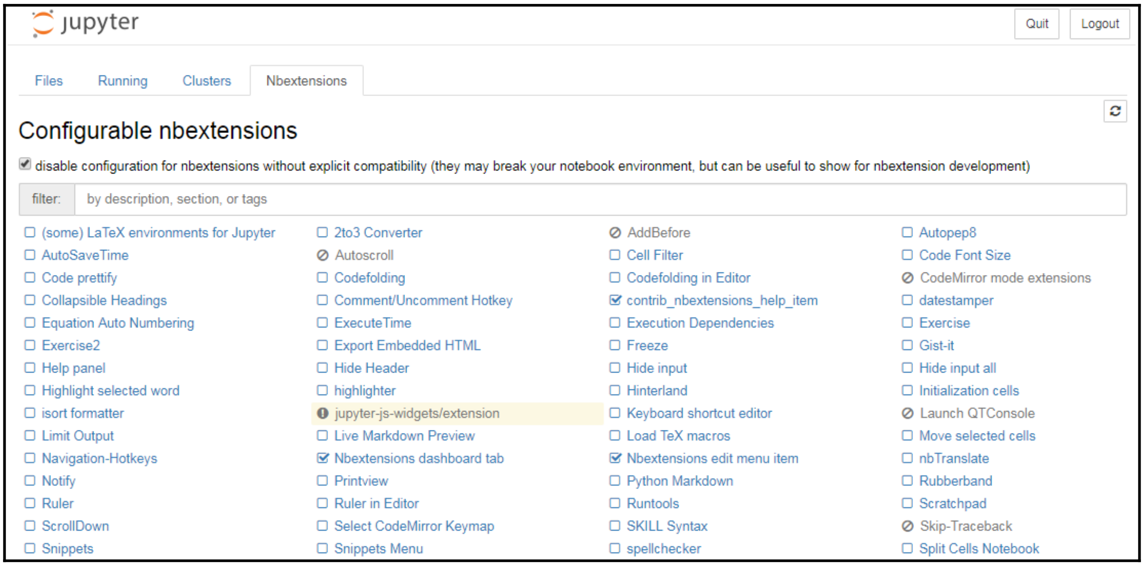
To install `nbextension` in Jupyter Notebook using `pip`, run the following command:

```
pip install jupyter_contrib_nbextensions && jupyter contrib nbextension
install
```

If you get permission errors on macOS, just run the following command:

```
pip install jupyter_contrib_nbextensions && jupyter contrib nbextension
install --user
```

All the configurable nbextensions will be shown in a different tab, as shown in the following screenshot:



Configurable nbextensions

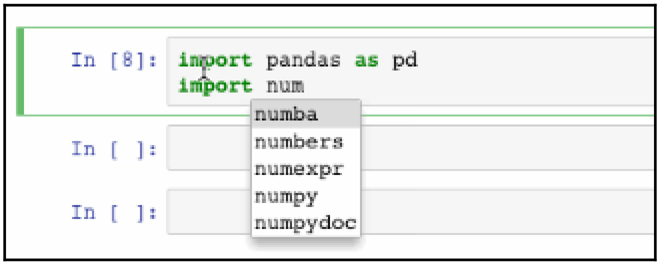
☑ disable configuration for nbextensions without explicit compatibility (they may break your notebook environment, but can be useful to show for nbextension development)

filter: by description, section, or tags

□ (some) LaTeX environments for Jupyter
□ AutoSaveTime
□ Code prettify
□ Collapsible Headings
□ Equation Auto Numbering
□ Exercise2
□ Help panel
□ Highlight selected word
□ isort formatter
□ Limit Output
□ Navigation-Hotkeys
□ Notify
□ Ruler
□ ScrollDown
□ Snippets
□ 2to3 Converter
⊘ Autoscroll
□ Codefolding
□ Comment/Uncomment Hotkey
□ ExecuteTime
□ Export Embedded HTML
□ Hide Header
□ highlighter
❗ jupyter-js-widgets/extension
□ Live Markdown Preview
☑ Nbextensions dashboard tab
□ Printview
□ Ruler in Editor
□ Select CodeMirror Keymap
□ Snippets Menu
⊘ AddBefore
□ Cell Filter
□ Codefolding in Editor
☑ contrib_nbextensions_help_item
□ Execution Dependencies
□ Freeze
□ Hide input
□ Hinterland
□ Keyboard shortcut editor
□ Load TeX macros
☑ Nbextensions edit menu item
□ Python Markdown
□ Runtools
□ SKILL Syntax
□ spellchecker
□ Autopep8
□ Code Font Size
⊘ CodeMirror mode extensions
□ datestamper
□ Exercise
□ Gist-it
□ Hide input all
□ Initialization cells
⊘ Launch QTConsole
□ Move selected cells
□ nbTranslate
□ Rubberband
□ Scratchpad
⊘ Skip-Traceback
□ Split Cells Notebook

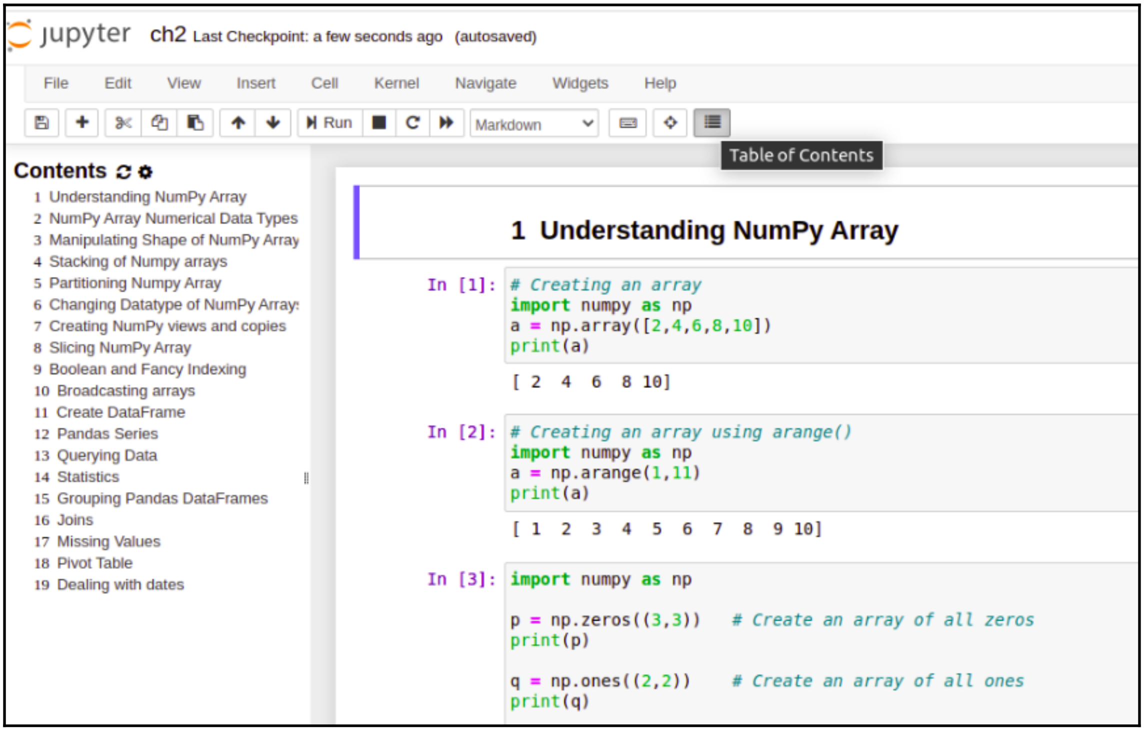
Now, let's explore a few useful features of Notebook extensions:

- **Hinterland**: This provides an autocompleting menu for each keypress that's made in cells and behaves like PyCharm:



```
In [8]: import pandas as pd
        import num

In [ ]:

In [ ]:
```

- **Table of Contents**: This extension shows all the headings in the sidebar or navigation menu. It is resizable, draggable, collapsible, and dockable:



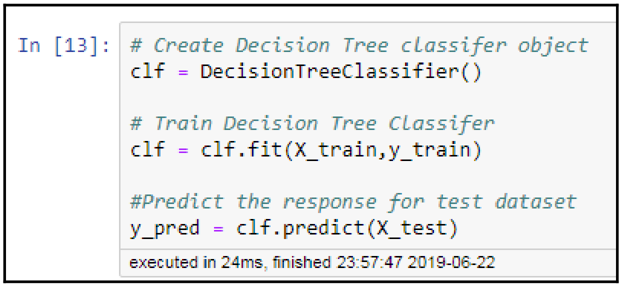
Contents

1 Understanding NumPy Array
2 NumPy Array Numerical Data Types
3 Manipulating Shape of NumPy Array
4 Stacking of Numpy arrays
5 Partitioning Numpy Array
6 Changing Datatype of NumPy Arrays
7 Creating NumPy views and copies
8 Slicing NumPy Array
9 Boolean and Fancy Indexing
10 Broadcasting arrays
11 Create DataFrame
12 Pandas Series
13 Querying Data
14 Statistics
15 Grouping Pandas DataFrames
16 Joins
17 Missing Values
18 Pivot Table
19 Dealing with dates

1 Understanding NumPy Array

```
In [1]: # Creating an array
        import numpy as np
        a = np.array([2,4,6,8,10])
        print(a)

        [ 2  4  6  8 10]

In [2]: # Creating an array using arange()
        import numpy as np
        a = np.arange(1,11)
        print(a)

        [ 1  2  3  4  5  6  7  8  9 10]

In [3]: import numpy as np

        p = np.zeros((3,3))    # Create an array of all zeros
        print(p)

        q = np.ones((2,2))     # Create an array of all ones
        print(q)
```

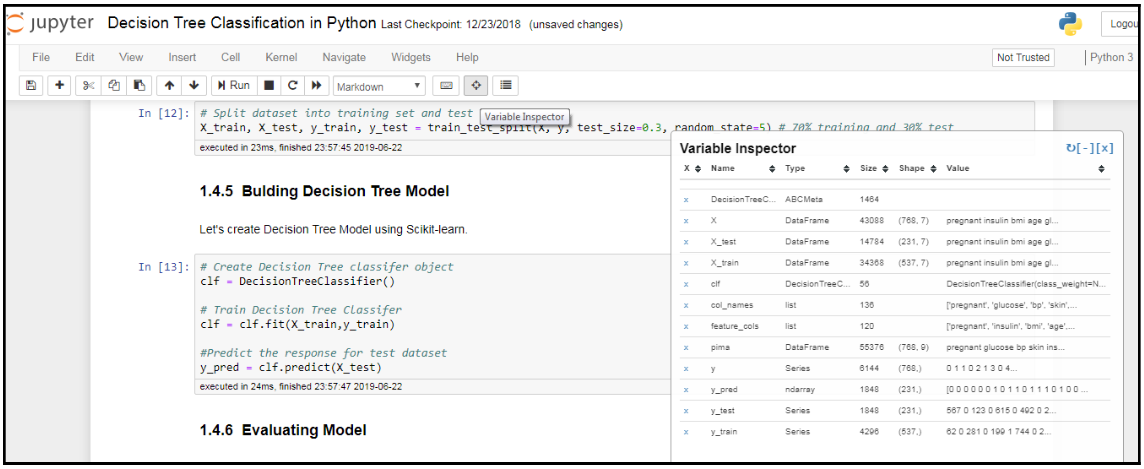
- **Execute Time**: This extension shows when the cells were executed and how much time it will take to complete the cell code:



```
In [13]: # Create Decision Tree classifer object
         clf = DecisionTreeClassifier()

         # Train Decision Tree Classifer
         clf = clf.fit(X_train,y_train)

         #Predict the response for test dataset
         y_pred = clf.predict(X_test)

         executed in 24ms, finished 23:57:47 2019-06-22
```

- **Spellchecker**: Spellchecker checks and verifies the spellings that are written in each cell and highlights any incorrectly written words.

- **Variable Selector**: This extension keeps track of the user's workspace. It shows the names of all the variables that the user created, along with their type, size, shape, and value.



```
In [12]: # Split dataset into training set and test
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=5) # 70% training and 30% test
```

executed in 23ms, finished 23:57:45 2019-06-22

1.4.5 Bulding Decision Tree Model

Let's create Decision Tree Model using Scikit-learn.

```
In [13]: # Create Decision Tree classifer object
         clf = DecisionTreeClassifier()

         # Train Decision Tree Classifer
         clf = clf.fit(X_train,y_train)

         #Predict the response for test dataset
         y_pred = clf.predict(X_test)
```

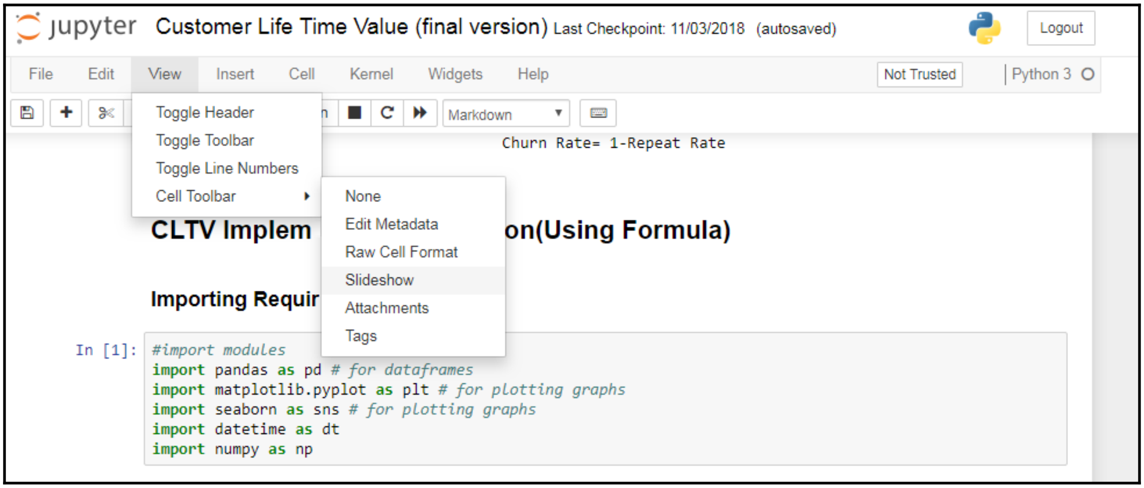
executed in 24ms, finished 23:57:47 2019-06-22

1.4.6 Evaluating Model

Variable Inspector ↻[-][x]

X	Name	Type	Size	Shape	Value
x	DecisionTreeC...	ABCMeta	1464		
x	X	DataFrame	43088	(768, 7)	pregnant insulin bmi age gl...
x	X_test	DataFrame	14784	(231, 7)	pregnant insulin bmi age gl...
x	X_train	DataFrame	34368	(537, 7)	pregnant insulin bmi age gl...
x	clf	DecisionTreeC...	56		DecisionTreeClassifier(class_weight=N...
x	col_names	list	136		['pregnant', 'glucose', 'bp', 'skin',...
x	feature_cols	list	120		['pregnant', 'insulin', 'bmi', 'age',...
x	pima	DataFrame	55376	(768, 9)	pregnant glucose bp skin ins...
x	y	Series	6144	(768,)	0 1 1 0 2 1 3 0 4...
x	y_pred	ndarray	1848	(231,)	[0 0 0 0 0 0 1 0 1 1 0 1 1 1 0 1 0 0 ...
x	y_test	Series	1848	(231,)	567 0 123 0 615 0 492 0 2...
x	y_train	Series	4296	(537,)	62 0 281 0 199 1 744 0 2...

- **Slideshow**: Notebook results can be communicated via Slideshow. This is a great tool for telling stories. Users can easily convert Jupyter Notebooks into slides without the use of PowerPoint. As shown in the following screenshot, Slideshow can be started using the **Slideshow** option in the cell toolbar of the view menu:



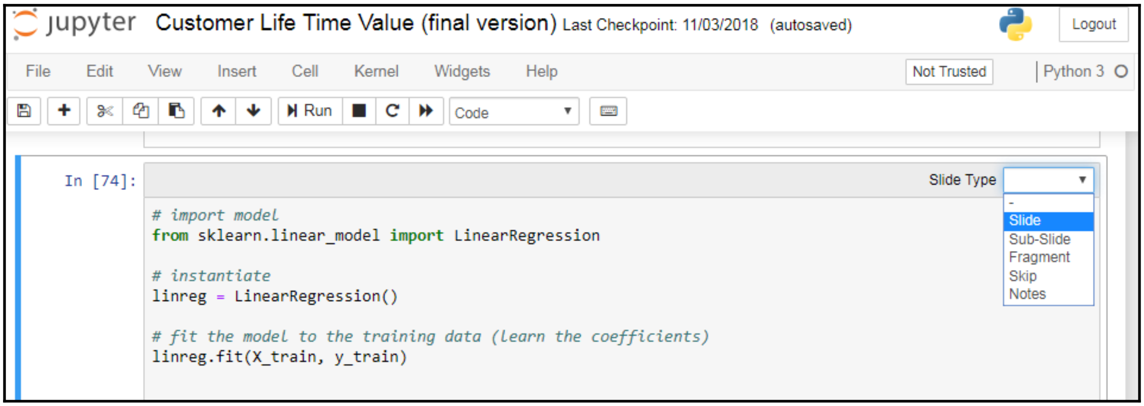
Churn Rate= 1-Repeat Rate

CLTV Implem on(Using Formula)

Importing Requir

```
In [1]: #import modules
        import pandas as pd # for dataframes
        import matplotlib.pyplot as plt # for plotting graphs
        import seaborn as sns # for plotting graphs
        import datetime as dt
        import numpy as np
```

Jupyter Notebook also allows you to show or hide any cell in Slideshow. After adding the **Slideshow** option to the cell toolbar of the view menu, you can use a **Slide Type** drop-down list in each cell and select various options, as shown in the following screenshot:

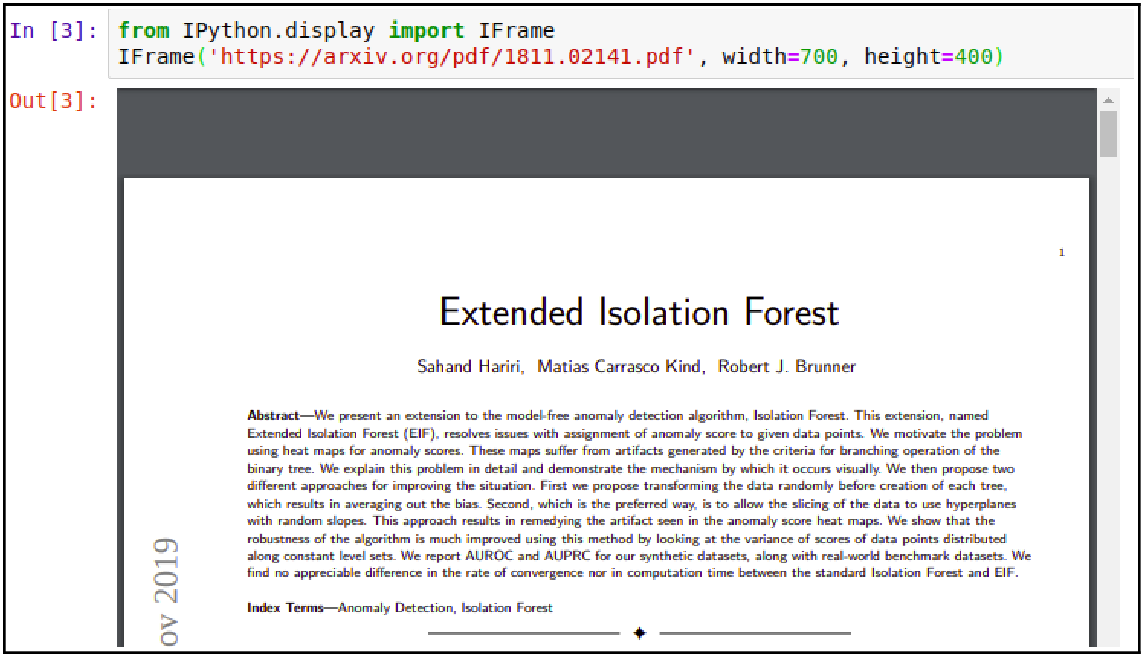


```
In [74]:
# import model
from sklearn.linear_model import LinearRegression

# instantiate
linreg = LinearRegression()

# fit the model to the training data (learn the coefficients)
linreg.fit(X_train, y_train)
```

- **Embedding PDF documents**: Jupyter Notebook users can easily add PDF documents. The following syntax needs to be run for PDf documents:

```
from IPython.display import IFrame
IFrame('https://arxiv.org/pdf/1811.02141.pdf', width=700,
height=400)
```

This results in the following output:



In [3]:
```
from IPython.display import IFrame
IFrame('https://arxiv.org/pdf/1811.02141.pdf', width=700, height=400)
```

Out[3]:

1

Extended Isolation Forest

Sahand Hariri, Matias Carrasco Kind, Robert J. Brunner

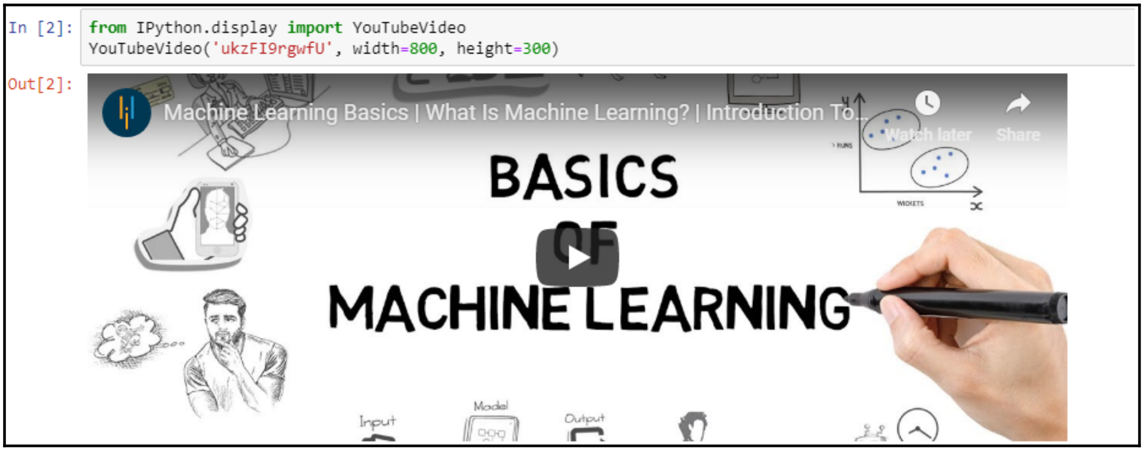
Abstract—We present an extension to the model-free anomaly detection algorithm, Isolation Forest. This extension, named Extended Isolation Forest (EIF), resolves issues with assignment of anomaly score to given data points. We motivate the problem using heat maps for anomaly scores. These maps suffer from artifacts generated by the criteria for branching operation of the binary tree. We explain this problem in detail and demonstrate the mechanism by which it occurs visually. We then propose two different approaches for improving the situation. First we propose transforming the data randomly before creation of each tree, which results in averaging out the bias. Second, which is the preferred way, is to allow the slicing of the data to use hyperplanes with random slopes. This approach results in remedying the artifact seen in the anomaly score heat maps. We show that the robustness of the algorithm is much improved using this method by looking at the variance of scores of data points distributed along constant level sets. We report AUROC and AUPRC for our synthetic datasets, along with real-world benchmark datasets. We find no appreciable difference in the rate of convergence nor in computation time between the standard Isolation Forest and EIF.

Index Terms—Anomaly Detection, Isolation Forest

- **Embedding Youtube Videos**: Jupyter Notebook users can easily add YouTube videos. The following syntax needs to be run for adding YouTube videos:

```
from IPython.display import YouTubeVideo
YouTubeVideo('ukzFI9rgwfU', width=700, height=400)
```

This results in the following output:



```
In [2]: from IPython.display import YouTubeVideo
        YouTubeVideo('ukzFI9rgwfU', width=800, height=300)
Out[2]:
```

Machine Learning Basics | What Is Machine Learning? | Introduction To...
Watch later
Share
BASICS
OF
MACHINE LEARNING
Input
Model
Output

With that, you now understand data analysis, the process that's undertaken by it, and the roles that it entails. You have also learned how to install Python and use Jupyter Lab and Jupyter Notebook. You will learn more about various Python libraries and data analysis techniques in the upcoming chapters.

Summary

In this chapter, we have discussed various data analysis processes, including KDD, SEMMA, and CRISP-DM. We then discussed the roles and skillsets of data analysts and data scientists. After that, we installed NumPy, SciPy, Pandas, Matplotlib, IPython, Jupyter Notebook, Anaconda, and Jupyter Lab, all of which we will be using in this book. Instead of installing all those modules, you can install Anaconda or Jupyter Lab, which has NumPy, Pandas, SciPy, and Scikit-learn built-in.

Then, we got a vector addition program working and learned how NumPy offers superior performance compared to the other libraries. We explored the available documentation and online resources. In addition, we discussed Jupyter Lab, Jupyter Notebook, and their features.

In the next chapter, Chapter 2, *NumPy and Pandas*, we will take a look at NumPy and Pandas under the hood and explore some of the fundamental concepts surrounding arrays and DataFrames.

2
NumPy and pandas

Now that we have understood data analysis, its process, and its installation on different platforms, it's time to learn about NumPy arrays and `pandas` DataFrames. This chapter acquaints you with the fundamentals of NumPy arrays and `pandas` DataFrames. By the end of this chapter, you will have a basic understanding of NumPy arrays, and `pandas` DataFrames and their related functions.

`pandas` is named after panel data (an econometric term) and Python data analysis and is a popular open-source Python library. We shall learn about basic `pandas` functionalities, data structures, and operations in this chapter. The official `pandas` documentation insists on naming the project `pandas` in all lowercase letters. The other convention the `pandas` project insists on is the `import pandas as pd` import statement.

In this chapter, our focus will be on the following topics:

- Understanding NumPy arrays
- NumPy array numerical data types
- Manipulating array shapes
- The stacking of NumPy arrays
- Partitioning NumPy arrays
- Changing the data type of NumPy arrays
- Creating NumPy views and copies
- Slicing NumPy arrays
- Boolean and fancy indexing
- Broadcasting arrays
- Creating `pandas` DataFrames
- Understanding `pandas` Series
- Reading and querying the Quandl data
- Describing `pandas` DataFrames

- Grouping and joining `pandas` DataFrames
- Working with missing values
- Creating pivot tables
- Dealing with dates

Technical requirements

This chapter has the following technical requirements:

- You can find the code and the dataset at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter02`.
- All the code blocks are available at `ch2.ipynb`.
- This chapter uses four CSV files (`WHO_first9cols.csv`, `dest.csv`, `purchase.csv`, and `tips.csv`) for practice purposes.
- In this chapter, we will use the NumPy, `pandas`, and Quandl Python libraries.

Understanding NumPy arrays

NumPy can be installed on a PC using `pip` or `brew` but if the user is using the Jupyter Notebook, then there is no need to install it. NumPy is already installed in the Jupyter Notebook. I will suggest to you to please use the Jupyter Notebook as your IDE because we are executing all the code in the Jupyter Notebook. We have already shown in `Chapter 1`, *Getting Started with Python Libraries*, how to install Anaconda, which is a complete suite for data analysis. NumPy arrays are a series of homogenous items. Homogenous means the array will have all the elements of the same data type. Let's create an array using NumPy. You can create an array using the `array()` function with a list of items. Users can also fix the data type of an array. Possible data types are `bool`, `int`, `float`, `long`, `double`, and `long double`.

Let's see how to create an empty array:

```
# Creating an array
import numpy as np
a = np.array([2,4,6,8,10])
print(a)

Output:
[ 2  4  6  8 10]
```

Another way to create a NumPy array is with `arange()`. It creates an evenly spaced NumPy array. Three values – start, stop, and step – can be passed to the `arange(start,[stop],step)` function. The start is the initial value of the range, the stop is the last value of the range, and the step is the increment in that range. The stop parameter is compulsory. In the following example, we have used `1` as the start and `11` as the stop parameter. The `arange(1,11)` function will return 1 to 10 values with one step because the step is, by default, 1. The `arrange()` function generates a value that is one less than the stop parameter value. Let's understand this through the following example:

```
# Creating an array using arange()
import numpy as np
a = np.arange(1,11)
print(a)

Output:
[ 1  2  3  4  5  6  7  8  9 10]
```

Apart from the `array()` and `arange()` functions, there are other options, such as `zeros()`, `ones()`, `full()`, `eye()`, and `random()`, which can also be used to create a NumPy array, as these functions are initial placeholders. Here is a detailed description of each function:

- `zeros()`: The `zeros()`function creates an array for a given dimension with all zeroes.
- `ones()`: The `ones()` function creates an array for a given dimension with all ones.
- `fulls()`: The `full()` function generates an array with constant values.
- `eyes()`: The `eye()` function creates an identity matrix.
- `random()`: The `random()` function creates an array with any given dimension.

Let's understand these functions through the following example:

```
import numpy as np

# Create an array of all zeros
p = np.zeros((3,3))
print(p)

# Create an array of all ones
q = np.ones((2,2))
print(q)

# Create a constant array
r = np.full((2,2), 4)
```

```
print(r)

# Create a 2x2 identity matrix
s = np.eye(4)
print(s)

# Create an array filled with random values
t = np.random.random((3,3))
print(t)
```

This results in the following output:

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

[[1. 1.]
 [1. 1.]]

[[4 4]
 [4 4]]

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

[[0.16681892 0.00398631 0.61954178]
 [0.52461924 0.30234715 0.58848138]
 [0.75172385 0.17752708 0.12665832]]
```

In the preceding code, we have seen some built-in functions for creating arrays with all-zero values, all-one values, and all-constant values. After that, we have created the identity matrix using the `eye()` function and a random matrix using the `random.random()` function. Let's see some other array features in the next section.

Array features

In general, NumPy arrays are a homogeneous kind of data structure that has the same types of items. The main benefit of an array is its certainty of storage size because of its same type of items. A Python list uses a loop to iterate the elements and perform operations on them. Another benefit of NumPy arrays is to offer vectorized operations instead of iterating each item and performing operations on it. NumPy arrays are indexed just like a Python list and start from 0. NumPy uses an optimized C API for the fast processing of the array operations.

Let's make an array using the `arange()` function, as we did in the previous section, and let's check its data type:

```
# Creating an array using arange()
import numpy as np
a = np.arange(1,11)

print(type(a))
print(a.dtype)

Output:
<class 'numpy.ndarray'>
int64
```

When you use `type()`, it returns `numpy.ndarray`. This means that the `type()` function returns the type of the container. When you use `dtype()`, it will return `int64`, since it is the type of the elements. You may also get the output as `int32` if you are using 32-bit Python. Both cases use integers (32- and 64-bit). One-dimensional NumPy arrays are also known as vectors.

Let's find out the shape of the vector that we produced a few minutes ago:

```
print(a.shape)
Output: (10,)
```

As you can see, the vector has 10 elements with values ranging from 1 to 10. The `shape` property of the array is a tuple; in this instance, it is a tuple of one element, which holds the length in each dimension.

Selecting array elements

In this section, we will see how to select the elements of the array. Let's see an example of a 2*2 matrix:

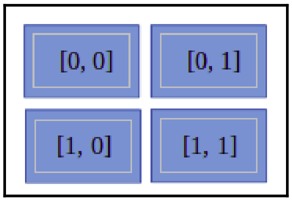
```
a = np.array([[5,6],[7,8]])
print(a)

Output:
[[5 6]
 [7 8]]
```

In the preceding example, the matrix is created using the `array()` function with the input list of lists.

Selecting array elements is pretty simple. We just need to specify the index of the matrix as `a[m,n]`. Here, `m` is the row index and `n` is the column index of the matrix. We will now select each item of the matrix one by one as shown in the following code:

```
print(a[0,0])
Output: 5

print(a[0,1])
Output: 6

printa([1,0])
Output: 7

printa([1,1])
Output: 8
```

In the preceding code sample, we have tried to access each element of an array using array indices. You can also understand this by the diagram mentioned here:



[0, 0] [0, 1]

[1, 0] [1, 1]

In the preceding diagram, we can see it has four blocks and each block represents the element of an array. The values written in each block show its indices.

In this section, we have understood the fundamentals of arrays. Now, let's jump to arrays of numerical data types.

NumPy array numerical data types

Python offers three types of numerical data types: integer type, float type, and complex type. In practice, we need more data types for scientific computing operations with precision, range, and size. NumPy offers a bulk of data types with mathematical types and numbers. Let's see the following table of NumPy numerical types:

Data Type	**Details**
`bool`	This is a Boolean type that stores a bit and takes `True` or `False` values.
`inti`	Platform integers can be either `int32` or `int64`.
`int8`	Byte store values range from `-128` to `127`.
`int16`	This stores integers ranging from `-32768` to `32767`.
`int32`	This stores integers ranging from `-2 ** 31` to `2 ** 31 -1`.
`int64`	This stores integers ranging from `-2 ** 63` to `2 ** 63 -1`.
`uint8`	This stores unsigned integers ranging from `0` to `255`.
`uint16`	This stores unsigned integers ranging from `0` to `65535`.
`uint32`	This stores unsigned integers ranging from `0` to `2 ** 32 – 1`.
`uint64`	This stores unsigned integers ranging from `0` to `2 ** 64 – 1`.
`float16`	Half-precision float; sign bit with 5 bits exponent and 10 bits mantissa.
`float32`	Single-precision float; sign bit with 8 bits exponent and 23 bits mantissa.
`float64` or `float`	Double-precision float; sign bit with 11 bits exponent and 52 bits mantissa.
`complex64`	Complex number stores two 32-bit floats: real and imaginary number.
`complex128` or `complex`	Complex number stores two 64-bit floats: real and imaginary number.

For each data type, there exists a matching conversion function:

```
print(np.float64(21))
Output: 21.0

print(np.int8(21.0))
Output: 42

print(np.bool(21))
Output: True

print(np.bool(0))
Output: False

print(np.bool(21.0))
Output: True

print(np.float(True))
Output: 1.0

print(np.float(False))
Output: 0.0
```

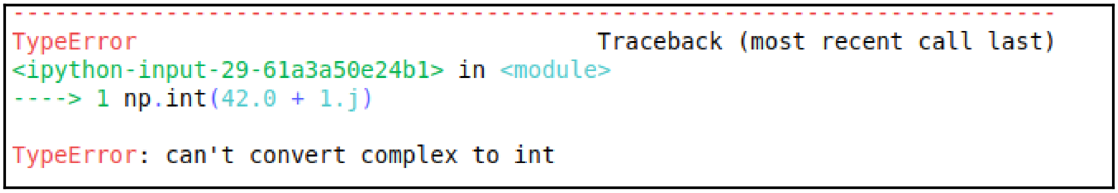
Many functions have a data type argument, which is frequently optional:

```
arr=np.arange(1,11, dtype= np.float32)
print(arr)

Output:
[ 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.]
```

It is important to be aware that you are not allowed to change a complex number into an integer. If you try to convert complex data types into integers, then you will get `TypeError`. Let's see the following example:

```
np.int(42.0 + 1.j)
```

This results in the following output:



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-61a3a50e24b1> in <module>
----> 1 np.int(42.0 + 1.j)

TypeError: can't convert complex to int
```

You will get the same error if you try the conversion of a complex number into a floating point.

But you can convert float values into complex numbers by setting individual pieces. You can also pull out the pieces using the `real` and `imag` attributes. Let's see that using the following example:

```
c= complex(42, 1)
print(c)

Output: (42+1j)

print(c.real,c.imag)

Output: 42.0 1.0
```

In the preceding example, you have defined a complex number using the `complex()` method. Also, you have extracted the real and imaginary values using the `real` and `imag` attributes. Let's now jump to `dtype` objects.

dtype objects

We have seen in earlier sections of the chapter that `dtype` tells us the type of individual elements of an array. NumPy array elements have the same data type, which means that all elements have the same `dtype`. `dtype` objects are instances of the `numpy.dtype` class:

```
# Creating an array
import numpy as np
a = np.array([2,4,6,8,10])

print(a.dtype)
Output: 'int64'
```

`dtype` objects also tell us the size of the data type in bytes using the `itemsize` property:

```
print(a.dtype.itemsize)
Output:8
```

Data type character codes

Character codes are included for backward compatibility with Numeric. Numeric is the predecessor of NumPy. Its use is not recommended, but the code is supplied here because it pops up in various locations. You should use the `dtype` object instead. The following table lists several different data types and the character codes related to them:

Type	**Character Code**
Integer	i
Unsigned integer	u
Single-precision float	f
Double-precision float	d
Bool	b
Complex	D
String	S
Unicode	U
Void	V

Let's take a look at the following code to produce an array of single-precision floats:

```
# Create numpy array using arange() function
var1=np.arange(1,11, dtype='f')
print(var1)

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

Likewise, the following code creates an array of complex numbers:

```
print(np.arange(1,6, dtype='D'))

Output:
[1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j]
```

dtype constructors

There are lots of ways to create data types using constructors. Constructors are used to instantiate or assign a value to an object. In this section, we will understand data type creation with the help of a floating-point data example:

- To try out a general Python float, use the following:

  ```
  print(np.dtype(float))
  Output: float64
  ```

- To try out a single-precision float with a character code, use the following:

  ```
  print(np.dtype('f'))
  Output: float32
  ```

- To try out a double-precision float with a character code, use the following:

  ```
  print(np.dtype('d'))
  Output: float64
  ```

- To try out a `dtype` constructor with a two-character code, use the following:

  ```
  print(np.dtype('f8'))
  Output: float64
  ```

Here, the first character stands for the type and a second character is a number specifying the number of bytes in the type, for example, 2, 4, or 8.

dtype attributes

The `dtype` class offers several useful attributes. For example, we can get information about the character code of a data type using the `dtype` attribute:

```
# Create numpy array
var2=np.array([1,2,3],dtype='float64')

print(var2.dtype.char)

Output: 'd'
```

The `type` attribute corresponds to the type of object of the array elements:

```
print(var2.dtype.type)

Output: <class 'numpy.float64'>
```

Now that we know all about the various data types used in NumPy arrays, let's start manipulating them in the next section.

Manipulating array shapes

In this section, our main focus is on array manipulation. Let's learn some new Python functions of NumPy, such as `reshape()`, `flatten()`, `ravel()`, `transpose()`, and `resize()`:

- `reshape()` will change the shape of the array:

  ```
  # Create an array
  arr = np.arange(12)
  print(arr)

  Output: [ 0  1  2  3  4  5  6  7  8  9 10 11]

  # Reshape the array dimension
  new_arr=arr.reshape(4,3)

  print(new_arr)

  Output: [[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]]

  # Reshape the array dimension
  new_arr2=arr.reshape(3,4)

  print(new_arr2)

  Output:
  array([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
  ```

- Another operation that can be applied to arrays is `flatten()`. `flatten()` transforms an n-dimensional array into a one-dimensional array:

  ```
  # Create an array
  arr=np.arange(1,10).reshape(3,3)
  print(arr)

  Output:
  ```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

print(arr.flatten())

Output:
[1 2 3 4 5 6 7 8 9]
```

- The `ravel()` function is similar to the `flatten()` function. It also transforms an n-dimensional array into a one-dimensional array. The main difference is that `flatten()` returns the actual array while `ravel()` returns the reference of the original array. The `ravel()` function is faster than the `flatten()` function because it does not occupy extra memory:

```
print(arr.ravel())

Output:
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The `transpose()` function is a linear algebraic function that transposes the given two-dimensional matrix. The word transpose means converting rows into columns and columns into rows:

```
# Transpose the matrix
print(arr.transpose())

Output:
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

- The `resize()` function changes the size of the NumPy array. It is similar to `reshape()` but it changes the shape of the original array:

```
# resize the matrix
arr.resize(1,9)
print(arr)

Output:[[1 2 3 4 5 6 7 8 9]]
```

In all the code in this section, we have seen built-in functions such as `reshape()`, `flatten()`, `ravel()`, `transpose()`, and `resize()` for manipulating size. Now, it's time to learn about the stacking of NumPy arrays.

The stacking of NumPy arrays

NumPy offers a stack of arrays. Stacking means joining the same dimensional arrays along with a new axis. Stacking can be done horizontally, vertically, column-wise, row-wise, or depth-wise:

- **Horizontal stacking**: In horizontal stacking, the same dimensional arrays are joined along with a horizontal axis using the `hstack()` and `concatenate()` functions. Let's see the following example:

```
arr1 = np.arange(1,10).reshape(3,3)
print(arr1)

Output:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

We have created one 3*3 array; it's time to create another 3*3 array:

```
arr2 = 2*arr1
print(arr2)

Output:
[[ 2 4 6]
 [ 8 10 12]
 [14 16 18]]
```

After creating two arrays, we will perform horizontal stacking:

```
# Horizontal Stacking
arr3=np.hstack((arr1, arr2))
print(arr3)

Output:
[[ 1 2 3 2 4 6]
 [ 4 5 6 8 10 12]
 [ 7 8 9 14 16 18]]
```

In the preceding code, two arrays are stacked horizontally along the *x* axis. The `concatenate()` function can also be used to generate the horizontal stacking with axis parameter value `1`:

```
# Horizontal stacking using concatenate() function
arr4=np.concatenate((arr1, arr2), axis=1)
print(arr4)
```

```
Output:
[[ 1  2  3  2  4  6]
 [ 4  5  6  8 10 12]
 [ 7  8  9 14 16 18]]
```

In the preceding code, two arrays have been stacked horizontally using the `concatenate()` function.

- **Vertical stacking**: In vertical stacking, the same dimensional arrays are joined along with a vertical axis using the `vstack()` and `concatenate()` functions. Let's see the following example:

```
# Vertical stacking
arr5=np.vstack((arr1, arr2))
print(arr5)

Output:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

In the preceding code, two arrays are stacked vertically along the *y* axis. The `concatenate()` function can also be used to generate vertical stacking with axis parameter value `0`:

```
arr6=np.concatenate((arr1, arr2), axis=0)
print(arr6)

Output:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

In the preceding code, two arrays are stacked vertically using the `concatenate()` function.

- **Depth stacking**: In depth stacking, the same dimensional arrays are joined along with a third axis (depth) using the `dstack()` function. Let's see the following example:

```
arr7=np.dstack((arr1, arr2))
print(arr7)

Output:
[[[ 1  2]
  [ 2  4]
  [ 3  6]]

 [[ 4  8]
  [ 5 10]
  [ 6 12]]

 [[ 7 14]
  [ 8 16]
  [ 9 18]]]
```

In the preceding code, two arrays are stacked in depth along with a third axis (depth).

- **Column stacking**: Column stacking stacks multiple sequence one-dimensional arrays as columns into a single two-dimensional array. Let's see an example of column stacking:

```
# Create 1-D array
arr1 = np.arange(4,7)
print(arr1)

Output: [4, 5, 6]
```

In the preceding code block, we have created a one-dimensional NumPy array.

```
# Create 1-D array
arr2 = 2 * arr1
print(arr2)

Output: [ 8, 10, 12]
```

In the preceding code block, we have created another one-dimensional NumPy array.

```
# Create column stack
arr_col_stack=np.column_stack((arr1,arr2))
print(arr_col_stack)
```

```
Output:
[[ 4  8]
 [ 5 10]
 [ 6 12]]
```

In the preceding code, we have created two one-dimensional arrays and stacked them column-wise.

- **Row stacking**: Row stacking stacks multiple sequence one-dimensional arrays as rows into a single two-dimensional arrays. Let's see an example of row stacking:

```
# Create row stack
arr_row_stack = np.row_stack((arr1,arr2))
print(arr_row_stack)

Output:
[[ 4  5  6]
 [ 8 10 12]]
```

In the preceding code, two one-dimensional arrays are stacked row-wise.

Let's now see how to partition a NumPy array into multiple sub-arrays.

Partitioning NumPy arrays

NumPy arrays can be partitioned into multiple sub-arrays. NumPy offers three types of split functionality: vertical, horizontal, and depth-wise. All the split functions by default split into the same size arrays but we can also specify the split location. Let's look at each of the functions in detail:

- **Horizontal splitting**: In horizontal split, the given array is divided into *N* equal sub-arrays along the horizontal axis using the `hsplit()` function. Let's see how to split an array:

```
# Create an array
arr=np.arange(1,10).reshape(3,3)
print(arr)

Output:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

# Peroform horizontal splitting
arr_hor_split=np.hsplit(arr, 3)
```

```
print(arr_hor_split)

Output:
[array([[1],
       [4],
       [7]]), array([[2],
       [5],
       [8]]), array([[3],
       [6],
       [9]])]
```

In the preceding code, the `hsplit(arr, 3)` function divides the array into three sub-arrays. Each part is a column of the original array.

- **Vertical splitting**: In vertical split, the given array is divided into *N* equal sub-arrays along the vertical axis using the `vsplit()` and `split()` functions. The `split` function with `axis=0` performs the same operation as the `vsplit()` function:

```
# vertical split
arr_ver_split=np.vsplit(arr, 3)

print(arr_ver_split)

Output:
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
```

In the preceding code, the `vsplit(arr, 3)` function divides the array into three sub-arrays. Each part is a row of the original array. Let's see another function, `split()`, which can be utilized as a vertical and horizontal split, in the following example:

```
# split with axis=0
arr_split=np.split(arr,3,axis=0)

print(arr_split)

Output:
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]

# split with axis=1
arr_split = np.split(arr,3,axis=1)

Output:
[array([[1],
        [4],
        [7]]), array([[2],
        [5],
        [8]]), array([[3],
        [6],
        [9]])]
```

In the preceding code, the `split(arr, 3)` function divides the array into three sub-arrays. Each part is a row of the original array. The split output is similar to the `vsplit()` function when `axis=0` and the split output is similar to the `hsplit()` function when `axis=1`.

Changing the data type of NumPy arrays

As we have seen in the preceding sections, NumPy supports multiple data types, such as `int`, `float`, and complex numbers. The `astype()` function converts the data type of the array. Let's see an example of the `astype()` function:

```
# Create an array
arr=np.arange(1,10).reshape(3,3)
print("Integer Array:",arr)

# Change datatype of array
arr=arr.astype(float)

# print array
print("Float Array:", arr)

# Check new data type of array
print("Changed Datatype:", arr.dtype)
```

In the preceding code, we have created one NumPy array and checked its data type using the `dtype` attribute.

Let's change the data type of an array using the `astype()` function:

```
# Change datatype of array
arr=arr.astype(float)

# Check new data type of array
print(arr.dtype)

Output:
float64
```

In the preceding code, we have changed the column data type from integer to float using `astype()`.

The `tolist()` function converts a NumPy array into a Python list. Let's see an example of the `tolist()` function:

```
# Create an array
arr=np.arange(1,10)

# Convert NumPy array to Python List
list1=arr.tolist()
print(list1)

Output:
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In the preceding code, we have converted an array into a Python list object using the `tolist()` function.

Creating NumPy views and copies

Some of the Python functions return either a copy or a view of the input array. A Python copy stores the array in another location while a view uses the same memory content. This means copies are separate objects and treated as a deep copy in Python. Views are the original base array and are treated as a shallow copy. Here are some properties of copies and views:

- Modifications in a view affect the original data whereas modifications in a copy do not affect the original array.
- Views use the concept of shared memory.
- Copies require extra space compared to views.
- Copies are slower than views.

Let's understand the concept of copy and view using the following example:

```
# Create NumPy Array
arr = np.arange(1,5).reshape(2,2)
print(arr)

Output:
[[1, 2],
[3, 4]]
```

After creating a NumPy array, let's perform object copy operations:

```
# Create no copy only assignment
arr_no_copy=arr

# Create Deep Copy
arr_copy=arr.copy()

# Create shallow copy using View
arr_view=arr.view()

print("Original Array: ",id(arr))
print("Assignment: ",id(arr_no_copy))
print("Deep Copy: ",id(arr_copy))
print("Shallow Copy(View): ",id(arr_view))

Output:
Original Array:  140426327484256
Assignment:  140426327484256
Deep Copy:  140426327483856
Shallow Copy(View):  140426327484496
```

In the preceding example, you can see the original array and the assigned array have the same object ID, meaning both are pointing to the same object. Copies and views both have different object IDs; both will have different objects, but view objects will reference the same original array and a copy will have a different replica of the object.

Let's continue with this example and update the values of the original array and check its impact on views and copies:

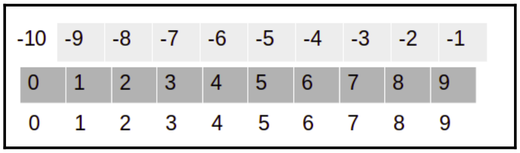
```
# Update the values of original array
arr[1]=[99,89]

# Check values of array view
print("View Array:\n", arr_view)

# Check values of array copy
print("Copied Array:\n", arr_copy)

Output:
View Array:
 [[ 1  2]
 [99 89]]
Copied Array:
 [[1 2]
 [3 4]]
```

In the preceding example, we can conclude from the results that the view is the original array. The values changed when we updated the original array and the copy is a separate object because its values remain the same.

Slicing NumPy arrays

Slicing in NumPy is similar to Python lists. Indexing prefers to select a single value while slicing is used to select multiple values from an array.

NumPy arrays also support negative indexing and slicing. Here, the negative sign indicates the opposite direction and indexing starts from the right-hand side with a starting value of `-1`:



-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

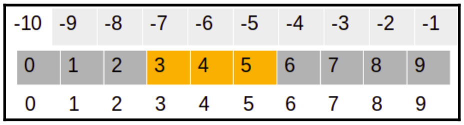
Let's check this out using the following code:

```
# Create NumPy Array
arr = np.arange(0,10)
print(arr)

Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In the slice operation, we use the colon symbol to select the collection of values. Slicing takes three values: start, stop, and step:

```
print(arr[3:6])
Output: [3, 4, 5]
```

This can be represented as follows:



-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

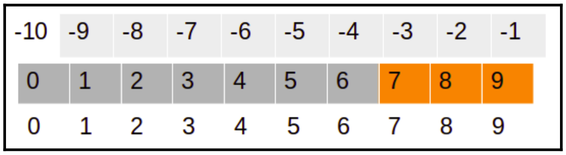
In the preceding example, we have used `3` as the starting index and `6` as the stopping index:

```
print(arr[3:])
Output: array([3, 4, 5, 6, 7, 8, 9])
```

In the preceding example, only the starting index is given. `3` is the starting index. This slice operation will select the values from the starting index to the end of the array:

```
print(arr[-3:])
Output: array([7, 8, 9])
```

This can be represented as follows:

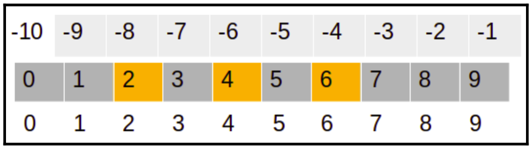


-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

In the preceding example, the slice operation will select values from the third value from the right side of the array to the end of the array:

```
print(arr[2:7:2])
Output: array([2, 4,6])
```

This can be represented as follows:



-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

In the preceding example, the start, stop, and step index are 2, 7, and 2, respectively. Here, the slice operation selects values from the second index to the sixth (one less than the stop value) index with an increment of 2 in the index value. So, the output will be 2, 4, and 6.

Boolean and fancy indexing

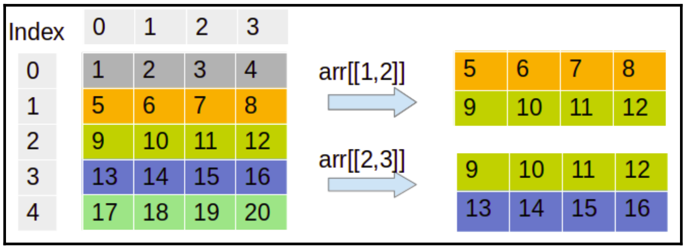
Indexing techniques help us to select and filter elements from a NumPy array. In this section, we will focus on Boolean and fancy indexing. Boolean indexing uses a Boolean expression in the place of indexes (in square brackets) to filter the NumPy array. This indexing returns elements that have a true value for the Boolean expression:

```
# Create NumPy Array
arr = np.arange(21,41,2)
print("Orignial Array:\n",arr)

# Boolean Indexing
print("After Boolean Condition:",arr[arr>30])

Output:
Orignial Array:
 [21 23 25 27 29 31 33 35 37 39]
After Boolean Condition: [31 33 35 37 39]
```

Fancy indexing is a special type of indexing in which elements of an array are selected by an array of indices. This means we pass the array of indices in brackets. Fancy indexing also supports multi-dimensional arrays. This will help us to easily select and modify a complex multi-dimensional set of arrays. Let's see an example as follows to understand fancy indexing:

```
# Create NumPy Array
arr = np.arange(1,21).reshape(5,4)
print("Orignial Array:\n",arr)

# Selecting 2nd and 3rd row
indices = [1,2]
print("Selected 1st and 2nd Row:\n", arr[indices])

# Selecting 3nd and 4th row
indices = [2,3]
print("Selected 3rd and 4th Row:\n", arr[indices])

Output:

Orignial Array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]
Selected 1st and 2nd Row:
 [[ 5  6  7  8]
 [ 9 10 11 12]]
Selected 3rd and 4th Row:
 [[ 9 10 11 12]
 [13 14 15 16]]
```

In the preceding code, we have created a 5*4 matrix and selected the rows using integer indices. You can also visualize or internalize this output from the following diagram:



We can see the code for this as follows:

```
# Create row and column indices
row = np.array([1, 2])
col = np.array([2, 3])

print("Selected Sub-Array:", arr[row, col])

Output:
Selected Sub-Array: [ 7 12]
```

The preceding example results in the first value, `[1,2]`, and second value, `[2,3]`, as the row and column index. The array will select the value at the first and second index values, which are 7 and 12.

Broadcasting arrays

Python lists do not support direct vectorizing arithmetic operations. NumPy offers a faster-vectorized array operation compared to Python list loop-based operations. Here, all the looping operations are performed in C instead of Python, which makes it faster. Broadcasting functionality checks a set of rules for applying binary functions, such as addition, subtraction, and multiplication, on different shapes of an array.

Let's see an example of broadcasting:

```
# Create NumPy Array
arr1 = np.arange(1,5).reshape(2,2)
print(arr1)

Output:
[[1 2]
 [3 4]]

# Create another NumPy Array
arr2 = np.arange(5,9).reshape(2,2)
print(arr2)

Output:
[[5 6]
 [7 8]]

# Add two matrices
print(arr1+arr2)

Output:
[[ 6 8]
 [10 12]]
```

In all three preceding examples, we can see the addition of two arrays of the same size. This concept is known as broadcasting:

```
# Multiply two matrices
print(arr1*arr2)

Output:
[[ 5 12]
 [21 32]]
```

In the preceding example, two matrices were multiplied. Let's perform addition and multiplication with a scalar value:

```
# Add a scaler value
print(arr1 + 3)

Output:
[[4 5]
 [6 7]]

# Multiply with a scalar value
print(arr1 * 3)
```

```
Output:
[[ 3  6]
 [ 9 12]]
```

In the preceding two examples, the matrix is added and multiplied by a scalar value.

Creating pandas DataFrames

The `pandas` library is designed to work with a panel or tabular data. `pandas` is a fast, highly efficient, and productive tool for manipulating and analyzing string, numeric, datetime, and time-series data. `pandas` provides data structures such as DataFrames and Series. A `pandas` DataFrame is a tabular, two-dimensional labeled and indexed data structure with a grid of rows and columns. Its columns are heterogeneous types. It has the capability to work with different types of objects, carry out grouping and joining operations, handle missing values, create pivot tables, and deal with dates. A `pandas` DataFrame can be created in multiple ways. Let's create an empty DataFrame:

```
# Import pandas library
import pandas as pd

# Create empty DataFrame
df = pd.DataFrame()

# Header of dataframe.
df.head()

Output:
_
```

In the preceding example, we have created an empty DataFrame. Let's create a DataFrame using a dictionary of the list:

```
# Create dictionary of list
data = {'Name': ['Vijay', 'Sundar', 'Satyam', 'Indira'], 'Age': [23, 45,
46, 52 ]}

# Create the pandas DataFrame
df = pd.DataFrame(data)

# Header of dataframe.
df.head()

Output:
  Name Age
0 Vijay 23
```

```
1 Sundar 45
2 Satyam 46
3 Indira 52
```

In the preceding code, we have used a dictionary of the list to create a DataFrame. Here, the keys of the dictionary are equivalent to columns, and values are represented as a list that is equivalent to the rows of the DataFrame. Let's create a DataFrame using the list of dictionaries:

```
# Pandas DataFrame by lists of dicts.
# Initialise data to lists.
data =[ {'Name': 'Vijay', 'Age': 23},{'Name': 'Sundar', 'Age': 25},{'Name':
'Shankar', 'Age': 26}]

# Creates DataFrame.
df = pd.DataFrame(data,columns=['Name','Age'])

# Print dataframe header
df.head()
```

In the preceding code, the DataFrame is created using a list of dictionaries. In the list, each item is a dictionary. Each key is the name of the column and the value is the cell value for a row. Let's create a DataFrame using a list of tuples:

```
# Creating DataFrame using list of tuples.
data = [('Vijay', 23),( 'Sundar', 45), ('Satyam', 46), ('Indira',52)]

# Create dataframe
df = pd.DataFrame(data, columns=['Name','Age'])

# Print dataframe header
df.head()

Output:
  Name Age
0 Vijay 23
1 Sundar 25
2 Shankar 26
```

In the preceding code, the DataFrame is created using a list of tuples. In the list, each item is a tuple and each tuple is equivalent to the row of columns.

Understanding pandas Series

`pandas` Series is a one-dimensional sequential data structure that is able to handle any type of data, such as string, numeric, datetime, Python lists, and dictionaries with labels and indexes. Series is one of the columns of a DataFrame. We can create a Series using a Python dictionary, NumPy array, and scalar value. We will also see the `pandas` Series features and properties in the latter part of the section. Let's create some Python Series:

- **Using a Python dictionary**: Create a dictionary object and pass it to the Series object. Let's see the following example:

```
# Creating Pandas Series using Dictionary
dict1 = {0 : 'Ajay', 1 : 'Jay', 2 : 'Vijay'}

# Create Pandas Series
series = pd.Series(dict1)

# Show series
series

Output:
0     Ajay
1      Jay
2    Vijay
dtype: object
```

- **Using a NumPy array**: Create a NumPy array object and pass it to the Series object. Let's see the following example:

```
#Load Pandas and NumPy libraries
import pandas as pd
import numpy as np

# Create NumPy array
arr = np.array([51,65,48,59, 68])

# Create Pandas Series
series = pd.Series(arr)
series

Output:
```

```
0    51
1    65
2    48
3    59
4    68
dtype: int64
```

- **Using a single scalar value**: To create a pandas Series with a scalar value, pass the scalar value and index list to a Series object:

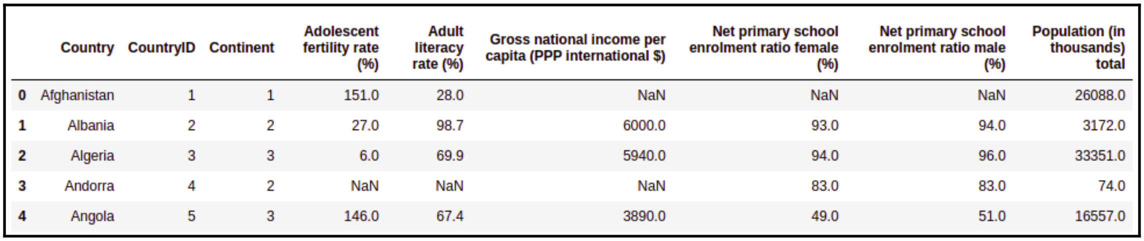
```
# load Pandas and NumPy
import pandas as pd
import numpy as np

# Create Pandas Series
series = pd.Series(10, index=[0, 1, 2, 3, 4, 5])
series

Output:
0    10
1    10
2    10
3    10
4    10
5    10
dtype: int64
```

Let's explore some features of pandas Series:

- We can also create a series by selecting a column, such as country, which happens to be the first column in the datafile. Then, show the type of the object currently in the local scope:

```
# Import pandas
import pandas as pd

# Load data using read_csv()
df = pd.read_csv("WHO_first9cols.csv")

# Show initial 5 records
df.head()
```

This results in the following output:



	Country	CountryID	Continent	Adolescent fertility rate (%)	Adult literacy rate (%)	Gross national income per capita (PPP international $)	Net primary school enrolment ratio female (%)	Net primary school enrolment ratio male (%)	Population (in thousands) total
0	Afghanistan	1	1	151.0	28.0	NaN	NaN	NaN	26088.0
1	Albania	2	2	27.0	98.7	6000.0	93.0	94.0	3172.0
2	Algeria	3	3	6.0	69.9	5940.0	94.0	96.0	33351.0
3	Andorra	4	2	NaN	NaN	NaN	83.0	83.0	74.0
4	Angola	5	3	146.0	67.4	3890.0	49.0	51.0	16557.0

In the preceding code, we have read the `WHO_first9cols.csv` file using the `read_csv()` function. You can download this file from the following GitHub location:`https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter02`. In the output, you can see the top five records in the `WHO_first9cols` dataset using the `head()` function:

```
# Select a series
country_series=df['Country']

# check datatype of series
type(country_series)

Output:
pandas.core.series.Series
```

- The `pandas` Series data structure shares some of the common attributes of DataFrames and also has a `name` attribute. Explore these properties as follows:

```
# Show the shape of DataFrame
print("Shape:", df.shape)

Output:
Shape: (202, 9)
```

Let's check the column list of a DataFrame:

```
# Check the column list of DataFrame
print("List of Columns:", df.columns)

Output:List of Columns: Index(['Country', 'CountryID', 'Continent',
'Adolescent fertility rate (%)',
       'Adult literacy rate (%)',
       'Gross national income per capita (PPP international $)',
       'Net primary school enrolment ratio female (%)',
       'Net primary school enrolment ratio male (%)',
       'Population (in thousands) total'],
       dtype='object')
```

Let's check the data types of DataFrame columns:

```
# Show the datatypes of columns
print("Data types:", df.dtypes)

Output:
Data types: Country
object
            CountryID
int64
            Continent
int64
            Adolescent fertility rate (%)
float64
            Adult literacy rate (%)
float64
            Gross national income per capita (PPP international $)
float64
            Net primary school enrolment ratio female (%)
float64
            Net primary school enrolment ratio male (%)
float64
            Population (in thousands) total
float64
            dtype: object
```

3. Let's see the slicing of a `pandas` Series:

```
# Pandas Series Slicing
country_series[-5:]

Output:
197              Vietnam
198    West Bank and Gaza
199                Yemen
200               Zambia
201             Zimbabwe
Name: Country, dtype: object
```

Now that we know how to use pandas Series, let's move on to using Quandl to work on databases.

Reading and querying the Quandl data

In the last section, we saw `pandas` DataFrames that have a tabular structure similar to relational databases. They offer similar query operations on DataFrames. In this section, we will focus on Quandl. Quandl is a Canada-based company that offers commercial and alternative financial data for investment data analyst. Quandl understands the need for investment and financial quantitative analysts. It provides data using API, R, Python, or Excel.

In this section, we will retrieve the Sunspot dataset from Quandl. We can use either an API or download the data manually in CSV format.

Let's first install the Quandl package using `pip`:

```
$ pip3 install Quandl
```

If you want to install the API, you can do so by downloading installers from `https://pypi.python.org/pypi/Quandl` or by running the preceding command.

> Using the API is free, but is limited to 50 API calls per day. If you require more API calls, you will have to request an authentication key. The code in this tutorial does not use a key. It should be simple to change the code to either use a key or read a downloaded CSV file. If you have difficulties, refer to the *Where to find help and references* section in `Chapter 1`, *Getting Started with Python Libraries*, or search through the Python docs at `https://docs.python.org/2/`.

Let's take a look at how to query data in a `pandas` DataFrame:

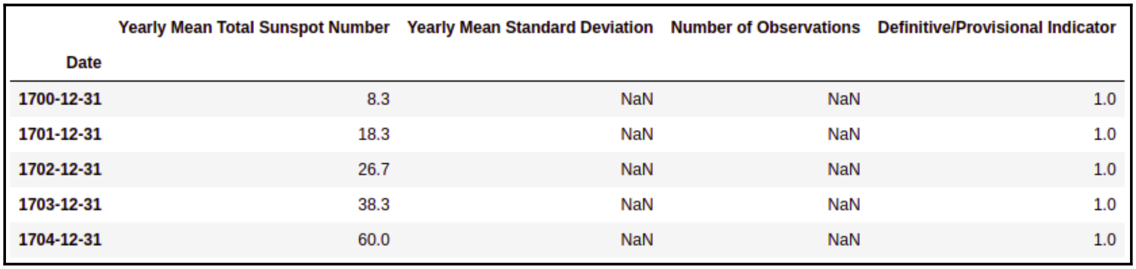
1. As a first step, we obviously have to download the data. After importing the Quandl API, get the data as follows:

```
import quandl
sunspots = quandl.get("SIDC/SUNSPOTS_A")
```

2. The `head()` and `tail()` methods have a purpose similar to that of the Unix commands with the same name. Select the first *n* and last *n* records of a DataFrame, where *n* is an integer parameter:

```
sunspots.head()
```

This results in the following output:

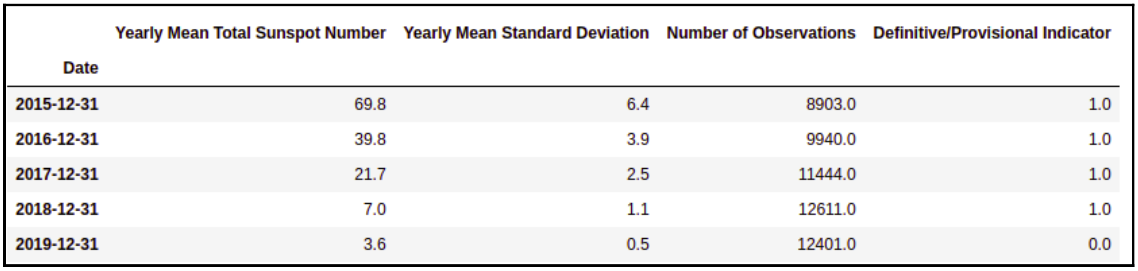


	Yearly Mean Total Sunspot Number	Yearly Mean Standard Deviation	Number of Observations	Definitive/Provisional Indicator
Date				
1700-12-31	8.3	NaN	NaN	1.0
1701-12-31	18.3	NaN	NaN	1.0
1702-12-31	26.7	NaN	NaN	1.0
1703-12-31	38.3	NaN	NaN	1.0
1704-12-31	60.0	NaN	NaN	1.0

Let's check out the `tail` function as follows:

```
sunspots.tail()
```

This results in the following output:



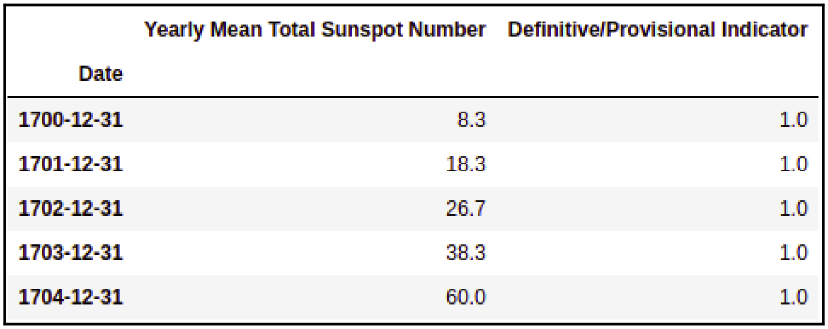
	Yearly Mean Total Sunspot Number	Yearly Mean Standard Deviation	Number of Observations	Definitive/Provisional Indicator
Date				
2015-12-31	69.8	6.4	8903.0	1.0
2016-12-31	39.8	3.9	9940.0	1.0
2017-12-31	21.7	2.5	11444.0	1.0
2018-12-31	7.0	1.1	12611.0	1.0
2019-12-31	3.6	0.5	12401.0	0.0

The `head()` and `tail()` methods give us the first and last five rows of the Sunspot data, respectively.

3. **Filtering columns**: `pandas` offers the ability to select columns. Let's select columns in a `pandas` DataFrame:

```
# Select columns
sunspots_filtered=sunspots[['Yearly Mean Total Sunspot
Number','Definitive/Provisional Indicator']]

# Show top 5 records
sunspots_filtered.head()
```

This results in the following output:

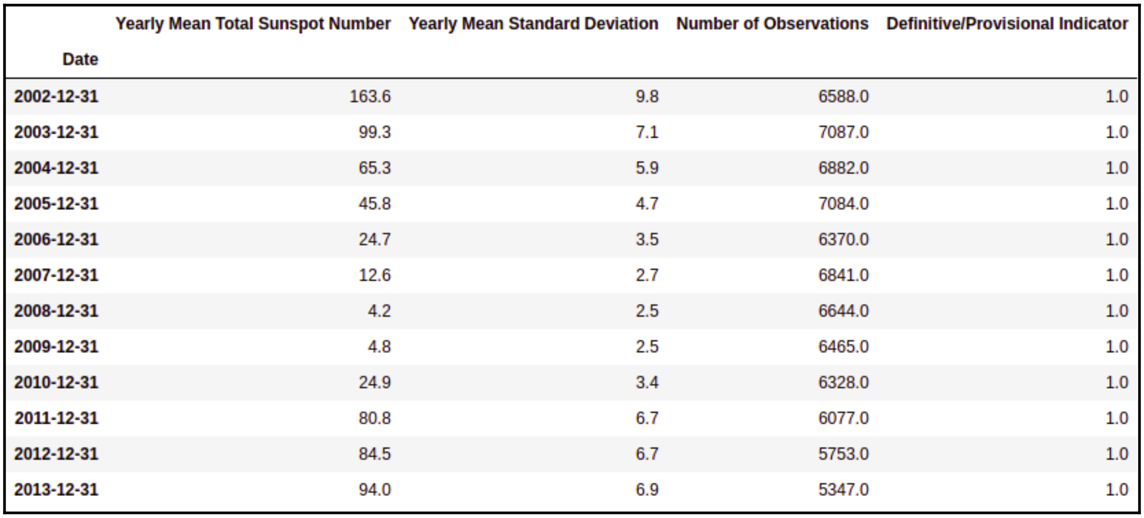


	Yearly Mean Total Sunspot Number	Definitive/Provisional Indicator
Date		
1700-12-31	8.3	1.0
1701-12-31	18.3	1.0
1702-12-31	26.7	1.0
1703-12-31	38.3	1.0
1704-12-31	60.0	1.0

4. **Filtering rows**: `pandas` offers the ability to select rows. Let's select rows in a `pandas` DataFrame:

```
# Select rows using index
sunspots["20020101": "20131231"]
```

This results in the following output:

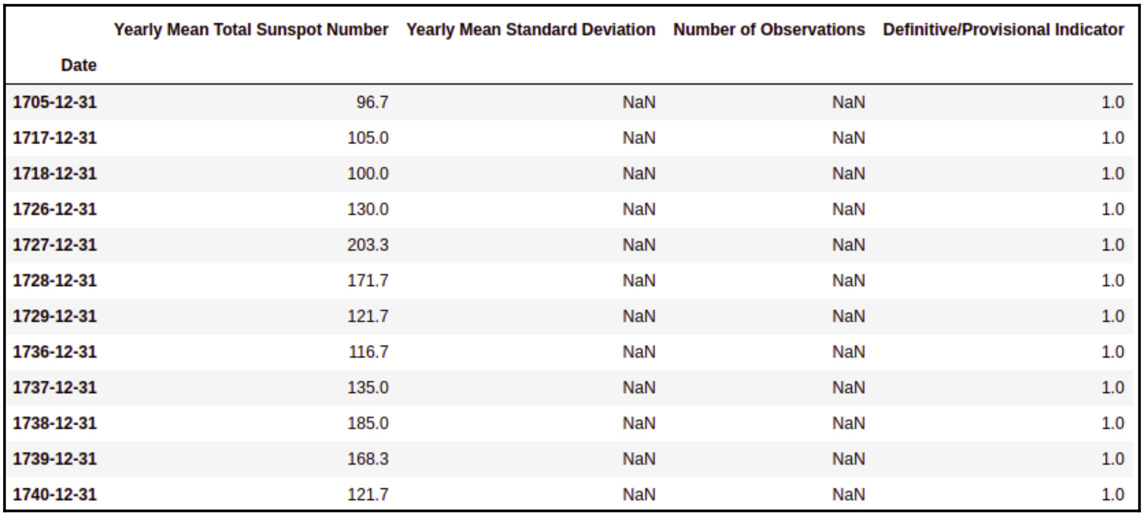


	Yearly Mean Total Sunspot Number	Yearly Mean Standard Deviation	Number of Observations	Definitive/Provisional Indicator
Date				
2002-12-31	163.6	9.8	6588.0	1.0
2003-12-31	99.3	7.1	7087.0	1.0
2004-12-31	65.3	5.9	6882.0	1.0
2005-12-31	45.8	4.7	7084.0	1.0
2006-12-31	24.7	3.5	6370.0	1.0
2007-12-31	12.6	2.7	6841.0	1.0
2008-12-31	4.2	2.5	6644.0	1.0
2009-12-31	4.8	2.5	6465.0	1.0
2010-12-31	24.9	3.4	6328.0	1.0
2011-12-31	80.8	6.7	6077.0	1.0
2012-12-31	84.5	6.7	5753.0	1.0
2013-12-31	94.0	6.9	5347.0	1.0

5. **Boolean filtering**: We can query data using Boolean conditions similar to the WHERE clause condition of SQL. Let's filter the data greater than the arithmetic mean:

```
# Boolean Filter
sunspots[sunspots['Yearly Mean Total Sunspot Number'] >
sunspots['Yearly Mean Total Sunspot Number'].mean()]
```

This results in the following output:



	Yearly Mean Total Sunspot Number	Yearly Mean Standard Deviation	Number of Observations	Definitive/Provisional Indicator
Date				
1705-12-31	96.7	NaN	NaN	1.0
1717-12-31	105.0	NaN	NaN	1.0
1718-12-31	100.0	NaN	NaN	1.0
1726-12-31	130.0	NaN	NaN	1.0
1727-12-31	203.3	NaN	NaN	1.0
1728-12-31	171.7	NaN	NaN	1.0
1729-12-31	121.7	NaN	NaN	1.0
1736-12-31	116.7	NaN	NaN	1.0
1737-12-31	135.0	NaN	NaN	1.0
1738-12-31	185.0	NaN	NaN	1.0
1739-12-31	168.3	NaN	NaN	1.0
1740-12-31	121.7	NaN	NaN	1.0

Describing pandas DataFrames

The pandas DataFrame has a dozen statistical methods. The following table lists these methods, along with a short description of each:

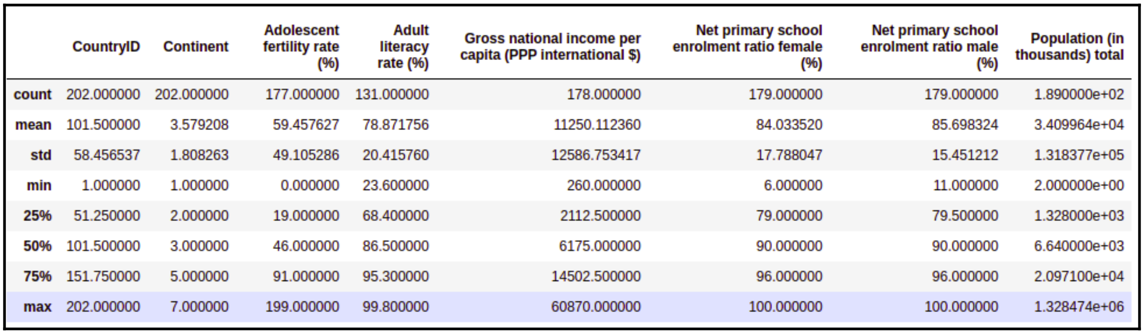
Method	**Description**
`describes`	This method returns a small table with descriptive statistics.
`count`	This method returns the number of non-NaN items.
`mad`	This method calculates the mean absolute deviation, which is a robust measure similar to standard deviation.
`median`	This method returns the median. This is equivalent to the value at the 50th percentile.
`min`	This method returns the minimum value.

`max`	This method returns the maximum value.
`mode`	This method returns the mode, which is the most frequently occurring value.
`std`	This method returns the standard deviation, which measures dispersion. It is the square root of the variance.
`var`	This method returns the variance.
`skew`	This method returns skewness. Skewness is indicative of the distribution symmetry.
`kurt`	This method returns kurtosis. Kurtosis is indicative of the distribution shape.

Using the same data used in the previous section, we will demonstrate these statistical methods:

```
# Describe the dataset
df.describe()
```

This results in the following output:

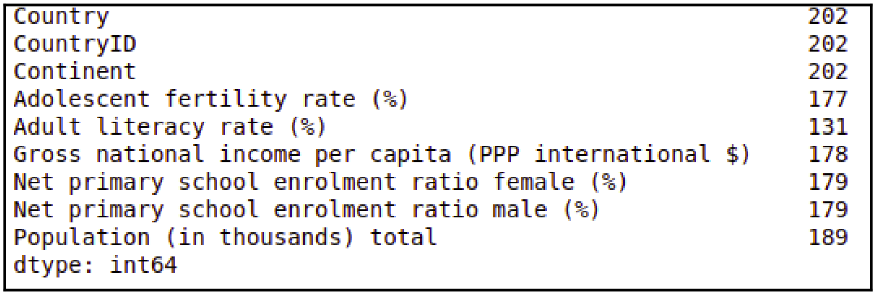


	CountryID	Continent	Adolescent fertility rate (%)	Adult literacy rate (%)	Gross national income per capita (PPP international $)	Net primary school enrolment ratio female (%)	Net primary school enrolment ratio male (%)	Population (in thousands) total
count	202.000000	202.000000	177.000000	131.000000	178.000000	179.000000	179.000000	1.890000e+02
mean	101.500000	3.579208	59.457627	78.871756	11250.112360	84.033520	85.698324	3.409964e+04
std	58.456537	1.808263	49.105286	20.415760	12586.753417	17.788047	15.451212	1.318377e+05
min	1.000000	1.000000	0.000000	23.600000	260.000000	6.000000	11.000000	2.000000e+00
25%	51.250000	2.000000	19.000000	68.400000	2112.500000	79.000000	79.500000	1.328000e+03
50%	101.500000	3.000000	46.000000	86.500000	6175.000000	90.000000	90.000000	6.640000e+03
75%	151.750000	5.000000	91.000000	95.300000	14502.500000	96.000000	96.000000	2.097100e+04
max	202.000000	7.000000	199.000000	99.800000	60870.000000	100.000000	100.000000	1.328474e+06

The `describe()` method will show most of the descriptive statistical measures for all columns:

```
# Count number of observation
df.count()
```

This results in the following output:

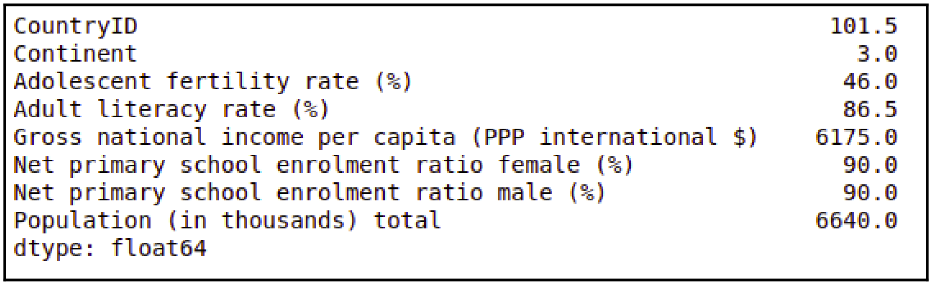


```
Country                                                 202
CountryID                                               202
Continent                                               202
Adolescent fertility rate (%)                           177
Adult literacy rate (%)                                 131
Gross national income per capita (PPP international $)  178
Net primary school enrolment ratio female (%)           179
Net primary school enrolment ratio male (%)             179
Population (in thousands) total                         189
dtype: int64
```

The `count()` method counts the number of observations in each column. It helps us to check the missing values in the dataset. Except for the initial three columns, all the columns have missing values. Similarly, you can compute the median, standard deviation, mean absolute deviation, variance, skewness, and kurtosis:

```
# Compute median of all the columns
df.median()
```

This results in the following output:

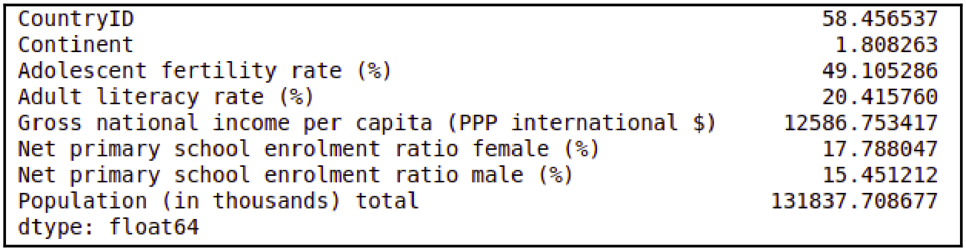


```
CountryID                                                101.5
Continent                                                  3.0
Adolescent fertility rate (%)                             46.0
Adult literacy rate (%)                                   86.5
Gross national income per capita (PPP international $)  6175.0
Net primary school enrolment ratio female (%)             90.0
Net primary school enrolment ratio male (%)               90.0
Population (in thousands) total                         6640.0
dtype: float64
```

We can compute deviation for all columns as follows:

```
# Compute the standard deviation of all the columns
df.std()
```

This results in the following output:



```
CountryID                                                    58.456537
Continent                                                     1.808263
Adolescent fertility rate (%)                                49.105286
Adult literacy rate (%)                                      20.415760
Gross national income per capita (PPP international $)    12586.753417
Net primary school enrolment ratio female (%)                17.788047
Net primary school enrolment ratio male (%)                  15.451212
Population (in thousands) total                          131837.708677
dtype: float64
```

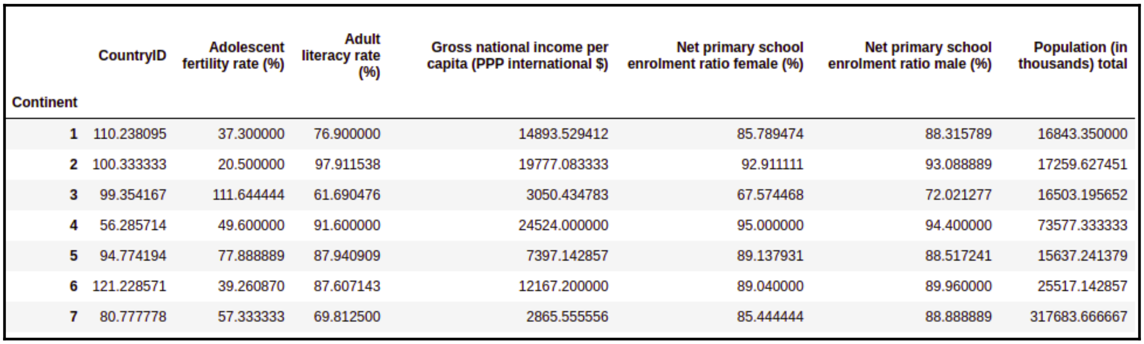
The preceding code example is computing the standard deviation for each numeric column.

Grouping and joining pandas DataFrame

Grouping is a kind of data aggregation operation. The grouping term is taken from a relational database. Relational database software uses the `group by` keyword to group similar kinds of values in a column. We can apply aggregate functions on groups such as mean, min, max, count, and sum. The `pandas` DataFrame also offers similar kinds of capabilities. Grouping operations are based on the split-apply-combine strategy. It first divides data into groups and applies the aggregate operation, such as mean, min, max, count, and sum, on each group and combines results from each group:

```
# Group By DataFrame on the basis of Continent column
df.groupby('Continent').mean()
```

This results in the following output:

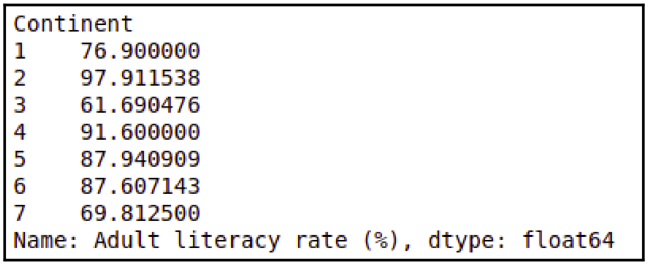


	CountryID	Adolescent fertility rate (%)	Adult literacy rate (%)	Gross national income per capita (PPP international $)	Net primary school enrolment ratio female (%)	Net primary school enrolment ratio male (%)	Population (in thousands) total
Continent							
1	110.238095	37.300000	76.900000	14893.529412	85.789474	88.315789	16843.350000
2	100.333333	20.500000	97.911538	19777.083333	92.911111	93.088889	17259.627451
3	99.354167	111.644444	61.690476	3050.434783	67.574468	72.021277	16503.195652
4	56.285714	49.600000	91.600000	24524.000000	95.000000	94.400000	73577.333333
5	94.774194	77.888889	87.940909	7397.142857	89.137931	88.517241	15637.241379
6	121.228571	39.260870	87.607143	12167.200000	89.040000	89.960000	25517.142857
7	80.777778	57.333333	69.812500	2865.555556	85.444444	88.888889	317683.666667

Let's now group the DataFrames based on literacy rates as well:

```
# Group By DataFrame on the basis of continent and select adult literacy
rate(%)
df.groupby('Continent').mean()['Adult literacy rate (%)']
```

This results in the following output:



```
Continent
1    76.900000
2    97.911538
3    61.690476
4    91.600000
5    87.940909
6    87.607143
7    69.812500
Name: Adult literacy rate (%), dtype: float64
```

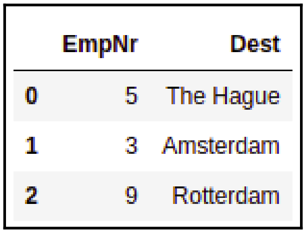
In the preceding example, the continent-wise average adult literacy rate in percentage was computed. You can also group based on multiple columns by passing a list of columns to the `groupby()` function.

Join is a kind of merge operation for tabular databases. The join concept is taken from the relational database. In relational databases, tables were normalized or broken down to reduce redundancy and inconsistency, and join is used to select the information from multiple tables. A data analyst needs to combine data from multiple sources. `pandas` also offers to join functionality to join multiple DataFrames using the `merge()` function.

To understand joining, we will take a taxi company use case. We are using two files: `dest.csv` and `tips.csv`. Every time a driver drops any passenger at their destination, we will insert a record (employee number and destination) into the `dest.csv` file. Whenever drivers get a tip, we insert the record (employee number and tip amount) into the `tips.csv` file. You can download both the files from the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Python-Data-Analysis-Third-Edition/Ch2`:

```
# Import pandas
import pandas as pd

# Load data using read_csv()
dest = pd.read_csv("dest.csv")

# Show DataFrame
dest.head()
```

This results in the following output:

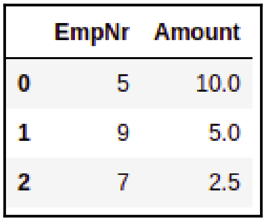


	EmpNr	Dest
0	5	The Hague
1	3	Amsterdam
2	9	Rotterdam

In the preceding code block, we have read the `dest.csv` file using the `read_csv()` method:

```
# Load data using read_csv()
tips = pd.read_csv("tips.csv")

# Show DataFrame
tips.head()
```

This results in the following output:



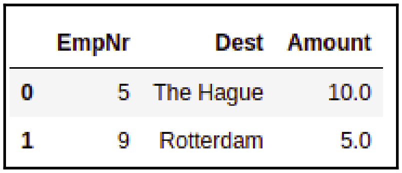
	EmpNr	Amount
0	5	10.0
1	9	5.0
2	7	2.5

In the preceding code block, we have read the `tips.csv` file using the `read_csv()` method. We will now check out the various types of joins as follows:

- **Inner join**: Inner join is equivalent to the intersection operation of a set. It will select only common records in both the DataFrames. To perform inner join, use the `merge()` function with both the DataFrames and common attribute on the parameter and inner value to show the parameter. The `on` parameter is used to provide the common attribute based on the join will be performed and `how` defines the type of join:

```
# Join DataFrames using Inner Join
df_inner= pd.merge(dest, tips, on='EmpNr', how='inner')
df_inner.head()
```

This results in the following output:

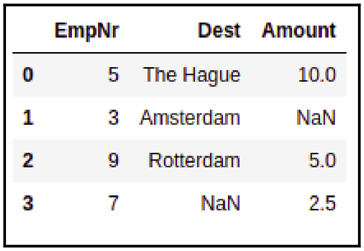


	EmpNr	Dest	Amount
0	5	The Hague	10.0
1	9	Rotterdam	5.0

- **Full outer join**: Outer join is equivalent to a union operation of the set. It merges the right and left DataFrames. It will have all the records from both DataFrames and fills NaNs where the match will not be found:

```
# Join DataFrames using Outer Join
df_outer= pd.merge(dest, tips, on='EmpNr', how='outer')
df_outer.head()
```

This results in the following output:

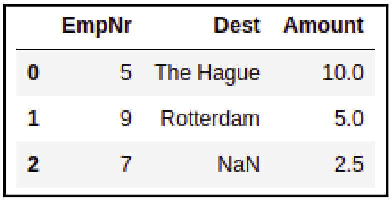


	EmpNr	Dest	Amount
0	5	The Hague	10.0
1	3	Amsterdam	NaN
2	9	Rotterdam	5.0
3	7	NaN	2.5

- **Right outer join**: In the right outer join, all the records from the right side of the DataFrame will be selected. If the matched records cannot be found in the left DataFrame, then it is filled with NaNs:

```
# Join DataFrames using Right Outer Join
df_right= pd.merge(dest, tips, on='EmpNr', how='right')
df_right.head()
```

This results in the following output:

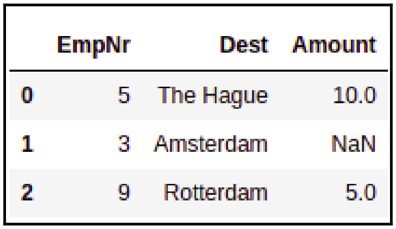


	EmpNr	Dest	Amount
0	5	The Hague	10.0
1	9	Rotterdam	5.0
2	7	NaN	2.5

- **Left outer join**: In the left outer join, all the records from the left side of the DataFrame will be selected. If the matched records cannot be found in the right DataFrame, then it is filled with NaNs:

```
# Join DataFrames using Left Outer Join
df_left= pd.merge(dest, tips, on='EmpNr', how='left')
df_left.head()
```

This results in the following output:



	EmpNr	Dest	Amount
0	5	The Hague	10.0
1	3	Amsterdam	NaN
2	9	Rotterdam	5.0

We will now move on to checking out missing values in the datasets.

Working with missing values

Most real-world datasets are messy and noisy. Due to their messiness and noise, lots of values are either faulty or missing. `pandas` offers lots of built-in functions to deal with missing values in DataFrames:

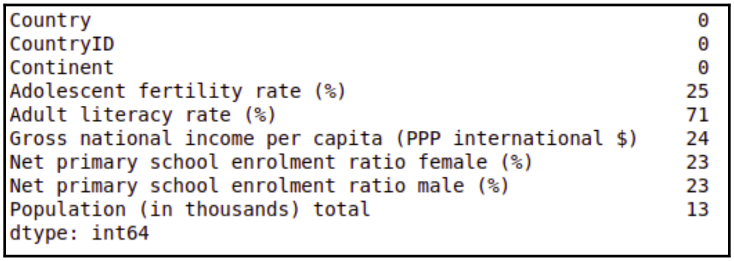
- **Check missing values in a DataFrame**: `pandas`' `isnull()` function checks for the existence of null values and returns `True` or `False`, where `True` is for null and `False` is for not-null values. The `sum()` function will sum all the `True` values and returns the count of missing values. We have tried two ways to count the missing values; both show the same output:

```
# Count missing values in DataFrame
pd.isnull(df).sum()
```

The following is the second method:

```
df.isnull().sum()
```

This results in the following output:

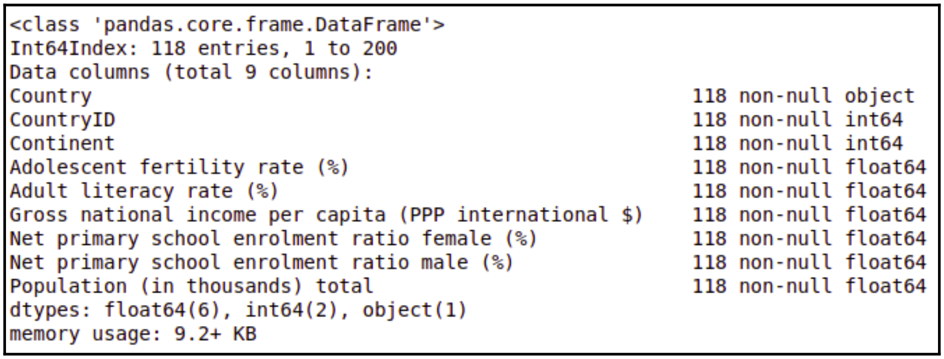


```
Country                                                0
CountryID                                              0
Continent                                              0
Adolescent fertility rate (%)                         25
Adult literacy rate (%)                               71
Gross national income per capita (PPP international $)   24
Net primary school enrolment ratio female (%)         23
Net primary school enrolment ratio male (%)           23
Population (in thousands) total                       13
dtype: int64
```

- **Drop missing values**: A very naive approach to deal with missing values is to drop them for analysis purposes. `pandas` has the `dropna()` function to drop or delete such observations from the DataFrame. Here, the `inplace=True` attribute makes the changes in the original DataFrame:

```
# Drop all the missing values
df.dropna(inplace=True)

df.info()
```

This results in the following output:



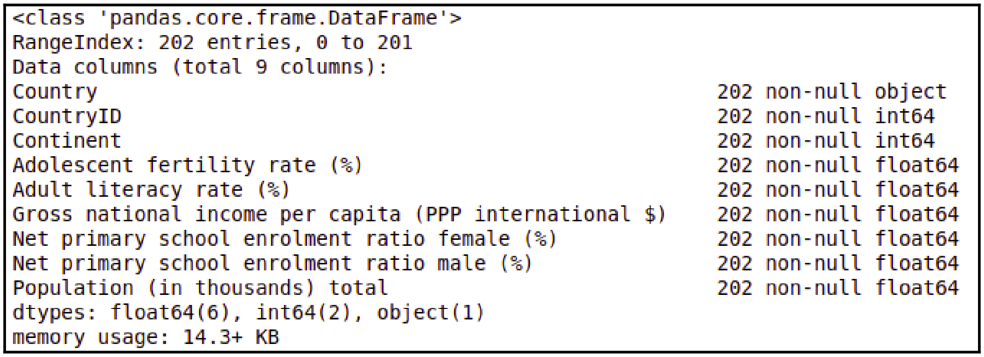
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 118 entries, 1 to 200
Data columns (total 9 columns):
Country                                                  118 non-null object
CountryID                                                118 non-null int64
Continent                                                118 non-null int64
Adolescent fertility rate (%)                            118 non-null float64
Adult literacy rate (%)                                  118 non-null float64
Gross national income per capita (PPP international $)   118 non-null float64
Net primary school enrolment ratio female (%)            118 non-null float64
Net primary school enrolment ratio male (%)              118 non-null float64
Population (in thousands) total                          118 non-null float64
dtypes: float64(6), int64(2), object(1)
memory usage: 9.2+ KB
```

Here, the number of observations is reduced to 118 from 202.

- **Fill the missing values**: Another approach is to fill the missing values with zero, mean, median, or constant values:

```
# Fill missing values with 0
df.fillna(0,inplace=True)

df.info()
```

This results in the following output:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 202 entries, 0 to 201
Data columns (total 9 columns):
Country                                               202 non-null object
CountryID                                             202 non-null int64
Continent                                             202 non-null int64
Adolescent fertility rate (%)                         202 non-null float64
Adult literacy rate (%)                               202 non-null float64
Gross national income per capita (PPP international $)   202 non-null float64
Net primary school enrolment ratio female (%)         202 non-null float64
Net primary school enrolment ratio male (%)           202 non-null float64
Population (in thousands) total                       202 non-null float64
dtypes: float64(6), int64(2), object(1)
memory usage: 14.3+ KB
```

Here, we have filled the missing values with 0. This is all about handling missing values.

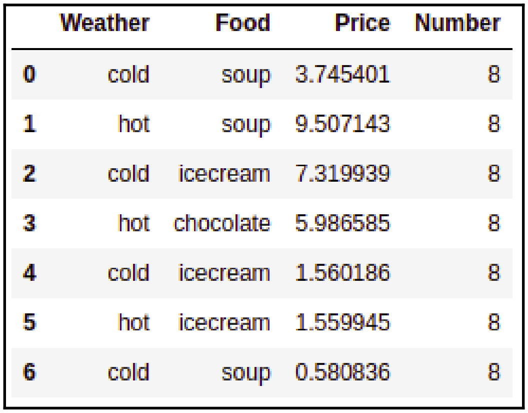
In the next section, we will focus on pivot tables.

Creating pivot tables

A pivot table is a summary table. It is the most popular concept in Excel. Most data analysts use it as a handy tool to summarize theire results. `pandas` offers the `pivot_table()` function to summarize DataFrames. A DataFrame is summarized using an aggregate function, such as mean, min, max, or sum. You can download the dataset from the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Python-Data-Analysis-Third-Edition/Ch2`:

```
# Import pandas
import pandas as pd

# Load data using read_csv()
purchase = pd.read_csv("purchase.csv")

# Show initial 10 records
purchase.head(10)
```

This results in the following output:



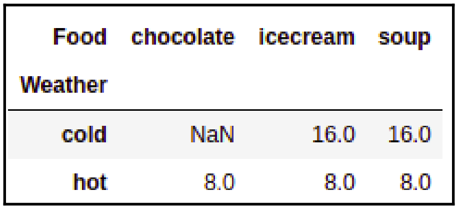
	Weather	Food	Price	Number
0	cold	soup	3.745401	8
1	hot	soup	9.507143	8
2	cold	icecream	7.319939	8
3	hot	chocolate	5.986585	8
4	cold	icecream	1.560186	8
5	hot	icecream	1.559945	8
6	cold	soup	0.580836	8

In the preceding code block, we have read the `purchase.csv` file using the `read_csv()` method.

Now, we will summarize the dataframe using the following code:

```
# Summarise dataframe using pivot table
pd.pivot_table(purchase,values='Number', index=['Weather',],
                    columns=['Food'], aggfunc=np.sum)
```

This results in the following output:



Food	chocolate	icecream	soup
Weather			
cold	NaN	16.0	16.0
hot	8.0	8.0	8.0

In the preceding example, the `purchase` DataFrame is summarized. Here, `index` is the `Weather` column, `columns` is the `Food` column, and `values` is the aggregated sum of the `Number` column. `aggfun` is initialized with the `np.sum` parameter. It's time to learn how to deal with dates in `pandas` DataFrames.

Dealing with dates

Dealing with dates is messy and complicated. You can recall the Y2K bug, the upcoming 2038 problem, and time zones dealing with different problems. In time-series datasets, we come across dates. `pandas` offers date ranges, resamples time-series data, and performs date arithmetic operations.

Create a range of dates starting from January 1, 2020, lasting for 45 days, as follows:

```
pd.date_range('01-01-2000', periods=45, freq='D')

Output:
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10', '2000-01-11', '2000-01-12',
               '2000-01-13', '2000-01-14', '2000-01-15', '2000-01-16',
               '2000-01-17', '2000-01-18', '2000-01-19', '2000-01-20',
               '2000-01-21', '2000-01-22', '2000-01-23', '2000-01-24',
               '2000-01-25', '2000-01-26', '2000-01-27', '2000-01-28',
               '2000-01-29', '2000-01-30', '2000-01-31', '2000-02-01',
               '2000-02-02', '2000-02-03', '2000-02-04', '2000-02-05',
               '2000-02-06', '2000-02-07', '2000-02-08', '2000-02-09',
               '2000-02-10', '2000-02-11', '2000-02-12', '2000-02-13',
               '2000-02-14'],
              dtype='datetime64[ns]', freq='D')
```

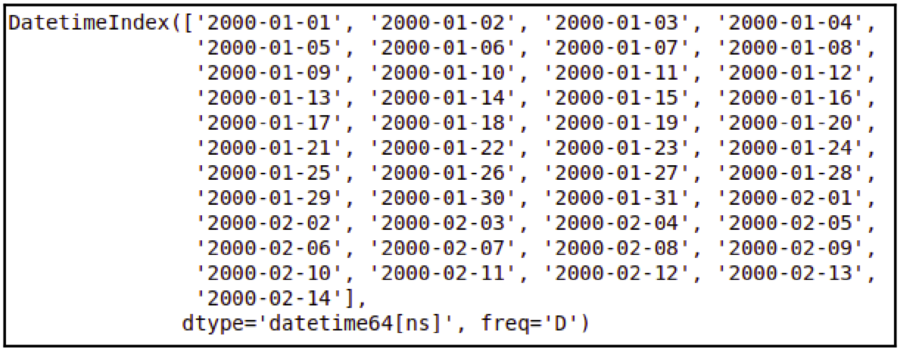
January has less than 45 days, so the end date falls in February, as you can check for yourself.

`date_range()` freq parameters can take values such as `B` for business day frequency, `W` for weekly frequency, `H` for hourly frequency, `M` for minute frequency, `S` for second frequency, `L` for millisecond frequency, and `U` for microsecond frequency. For more details, you can refer to the official documentation at `https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html`.

- **pandas date range:** The `date_range()` function generates sequences of date and time with a fixed-frequency interval:

  ```
  # Date range function
  pd.date_range('01-01-2000', periods=45, freq='D')
  ```

This results in the following output:



```
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10', '2000-01-11', '2000-01-12',
               '2000-01-13', '2000-01-14', '2000-01-15', '2000-01-16',
               '2000-01-17', '2000-01-18', '2000-01-19', '2000-01-20',
               '2000-01-21', '2000-01-22', '2000-01-23', '2000-01-24',
               '2000-01-25', '2000-01-26', '2000-01-27', '2000-01-28',
               '2000-01-29', '2000-01-30', '2000-01-31', '2000-02-01',
               '2000-02-02', '2000-02-03', '2000-02-04', '2000-02-05',
               '2000-02-06', '2000-02-07', '2000-02-08', '2000-02-09',
               '2000-02-10', '2000-02-11', '2000-02-12', '2000-02-13',
               '2000-02-14'],
              dtype='datetime64[ns]', freq='D')
```

- `to_datetime():to_datetime()` converts a timestamp string into datetime:

```
# Convert argument to datetime
pd.to_datetime('1/1/1970')

Output:  Timestamp('1970-01-01 00:00:00')
```

- We can convert a timestamp string into a datetime object in the specified format:

```
# Convert argument to datetime in specified format
pd.to_datetime(['20200101', '20200102'], format='%Y%m%d')

Output:
DatetimeIndex(['2020-01-01', '2020-01-02'], dtype='datetime64[ns]',
freq=None)
```

- **Handling an unknown format string**: Unknown input format can cause value errors. We can handle this by using an errors parameter with `coerce`. `coerce` will set invalid strings to NaT:

```
# Value Error
pd.to_datetime(['20200101', 'not a date'])

Output:
ValueError: ('Unknown string format:', 'not a date')

# Handle value error
pd.to_datetime(['20200101', 'not a date'], errors='coerce')

Output:
DatetimeIndex(['2020-01-01', 'NaT'], dtype='datetime64[ns]',
freq=None)
```

In the preceding example, the second date is still not valid and cannot be converted into a datetime object. The errors parameter helped us to handle such errors by inputting the value `NaT` (not a time).

Summary

In this chapter, we have explored the NumPy and pandas libraries. Both libraries help deal with arrays and DataFrames. NumPy arrays have the capability to deal with n-dimensional arrays. We have learned about various array properties and operations. Our main focus is on data types, data type as an object, reshaping, stacking, splitting, slicing, and indexing.

We also focused on the `pandas` library for Python data analysis. We saw how `pandas` mimics the relational database table functionality. It offers functionality to query, aggregate, manipulate, and join data efficiently.

NumPy and `pandas` work well together as a tool and make it possible to perform basic data analysis. At this point, you might be tempted to think that `pandas` is all we need for data analysis. However, there is more to data analysis than meets the eye.

Having picked up the fundamentals, it's time to proceed to data analysis with the commonly used statistics functions in `Chapter 3`, *Statistics*. This includes the usage of statistical concepts.

You are encouraged to read the books mentioned in the *References* section for exploring NumPy and `pandas` in further detail and depth.

References

- Ivan Idris, *NumPy Cookbook – Second Edition*, Packt Publishing, 2015.
- Ivan Idris, *Learning NumPy Array*, Packt Publishing, 2014.
- Ivan Idris, *NumPy: Beginner's Guide – Third Edition*, Packt Publishing, 2015.
- L. (L.-H.) Chin and T. Dutta, *NumPy Essentials*, Packt Publishing, 2016.
- T. Petrou, *Pandas Cookbook*, Packt Publishing, 2017.
- F. Anthony, *Mastering pandas*, Packt Publishing, 2015.
- M. Heydt, *Mastering pandas for Finance*, Packt Publishing, 2015.
- T. Hauck, *Data-Intensive Apps with pandas How-to*, Packt Publishing, 2013.
- M. Heydt, *Learning pandas*, Packt Publishing, 2015.

3
Statistics

Exploratory data analysis (**EDA**) is the first step toward data analysis and building a machine learning model. Statistics provide fundamental knowledge and a set of tools for exploratory or descriptive data analysis. This chapter is designed to make you data-ready. For any kind of data professional role, you need to understand real-world data that is generally noisy, has missing values, and is collected from various sources.

Before performing any kind of preprocessing and analysis, you need to get familiar with the data present, and statistics is the only tool that will help you here. This makes statistics a primary and very necessary skill for data professionals, helping them gain initial insights and an understanding of the data. For example, the arithmetic mean of the monthly working hours of an employee can help us to understand the load of an employee in an organization. Similarly, the standard deviation of monthly working hours can help us to infer the range of working hours. Correlation between two variables such as blood pressure and age of patients can help us understand the relationship between blood pressure and age. Sampling methods can be useful in any kind of primary data collection. We can also perform parametric and non-parametric hypothesis tests to infer facts about the population.

In this chapter, we will cover the following topics:

- Understanding attributes and their types
- Measuring central tendency
- Measuring dispersion
- Skewness and kurtosis
- Understanding relationships using covariance and correlation coefficients
- Central limit theorem
- Collecting samples
- Performing parametric tests
- Performing non-parametric tests

Technical requirements

For this chapter, the following technical information is available:

- You can find the code and the dataset at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter03`.
- All the code blocks are available in `ch3.ipynb`.
- In this chapter, we will use the NumPy, Pandas, and SciPy Python libraries.

Understanding attributes and their types

Data is the collection of raw facts and statistics such as numbers, words, and observations. An attribute is a column or data field or series that represents the characteristics of an object and is also known as a variable, a feature, or a dimension. Statisticians use the term *variable*, while machine learning engineers prefer the term *feature*. The term *dimension* is used in data warehousing, while database professionals use the term *attribute*.

Types of attributes

The data type of attributes is more crucial for data analysis because certain situations require certain data types. The data type of attributes helps analysts select the correct method for data analysis and visualization plots. The following list shows the various attributes:

1. **Nominal attributes:** Nominal refers to names or labels of categorized variables. The value of a nominal attribute can be the symbol or name of items. The values are categorical, qualitative, and unordered in nature such as product name, brand name, zip code, state, gender, and marital status. Finding the mean and median values of qualitative and categorical values will not make any sense but data analysts can calculate the mode, which is the most commonly occurring value.
2. **Ordinal attributes:** Ordinal refers to names or labels with a meaningful order or ranking, but the magnitude of values is not known. These types of attributes measure subjective qualities alone. That is why they are used in surveys for customer satisfaction ratings, product ratings, and movie rating reviews. Customer satisfaction ratings appear in the following order:
    - 1: Very dissatisfied
    - 2: Somewhat dissatisfied

- 3: Neutral
- 4: Satisfied
- 5: Very satisfied

Another example could be the size of a drink: small, medium, or large. Ordinal attributes can only be measured via the mode and the median. The mean cannot be calculated for ordinal attributes because of their qualitative nature. Ordinal attributes can also be recreated by discretization of a quantitative variable by dividing their values in a range of finite numbers.

3. **Numeric attributes:** A numeric attribute is quantitatively presented as integer or real values. Numeric attributes can be of two types: interval-scaled or ratio-scaled.

Interval-scaled attributes are measured on an ordered scale of equal-sized units. The meaningful difference between the two values of an interval-scaled attribute can be calculated, and this allows a comparison between the two values—for example, the birth year, and the temperature in °C. The main problem with interval-scaled attribute values is that they don't have a "true zero"—for example, if the temperature in °C is 0 then it doesn't mean that temperature doesn't exist. Interval-scaled data can add and subtract but can't multiply and divide because of no true zero. We can also calculate the mean value of an interval-scaled attribute, in addition to the median and mode.

Ratio-scaled attributes are measured on an ordered scale of equal-sized units, similar to an interval scale with an inherent zero point. Examples of ratio-scaled attributes are height, weight, latitude, longitude, years of experience, and the number of words in a document. We can perform multiplication and division, and calculate the difference between ratio-scaled values. We can also compute central tendency measures such as mean, median, and mode. The Celsius and Fahrenheit temperature scales are measured on an interval scale, while the Kelvin temperature scale is measured on a ratio scale because it has a true zero point.

Discrete and continuous attributes

There are various ways to classify attributes. In the previous sub-section, we have seen nominal, ordinal, and numeric attributes. In this sub-section, we will see another type of attribute classification. Here, we will talk about discrete or continuous attributes. A discrete variable accepts only a countable finite number, such as how many students are present in a class, how many cars are sold, and how many books are published. It can be obtained by counting numbers. A continuous variable accepts an infinite number of possible values, such as the weight and height of students. It can be obtained by measuring.

A discrete variable accepts integral values, while a continuous variable accepts real values. In other words, we can say a discrete variable accepts values whose fraction doesn't make sense, whereas a continuous variable accepts values whose fraction makes sense. A discrete attribute uses a limited number of values, while a continuous attribute uses an unlimited number of values.

After understanding the attributes and their types, it's time to focus on basic statistical descriptions such as central tendency measures.

Measuring central tendency

Central tendency is the trend of values clustered around the averages such as the mean, mode, and median values of data. The main objective of central tendency is to compute the center-leading value of observations. Central tendency determines the descriptive summary and provides quantitative information about a group of observations. It has the capability to represent a whole set of observations. Let's see each type of central tendency measure in detail in the coming sections.

Mean

The mean value is the arithmetic mean or average, which is computed by the sum of observations divided by the number of observations. It is sensitive to outliers and noise, with the result that whenever uncommon or unusual values are added to a group, its mean gets deviated from the typical central value. Assume x_1, x_2, . . . , x_N is *N* observations. The formula for the mean of these values is shown here:

$$Mean = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Let's compute the mean value of the communication skill score column using the `pandas` library, as follows:

```
# Import pandas library
import pandas as pd

# Create dataframe
sample_data = {'name': ['John', 'Alia', 'Ananya', 'Steve', 'Ben'],
               'gender': ['M', 'F', 'F', 'M', 'M'],
               'communication_skill_score': [40, 45, 23, 39, 39],
               'quantitative_skill_score': [38, 41, 42, 48, 32]}

data = pd.DataFrame(sample_data, columns = ['name', 'gender',
'communcation_skill_score', 'quantitative_skill_score'])

# find mean of communication_skill_score column
data['communcation_skill_score'].mean(axis=0)

Output:
37.2
```

In the preceding code block, we have created one DataFrame named `data` that has four columns (`name`, `gender`, `communication_skill_score`, and `quantitative_skill_score`) and computed the mean using the `mean(axis=0)` function. Here, `axis=0` represents the mean along the rows.

Mode

The mode is the highest-occurring item in a group of observations. The mode value occurs frequently in data and is mostly used for categorical values. If all the values in a group are unique or non-repeated, then there is no mode. It is also possible that more than one value has the same occurrence frequency. In such cases, there can be multiple modes.

Let's compute the mode value of the communication skill score column using the `pandas` library, as follows:

```
# find mode of communication_skill_score column
data['communcation_skill_score'].mode()

Output:
39
```

In the preceding code block, we have computed the mode of the communication skill score column using the mode() function. Let's compute another central tendency measure: the median.

Median

The median is the midpoint or middle value in a group of observations. It is also called the 50th percentile. The median is less affected by outliers and noise than the mean, and that is why it is considered a more suitable statistic measure for reporting. It is much near to a typical central value. Let's compute the median value of the communication skill score column using the pandas library, as follows:

```
# find median of communication_skill_score column
data['communcation_skill_score'].median()

Output:
39.0
```

In the preceding code block, we have computed the median of the communication skill score column using the median() function. Let's understand dispersion measures and compute them in the next section.

Measuring dispersion

As we have seen, central tendency presents the middle value of a group of observations but does not provide the overall picture of an observation. Dispersion metrics measure the deviation in observations. The most popular dispersion metrics are range, **interquartile range** (**IQR**), variance, and standard deviation. These dispersion metrics value the variability in observations or the spread of observations. Let's see each dispersion measure in detail, as follows:

- **Range:** The range is the difference between the maximum and minimum value of an observation. It is easy to compute and easy to understand. Its unit is the same as the unit of observations. Let's compute the range of communication skill scores, as follows:

```
column_range=data['communcation_skill_score'].max()-
data['communcation_skill_score'].min()
print(column_range)

Output:
22
```

In the preceding code block, we have computed the range of communication skill scores by finding the difference between the maximum and minimum scores. The maximum and minimum scores were computed using the `max()` and `min()` functions.

- **IQR:** IQR is the difference between the third and first quartiles. It is easy to compute and easy to understand. Its unit is the same as the unit of observations. It measures the middle 50% in the observation. It represents the range where most of the observation lies. IQR is also known as midspread or middle 50%, or H-spread. Let's compute the IQR of communication skill scores, as follows:

```
# First Quartile
q1 = data['communcation_skill_score'].quantile(.25)

# Third Quartile
q3 = data['communcation_skill_score'].quantile(.75)

# Inter Quartile Ratio
iqr=q3-q1
print(iqr)

Output:
1.0
```

In the preceding code block, we have computed the IQR of communication skill scores by finding the difference between the first and third quartile of scores. Both the first and third quartile scores were computed using the `quantile(.25)` and `quantile(.75)` functions.

- **Variance:** The variance measures the deviation from the mean. It is the average value of the squared difference between observed values and the mean. The main problem with the variance is its unit of measurement because of squaring the difference between observations and mean. Let's assume x_1 , x_2, . . . , x_N are *N* observations. The formula for the variance of these values will then be the following:

$$Variance = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

Let's compute the variance of communication skill scores, as follows:

```
# Variance of communication_skill_score
data['communcation_skill_score'].var()
```

```
Output:
69.2
```

In the preceding code block, we have computed the variance of communication skill scores using the `var()` function.

- **Standard deviation:** This is the square root of the variance. Its unit is the same as for the original observations. This makes it easier for an analyst to evaluate the exact deviation from the mean. The lower value of standard deviation represents the lesser distance of observations from the mean; this means observations are less widely spread. The higher value of standard deviation represents a large distance of observations from the mean—that is, observations are widely spread. Standard deviation is mathematically represented by the Greek letter sigma (Σ). Assume x_1, x_2, . . . , x_N are *N* observations. The formula for the standard deviation of these values is the following:

$$StandardDeviation = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

Let's compute the standard deviation of communication skill scores, as follows:

```
# Standard deviation of communication_skill_score
data['communcation_skill_score'].std()

Output:
8.318653737234168
```

In the preceding code block, we have computed the standard deviation of communication skill scores using the `std()` function.

We can also try to describe the function to get all the summary statistics in a single command. The `describe()` function returns the count, mean, standard deviation, first quartile, median, third quartile, and minimum and maximum values for each numeric column in the DataFrame, and is illustrated in the following code block:

```
# Describe dataframe
data.describe()

Output:
      communcation_skill_score quantitative_skill_score
count         5.000000         5.000000
mean         37.200000         40.200000
std           8.318654         5.848077
min          23.000000         32.000000
25%          39.000000         38.000000
50%          39.000000         41.000000
75%          40.000000         42.000000
max          45.000000         48.000000
```

In the preceding code block, we have generated a descriptive statistics summary of data using the `describe()` method.

Skewness and kurtosis

Skewness measures the symmetry of a distribution. It shows how much the distribution deviates from a normal distribution. Its values can be zero, positive, and negative. A zero value represents a perfectly normal shape of a distribution. Positive skewness is shown by the tails pointing toward the right—that is, outliers are skewed to the right and data stacked up on the left. Negative skewness is shown by the tails pointing toward the left—that is, outliers are skewed to the left and data stacked up on the right. Positive skewness occurs when the mean is greater than the median and the mode. Negative skewness occurs when the mean is less than the median and mode. Let's compute skewness in the following code block:

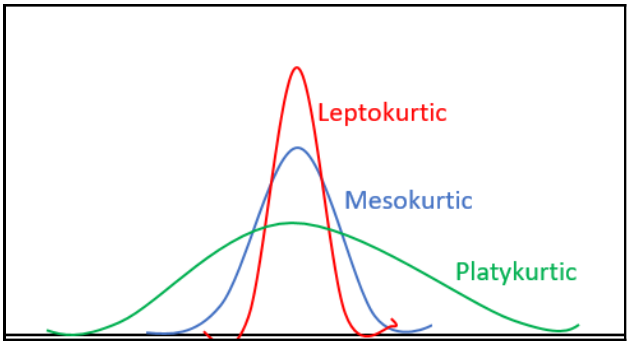
```
# skewness of communication_skill_score column
data['communcation_skill_score'].skew()

Output:
-1.704679180800373
```

In the preceding code block, we have computed the skewness of the communication skill score column using the `skew()` method.

Kurtosis measures the tailedness (thickness of tail) compared to a normal distribution. High kurtosis is heavy-tailed, which means more outliers are present in the observations, and low values of kurtosis are light-tailed, which means fewer outliers are present in the observations. There are three types of kurtosis shapes: mesokurtic, platykurtic, and leptokurtic. Let's define them one by one, as follows:

- A normal distribution having zero kurtosis is known as a mesokurtic distribution.
- A platykurtic distribution has a negative kurtosis value and is thin-tailed compared to a normal distribution.
- A leptokurtic distribution has a kurtosis value greater than 3 and is fat-tailed compared to a normal distribution.

Let's see the type of kurtosis shapes in the following diagram:



A histogram is an effective medium to present skewness and kurtosis. Let's compute the kurtosis of the communication skill score column, as follows:

```
# kurtosis of communication_skill_score column
data['communcation_skill_score'].kurtosis()

Output:
3.6010641852384015
```

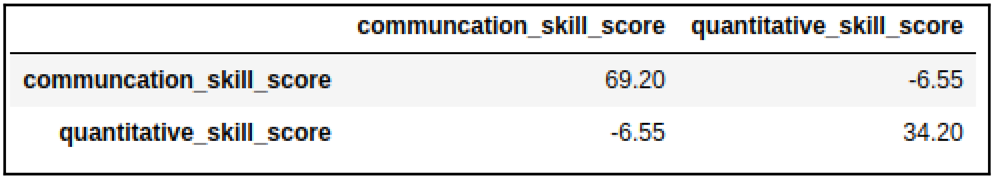
In the preceding code block, we have computed the kurtosis of the communication skill score column using the `kurtosis()` method.

Understanding relationships using covariance and correlation coefficients

Measuring the relationship between variables will be helpful for data analysts to understand the dynamics between variables—for example, an HR manager needs to understand the strength of the relationship between employee performance score and satisfaction score. Statistics offers two measures of covariance and correlation to understand the relationship between variables. Covariance measures the relationship between a pair of variables. It shows the degree of change in the variables—that is, how the change in one variable affects the other variable. Its value ranges from -infinity to + infinity. The problem with covariance is that it does not provide effective conclusions because it is not normalized. Let's find the relationship between the communication and quantitative skill score using covariance, as follows:

```
# Covariance between columns of dataframe
data.cov()
```

This results in the following output:



	communcation_skill_score	quantitative_skill_score
communcation_skill_score	69.20	-6.55
quantitative_skill_score	-6.55	34.20

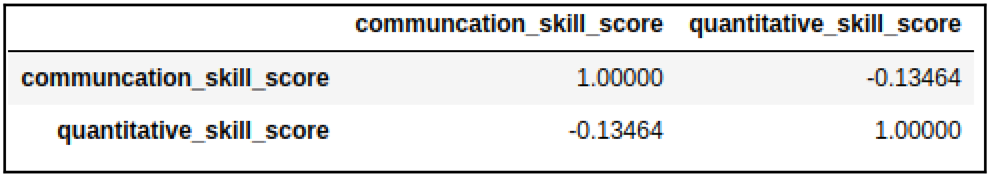
In the preceding code block, covariance is computed using the `cov()` method. Here, the output of this method is the covariance matrix.

Pearson's correlation coefficient

Correlation shows how variables are correlated with each other. Correlation offers a better understanding than covariance and is a normalized version of covariance. Correlation ranges from -1 to 1. A negative value represents the increase in one variable, causing a decrease in other variables or variables to move in the same direction. A positive value represents the increase in one variable, causing an increase in another variable, or a decrease in one variable causes decreases in another variable. A zero value means that there is no relationship between the variable or that variables are independent of each other. Have a look at the following code snippet:

```
# Correlation between columns of dataframe
data.corr(method ='pearson')
```

This results in the following output:



	communcation_skill_score	quantitative_skill_score
communcation_skill_score	1.00000	-0.13464
quantitative_skill_score	-0.13464	1.00000

The `'method'` parameter can take one of the following three parameters:

- `pearson`: Standard correlation coefficient
- `kendall`: Kendall's tau correlation coefficient
- `spearman`: Spearman's rank correlation coefficient

Spearman's rank correlation coefficient

Spearman's rank correlation coefficient is Pearson's correlation coefficient on the ranks of the observations. It is a non-parametric measure for rank correlation. It assesses the strength of the association between two ranked variables. Ranked variables are ordinal numbers, arranged in order. First, we rank the observations and then compute the correlation of ranks. It can apply to both continuous and discrete ordinal variables. When the distribution of data is skewed or an outlier is affected, then Spearman's rank correlation is used instead of Pearson's correlation because it doesn't have any assumptions for data distribution.

Kendall's rank correlation coefficient

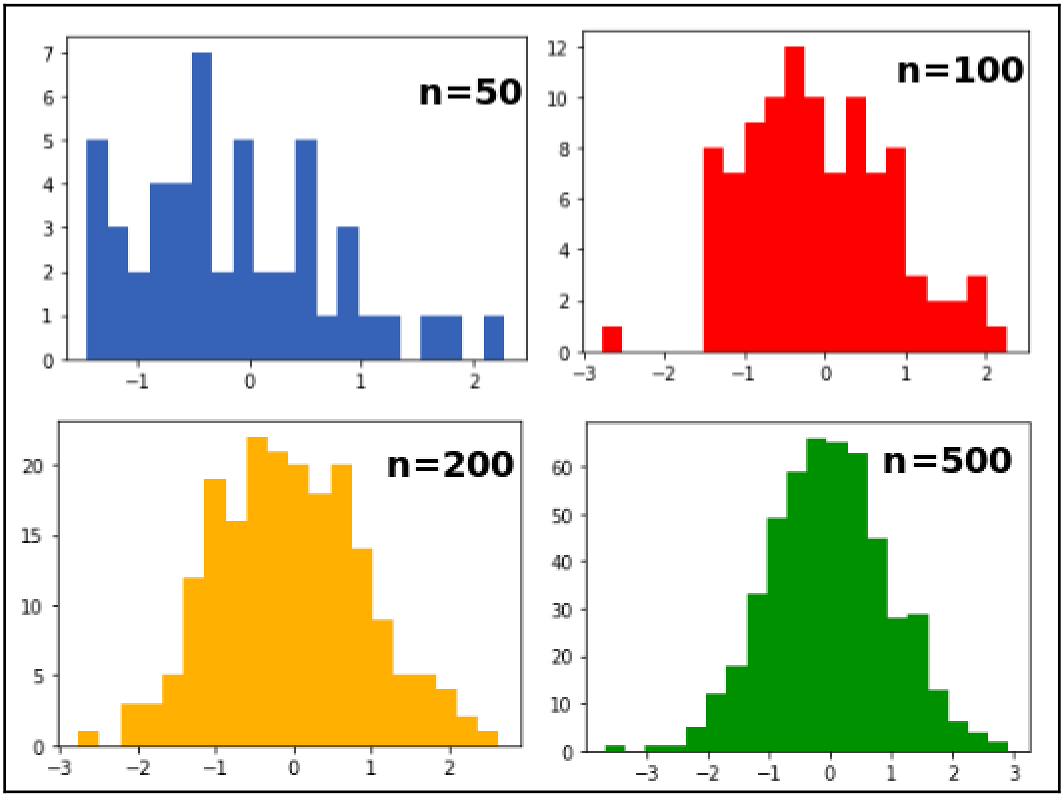
Kendall's rank correlation coefficient or Kendall's tau coefficient is a non-parametric statistic used to measure the association between two ordinal variables. It is a type of rank correlation. It measures the similarity or dissimilarity between two variables. If both the variables are binary, then Pearson's = Spearman's = Kendall's tau.

Till now, we have seen descriptive statistics topics such as central measures, dispersion measures, distribution measures, and variable relationship measures. It's time to jump to the inferential statistics topics such as the central limit theorem, sampling techniques, and parametric and non-parametric tests.

Central limit theorem

Data analysis methods involve hypothesis testing and deciding confidence intervals. All statistical tests assume that the population is normally distributed. The central limit theorem is the core of hypothesis testing. According to this theorem, the sampling distribution approaches a normal distribution with an increase in the sample size. Also, the mean of the sample gets closer to the population means and the standard deviation of the sample gets reduced. This theorem is essential for working with inferential statistics, helping data analysts figure out how samples can be useful in getting insights about the population.

Does it provide answers to questions such as what size of sample should be taken or which sample size is an accurate representation of the population? You can understand this with the help of the following diagram:



In the preceding diagram, you can see four histograms for different-different sample sizes 50, 100, 200, and 500. If you observe here, as the sample size increases, the histogram approaches a normal curve. Let's learn sampling techniques in the next section.

Collecting samples

A sample is a small set of the population used for data analysis purposes. Sampling is a method or process of collecting sample data from various sources. It is the most crucial part of data collection. The success of an experiment depends upon how well the data is collected. If anything goes wrong with sampling, it will hugely affect the final interpretations. Also, it is impossible to collect data for the whole population. Sampling helps researchers to infer the population from the sample and reduces the survey cost and workload to collect and manage data. There are lots of sampling techniques available, for various purposes. These techniques can be categorized into two categories: probability sampling and non-probability sampling, described in more detail here:

- **Probability sampling:** With this technique, there is a random selection of every respondent of the population, with an equal chance of the selected sample. Such types of sampling techniques are more time-consuming and expensive, and include the following:
    - **Simple random sampling:** With this technique, each respondent is selected by chance, meaning that each respondent has an equal chance of being selected. It is a simple and straightforward method—for example, 20 products being randomly selected from 500 products for quality testing.
    - **Stratified sampling:** With this technique, the whole population is divided into small groups known as strata that are based on some similarity criteria. These strata can be of unequal size. This technique improves accuracy by reducing selection bias.
    - **Systematic sampling:** With this technique, respondents are selected at regular intervals. In other words, we can say respondents are selected in systematic order from the target population, such as every *n*th respondent from the population.
    - **Cluster sampling:** With this sampling technique, the entire population is divided into clusters or sections. Clusters are formed based on gender, location, occupation, and so on. These entire clusters are used for sampling rather than the individual respondent.

- **Non-probability sampling:** This sampling non-randomly selects every respondent of the population, with an unequal chance of the selected sample. Its outcome might be biased. Such types of sampling techniques are cheaper and more convenient, and include the following:
    - **Convenience sampling:** This is the easiest technique for data collection. It selects respondents based on their availability and willingness to participate. Statisticians prefer this technique for the initial survey due to cost and fast collection of data, but the results are more prone to bias.
    - **Purposive sampling:** This is also known as judgmental sampling because it depends upon the statistician's judgment. Statisticians decide at runtime who will participate in the survey based on certain predefined characteristics. News reporters use this technique to select people whose opinions they wish to obtain.
    - **Quota sampling:** This technique predefines the properties of strata and proportions for the sample. Sample respondents are selected until a definitive proportion is met. It differs from stratified sampling in terms of selection strategy; it selects items in strata using random sampling.
    - **Snowball sampling:** This technique is used in a situation where finding respondents in a population is rare and difficult to trace, in areas such as illegal immigration or HIV. Statisticians contact volunteers to reach out to the victims. It is also known as referral sampling because the initial person taking part in the survey refers to another person who fits the sample description.

In this section, we have seen sampling methods and their types: probability sampling and non-probability sampling. Now, it's time to jump to hypothesis testing techniques. In upcoming sections, we will focus on parametric and non-parametric hypothesis testing.

Performing parametric tests

The hypothesis is the main core topic of inferential statistics. In this section, we will focus on parametric tests. The basic assumption of a parametric test is the underlying statistical distribution. Most elementary statistical methods are parametric in nature. Parametric tests are used for quantitative and continuous data. Parameters are numeric quantities that represent the whole population. Parametric tests are more powerful and reliable than non-parametric tests. The hypothesis is developed on the parameters of the population distribution. Here are some examples of parametric tests:

- A t-test is a kind of parametric test that is used for checking if there is a significant difference between the means of the two groups concerned. It is the most commonly used inferential statistic that follows the normal distribution. A t-test has two types: a one-sample t-test and a two-sample t-test. A one-sample t-test is used for checking if there is a significant difference between a sample and hypothesized population means. Let's take 10 students and check whether their average weight is 68 kg or not by using a t-test, as follows:

```
import numpy as np

from scipy.stats import ttest_1samp

# Create data
data=np.array([63, 75, 84, 58, 52, 96, 63, 55, 76, 83])

# Find mean
mean_value = np.mean(data)

print("Mean:",mean_value)
```

Output:

```
Mean: 70.5
```

In the preceding code block, we have created an array of 10 students' weight and computed its arithmetic mean using `numpy.mean()`.

Let's perform a one-sample t-test, as follows:

```
# Perform one-sample t-test
t_test_value, p_value = ttest_1samp(data, 68)

print("P Value:",p_value)

print("t-test Value:",t_test_value)

# 0.05 or 5% is significance level or alpha.
if p_value < 0.05:

    print("Hypothesis Rejected")

else:
    print("Hypothesis Accepted")
```

Output:

```
P Value: 0.5986851106160134
```

```
t-test Value: 0.5454725779039431
Hypothesis Accepted
```

In the preceding code block, we have tested the null hypothesis (average weight of 10 students is 68 kg) by using `ttest_1samp()`. The output results have shown that the null hypothesis is accepted with a 95% confidence interval, which means that the average weight of 10 students is 68 kg.

- A two-sample t-test is used for comparing the significant difference between two independent groups. This test is also known as an independent samples t-test. Let's compare the average weight of two independent student groups, as follows:

  Null Hypothesis H_0: Sample means are equal—$\mu_1 = \mu_2$

  Alternative Hypothesis H_a: Sample means are not equal—$\mu_1 > \mu_2$ or $\mu_2 > \mu_1$

Have a look at the following code block:

```
from scipy.stats import ttest_ind

# Create numpy arrays
data1=np.array([63, 75, 84, 58, 52, 96, 63, 55, 76, 83])

data2=np.array([53, 43, 31, 113, 33, 57, 27, 23, 24, 43])
```

In the preceding code block, we have created two arrays of 10 students' weights.

Let's perform a two-sample t-test, as follows:

```
# Compare samples

stat, p = ttest_ind(data1, data2)

print("p-values:",p)

print("t-test:",stat)

# 0.05 or 5% is significance level or alpha.

if p < 0.05:

    print("Hypothesis Rejected")

else:

    print("Hypothesis Accepted")
```

```
Output:
p-values: 0.015170931362451255
t-test: 2.6835879913819185
Hypothesis Rejected
```

In the preceding code block, we have tested the hypothesis average weight of two groups using the `ttest_ind()` method, and results show that the null hypothesis is rejected with a 95% confidence interval, which means that the sample means are different.

- A paired sample t-test is a dependent sample t-test, which is used to decide whether the mean difference between two observations of the same group is zero—for example, to compare the difference in blood pressure level for a group of patients before and after some drug treatment. This is equivalent to a one-sample t-test and is also known as a dependent sample t-test. Let's perform a paired t-test to assess the impact of weight loss treatment. We have collected the weight of patients before and after treatment. This can be represented using the following hypothesis:

  ***Null Hypothesis H_0*:** Mean difference between the two dependent samples is 0.

  ***Alternative Hypothesis H_a*:** Mean difference between the two dependent samples is not 0.

Have a look at the following code block:

```
# paired test
from scipy.stats import ttest_rel

# Weights before treatment
data1=np.array([63, 75, 84, 58, 52, 96, 63, 65, 76, 83])

# Weights after treatment
data2=np.array([53, 43, 67, 59, 48, 57, 65, 58, 64, 72])
```

In the preceding code block, we have created two arrays of 10 patients' weights before and after treatment. Let's perform a paired sample t-test, as follows:

```
# Compare weights

stat, p = ttest_rel(data1, data2)

print("p-values:",p)

print("t-test:",stat)
```

```
# 0.05 or 5% is the significance level or alpha.

if p < 0.05:
    print("Hypothesis Rejected")

else:

    print("Hypothesis Accepted")
```

```
Output:
p-values: 0.013685575312467715
t-test: 3.0548295044306903
Hypothesis Rejected
```

In the preceding code block, we have tested the hypothesis of the average weight of two groups before and after treatment using the `ttest_rel()` method. Results show that the null hypothesis is rejected with a 95% confidence interval, which means that weight loss treatment has a significant impact on the patient's weight.

- ANOVA: A t-test only deals with two groups, but sometimes we have more than two groups or multiple groups at the same time to compare. **ANOVA** (**ANalysis Of VAriance**) is a statistical inference test used for comparing multiple groups. It analyzes the variance between and within multiple groups and tests several null hypotheses at the same time. It usually compares more than two sets of data and checks statistical significance. We can use ANOVA in three ways: one-way ANOVA, two-way ANOVA, and N-way multivariate ANOVA.
- With the one-way ANOVA method, we compare multiple groups based on only one independent variable—for example, an IT company wants to compare multiple employee groups' or teams' productivity based on performance score. In our example, we are comparing the performance of employees in an IT company based in three locations: Mumbai, Chicago, and London. Here, we will perform a one-way ANOVA test and check for a significant difference in performance. Let's define the null and alternative hypotheses, as follows:

  ***Null Hypothesis H_0*:** There is no difference between the mean performance score of multiple locations.

  ***Alternative Hypothesis H_a*:** There is a difference between the mean performance score of multiple locations.

  Have a look at the following code block:

```
from scipy.stats import f_oneway

# Performance scores of Mumbai location
```

```
mumbai=[0.14730927, 0.59168541, 0.85677052, 0.27315387,
0.78591207,0.52426114, 0.05007655, 0.64405363, 0.9825853 ,
0.62667439]

# Performance scores of Chicago location
chicago=[0.99140754, 0.76960782, 0.51370154, 0.85041028,
0.19485391,0.25269917, 0.19925735, 0.80048387, 0.98381235,
0.5864963 ]

# Performance scores of London location
london=[0.40382226, 0.51613408, 0.39374473, 0.0689976 ,
0.28035865,0.56326686, 0.66735357, 0.06786065, 0.21013306,
0.86503358]
```

In the preceding code block, we have created three lists of employee performance scores for three locations: Mumbai, Chicago, and London.

Let's perform a one-way ANOVA test, as follows:

```
# Compare results using Oneway ANOVA
stat, p = f_oneway(mumbai, chicago, london)

print("p-values:", p)

print("ANOVA:", stat)

if p < 0.05:
    print("Hypothesis Rejected")

else:

    print("Hypothesis Accepted")

Output:
p-values: 0.27667556390705783
ANOVA: 1.3480446381965452
Hypothesis Accepted
```

In the preceding code block, we have tested the hypothesis that there is no difference between the mean performance score of various locations using the `f_oneway()` method. The preceding results show that the null hypothesis is accepted with a 95% confidence interval, which means that there is no significant difference between the mean performance score of all the locations.

- With the two-way ANOVA method, we compare multiple groups based on two independent variables—for example, if an IT company wants to compare multiple employee groups' or teams' productivity based on working hours and project complexity.
- In N-way ANOVA, we compare multiple groups based on *N* independent variables—for example, if an IT company wants to compare multiple employee groups' or teams' productivity based on working hours, project complexity, employee training, and other employee perks and facilities.

In this section, we have explored parametric tests such as the t-test and ANOVA tests in detail. Let's jump to the non-parametric hypothesis test.

Performing non-parametric tests

A non-parametric test doesn't rely on any statistical distribution; that is why it is known as a "distribution-free" hypothesis test. Non-parametric tests don't have parameters of the population. Such types of tests are used for order and rank of observations and require special ranking and counting methods. Here are some examples of non-parametric tests:

- A **Chi-Square test** is determined by a significant difference or relationship between two categorical variables from a single population. In general, this test assesses whether distributions of categorical variables differ from each other. It is also known as a Chi-Square goodness of fit test or a Chi-Square test for independence. A small value of the Chi-Square statistic means observed data fit with expected data, and a larger value of the Chi-Square statistic means observed data doesn't fit with expected data. For example, the impact of gender on voting preference or the impact of company size on health insurance coverage can be assessed by a Chi-Square test:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

Here, *O* is the observed value, *E* is the expected value, and "*i*" is the "ith" position in the contingency table.

Let's understand the Chi-Square test using an example. Suppose we have done a survey in a company of 200 employees and asked about their highest qualification such as High School, Higher Secondary, Graduate, Post-Graduate, and compare it with performance levels such as Average and Outstanding. Here is the hypothesis and contingency criteria:

> ***Null Hypothesis H_0*:** The two categorical variables are independent—that is, employee performance is independent of the highest qualification level.
>
> ***Alternative Hypothesis H_a*:** The two categorical variables are not independent—that is, employee performance is not independent of the highest qualification level.

The contingency table can be represented as follows:

	High School	Higher Secondary	Graduate	Post-Graduate
Average	20	16	13	7
Outstanding	31	40	50	13

Let's perform a Chi-Square test and check for a significant difference in the association between variables, as follows:

```
from scipy.stats import chi2_contingency

# Average performing employees
average=[20, 16, 13, 7]

# Outstanding performing employees
outstanding=[31, 40, 60, 13]

# contingency table
contingency_table= [average, outstanding]
```

In the preceding code block, we have created two lists of average and outstanding performing employees and created a contingency table.

Let's perform a Chi-Square test, as follows:

```
# Apply Test
stat, p, dof, expected = chi2_contingency(contingency_table)

print("p-values:",p)

if p < 0.05:
    print("Hypothesis Rejected")
```

```
else:

    print("Hypothesis Accepted")
```

Output:

```
p-values: 0.059155602774381234
Hypothesis Accepted
```

In the preceding code block, we have tested the hypothesis that employee performance is independent of the highest qualification level. The preceding results show that the null hypothesis is accepted with a 95% confidence interval, which means that employee performance is independent of the highest qualification level.

- The **Mann-Whitney U test** is the non-parametric counterpart of the t-test for two samples. It doesn't assume that the difference between the samples is normally distributed. The Mann-Whitney U test is used when the observation is ordinal and assumptions of the t-test were not met—for example, comparing two groups of movie test preferences from their given movie ratings. Let's compare two groups of movie ratings using the following criteria:

  Null Hypothesis H_0: There is no difference between the two sample distributions.

  Alternative Hypothesis H_a: There is a difference between the two sample distributions.

Have a look at the following code block:

```
from scipy.stats import mannwhitneyu

# Sample1

data1=[7,8,4,9,8]

# Sample2

data2=[3,4,2,1,1]
```

In the preceding code block, we have created two lists of data.

Let's perform a Mann-Whitney U test, as follows:

```
# Apply Test

stat, p = mannwhitneyu(data1, data2)
```

```
print("p-values:",p)

# 0.01 or 1% is significance level or alpha.

if p < 0.01:

    print("Hypothesis Rejected")

else:
    print("Hypothesis Accepted")

Output:
p-values: 0.007666581056801412
Hypothesis Rejected
```

In the preceding code block, we have tested the hypothesis that there is no difference between the distribution of two movie rating groups using the `mannwhitneyu()` method. The results show that the null hypothesis is rejected with a 99% confidence interval, which means that there is a significant difference between the two movie rating groups.

- The **Wilcoxon signed-rank test** compares two paired samples. It is a non-parametric counterpart version of the paired t-test. It tests the null hypothesis as to whether the two paired samples belong to the same distribution or not—for example, to compare the difference between two treatment observations for multiple groups. Let's compare the difference between two treatment observations using the following criteria:

  ***Null Hypothesis H_0*:** There is no difference between the dependent sample distributions.

  ***Alternative Hypothesis H_a*:** There is a difference between the dependent sample distributions.

  Have a look at the following code block:

```
from scipy.stats import wilcoxon

# Sample-1
data1 = [1, 3, 5, 7, 9]

# Sample-2 after treatement
data2 = [2, 4, 6, 8, 10]
```

In the preceding code block, we have created two lists of data.

Let's perform a Wilcoxon signed-rank test, as follows:

```
# Apply
stat, p = wilcoxon(data1, data2)

print("p-values:",p)

# 0.01 or 1% is significance level or alpha.

if p < 0.01:

    print("Hypothesis Rejected")

else:
    print("Hypothesis Accepted")
```

```
Output:
p-values: 0.025347318677468252
Hypothesis Accepted
```

In the preceding code block, we have tested the hypothesis that there is no difference between the distribution of groups before and after treatment using the `wilcoxon()` method. The preceding results show that the null hypothesis is accepted with a 99% confidence interval, which means that there is no significant difference between the groups before and after treatment.

- The **Kruskal-Wallis** test is the non-parametric version of one-way ANOVA, to assess whether samples belong to the same distribution or not. It compares two or more independent samples. It extends the limit of the Mann-Whitney U test, which compares only two groups. Let's compare three sample groups using the following code:

```
from scipy.stats import kruskal

# Data sample-1
x = [38, 18, 39, 83, 15, 38, 63, 1, 34, 50]

# Data sample-2
y = [78, 32, 58, 59, 74, 77, 29, 77, 54, 59]

# Data sample-3
z = [117, 92, 42, 79, 58, 117, 46, 114, 86, 26]
```

In the preceding code block, we have created three lists of data. Let's perform a Kruskal-Wallis test, as follows:

```
# Apply kruskal-wallis test
stat, p = kruskal(x,y,z)

print("p-values:",p)

# 0.01 or 1% is significance level or alpha.

if p < 0.01:

    print("Hypothesis Rejected")

else:
    print("Hypothesis Accepted")

Output:
p-values: 0.01997922369138151
Hypothesis Accepted
```

In the preceding code block, we have tested the hypothesis that there is no difference between the three sample groups using the `kruskal()` method. The preceding results show that the null hypothesis is accepted with a 99% confidence interval, which means that there is no difference between the three sample groups. Let's compare both parametric and non-parametric tests, as follows:

Features	Parametric Tests	Non-Parametric Tests
Test Statistic	Distribution	Arbitrary or "Distribution-Free"
Attribute Type	Numeric	Nominal and Ordinal
Central Tendency Measures	Mean	Median
Correlation Tests	Pearson's Correlation	Spearman's Correlation
Information about Population	Complete Information	No Information

In the preceding table, you have seen examples of parametric and non-parametric tests based on various features such as test statistic, attribute type, central tendency measures, correlation tests, and population information. Finally, you made it to the end. In this chapter, we have explored the fundamentals of descriptive as well as inferential statistics with Python.

Summary

The core fundamentals of statistics will provide the foundation for data analysis, facilitating how data is described and understood. In this chapter, you have learned the basics of statistics such as attributes and their different types such as nominal, ordinal, and numeric. You have also learned about mean, median, and mode for measuring central tendency. Range, IQR, variance, and standard deviation measures are used to estimate variability in the data; skewness and kurtosis are used for understanding data distribution; covariance and correlation are used to understand the relationship between variables. You have also seen inferential statistics topics such as the central limit theorem, collecting samples, and parametric and non-parametric tests. You have also performed hands-on coding on statistics concepts using the `pandas` and `scipy.stats` libraries.

The next chapter, `Chapter 4`, *Linear Algebra*, will help us to learn how to solve the linear system of equations, find Eigenvalues and Eigenvectors, and learn about binomial and normal distribution, normality tests, and masked arrays using the Python packages NumPy and SciPy.

4
Linear Algebra

Both linear algebra and statistics are the foundation for any kind of data analysis activity. Statistics help us to get an initial descriptive understanding and make inferences from data. In the previous chapter, we have understood descriptive and inferential statistical measures for data analysis. On the other side, linear algebra is one of the fundamental mathematical subjects that is the core foundation for any data professional. Linear algebra is useful for working with vectors and matrices. Most of the data is available in the form of either a vector or a matrix. In-depth knowledge of linear algebra helps data analysts and data scientists understand the workflow of machine learning and deep learning algorithms, giving them the flexibility to design and modify the algorithms as per your business needs. For example, if you want to work with **principal component analysis** (**PCA**) you must know the basics of Eigenvalues and Eigenvectors, or if you want to develop a recommender system you must know **singular value decomposition** (**SVD**). A solid background in mathematics and statistics will facilitate a smoother transition into the world of data analytics.

This chapter mainly focuses on the core concepts of linear algebra, such as polynomials, determinant, matrix inverse; solving linear equations; eigenvalues and eigenvectors; SVD; random numbers; binomial and normal distributions; normality tests; and masked arrays. We can also perform these operations in Python using the NumPy and SciPy packages. NumPy and SciPy both offer the `linalg` package for linear algebra operations.

In this chapter, we will cover the following topics:

- Fitting to polynomials with NumPy
- Determinant
- Finding the rank of a matrix
- Matrix inverse using NumPy
- Solving linear equations using NumPy
- Decomposing a matrix using SVD
- Eigenvectors and Eigenvalues using NumPy

- Generating random numbers
- Binomial distribution
- Normal distribution
- Testing normality of data using SciPy
- Creating a masked array using the `numpy.ma` subpackage

Technical requirements

For this chapter, the following technical information is available:

- You can find the code and the dataset at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter04`.
- All the code blocks are available in `ch4.ipynb`.
- In this chapter, we will use the NumPy, SciPy, Matplotlib, and Seaborn Python libraries.

Fitting to polynomials with NumPy

Polynomials are mathematical expressions with non-negative strategies. Examples of polynomial functions are linear, quadratic, cubic, and quartic functions. NumPy offers the `polyfit()` function to generate polynomials using least squares. This function takes *x*-coordinate, *y*-coordinate, and degree as parameters, and returns a list of polynomial coefficients.

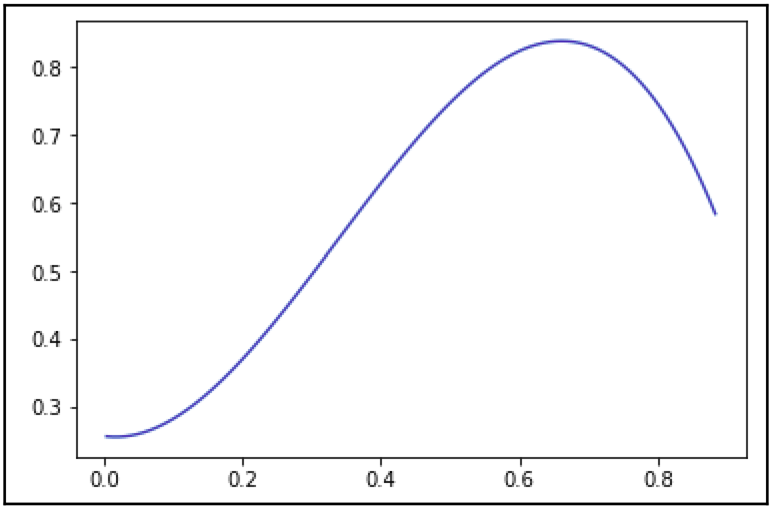
NumPy also offers `polyval()` to evaluate the polynomial at given values. This function takes coefficients of polynomials and arrays of points and returns resultant values of polynomials. Another function is `linspace()`, which generates a sequence of equally separated values. It takes the start, stop, and the number of values between the start-stop range and returns equally separated values in the closed interval.

Let's see an example to generate and evaluate polynomials using NumPy, as follows:

```
# Import required libraries NumPy, polynomial and matplotlib
import numpy as np
import matplotlib.pyplot as plt

# Generate two random vectors
v1=np.random.rand(10)
```

```
v2=np.random.rand(10)

# Creates a sequence of equally separated values
sequence = np.linspace(v1.min(),v1.max(), num=len(v1)*10)

# Fit the data to polynomial fit data with 4 degrees of the polynomial
coefs = np.polyfit(v1, v2, 3)

# Evaluate polynomial on given sequence
polynomial_sequence = np.polyval(coefs,sequence)

# plot the polynomial curve
plt.plot(sequence, polynomial_sequence)

# Show plot
plt.show()
```

This results in the following output:



The graph shown in the preceding screenshot will change in each iteration using the program written previously. The reason for this fluctuation is the random value generation of vectors.

Let's jump on to the next topic: *Determinant*. We will perform most of the linear algebra operations using the `numpy.linalg` subpackage. NumPy offers the `linalg` subpackage for linear algebra. We can use linear algebra for matrix operations such as inverse, rank, eigenvalues, eigenvectors, solving linear equations, and performing linear regression.

Determinant

The determinant is the most essential concept of linear algebra. It is a scalar value that is calculated from a square matrix. The determinant is a fundamental operation that helps us in the inverse matrix and in solving linear equations. Determinants are only calculated for square matrices. A square matrix has an equal number of rows and columns. The `numpy.linalg` subpackage provides the `det()` function for calculating the determinant of a given input matrix. Let's compute the determinant in the following code block:

```
# Import numpy
import numpy as np

# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Matrix:\n",mat)

# Calculate determinant
print("Determinant:",np.linalg.det(mat))
```

This results in the following output:

```
Matrix:
[[2 4]
[5 7]]
Determinant: -5.999999999999998
```

In the preceding code block, we have calculated the determinant of a given matrix using the `np.linalg.det()` method. Let's understand one more concept of linear algebra, which is rank, and compute it using the `numpy.linalg` subpackage.

Finding the rank of a matrix

Rank is a very important concept when it comes to solving linear equations. The rank of a matrix represents the amount of information that is kept in the matrix. A lower rank means less information, and a higher rank means a high amount of information. Rank can be defined as the number of independent rows or columns of a matrix. The `numpy.linalg` subpackage provides the `matrix_rank()` function. The `matrix_rank()` function takes the matrix as input and returns the computed rank of the matrix. Let's see an example of the `matrix_rank()` function in the following code block:

```
# import required libraries
import numpy as np
from numpy.linalg import matrix_rank
```

```
# Create a matrix
mat=np.array([[5, 3, 1],[5, 3, 1],[1, 0, 5]])

# Compute rank of matrix
print("Matrix: \n", mat)
print("Rank:",matrix_rank(mat))
```

This results in the following output:

```
Matrix:
[[5 3 1]
[5 3 1]
[1 0 5]]
Rank: 2
```

In the preceding code block, the `matrix_rank()` function of `numpy.linalg` is used to generate the rank of the matrix. Let's see another important concept of linear algebra: matrix inverse.

Matrix inverse using NumPy

A matrix is a rectangular sequence of numbers, expressions, and symbols organized in rows and columns. The multiplication of a square matrix and its inverse is equal to the identity matrix I. We can write it using the following equation:

$$AA^{-1}= I$$

The `numpy.linalg` subpackage provides a function for an inverse operation: the `inv()` function. Let's invert a matrix using the `numpy.linalg` subpackage. First, we create a matrix using the `mat()` function and then find the inverse of the matrix using the `inv()` function, as illustrated in the following code block:

```
# Import numpy
import numpy as np

# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Input Matrix:\n",mat)

# Find matrix inverse
inverse = np.linalg.inv(mat)
print("Inverse:\n",inverse)
```

This results in the following output:

```
Input Matrix:
[[2 4]
[5 7]]
Inverse:
[[-1.16666667 0.66666667]
[ 0.83333333 -0.33333333]]
```

In the preceding code block, we have computed the inverse of a matrix using the `inv()` function of the `numpy.linalg` subpackage.

If the given input matrix is not a square matrix and a singular matrix, it will raise a `LinAlgError` error. If you want, you can test the `inv()` function manually. I will leave this as an activity for you.

Solving linear equations using NumPy

Matrix operations can transform one vector into another vector. These operations will help us to find the solution for linear equations. NumPy provides the `solve()` function to solve linear equations in the form of Ax=B. Here, A is the n*n matrix, B is a one-dimensional array and x is the unknown one-dimensional vector. We will also use the `dot()` function to compute the dot product of two floating-point number arrays.

Let's solve an example of linear equations, as follows:

1. Create matrix A and array B for a given equation, like this:

$$x1+x2 = 200$$
$$3x1+2x2 = 450$$

This is illustrated in the following code block

```
# Create matrix A and Vector B using NumPy
A=np.mat([[1,1],[3,2]])
print("Matrix A:\n",A)

B = np.array([200,450])
print("Vector B:", B)
```

This results in the following output:

```
Matrix A:
[[1 1]
[3 2]]
Vector B: [200 450]
```

In the preceding code block, we have created a 2*2 matrix and a vector.

2. Solve a linear equation using the `solve()` function, like this:

   ```
   # Solve linear equations
   solution = np.linalg.solve(A, B)
   print("Solution vector x:", solution)
   ```

   This results in the following output:

   ```
   Solution vector x: [ 50. 150.]
   ```

   In the preceding code block, we have solved a linear equation using the `solve()` function of the `numpy.linalg` subpackage.

3. Check the solution using the `dot()` function, like this:

   ```
   # Check the solution
   print("Result:",np.dot(A,solution))
   ```

   This results in the following output:

   ```
   Result: [[200. 450.]]
   ```

In the preceding code block, we have assessed the solution using the `dot()` function. You can see the dot product of A and the solution is equivalent to B. Till now, we have seen the determinant, rank, inverse, and how to solve linear equations. Let's jump to SVD for matrix decomposition.

Decomposing a matrix using SVD

Matrix decomposition is the process of splitting a matrix into parts. It is also known as matrix factorization. There are lots of matrix decomposition methods available such as **lower-upper** (**LU**) decomposition, **QR** decomposition (where **Q** is orthogonal and **R** is upper-triangular), Cholesky decomposition, and SVD.

Eigenanalysis decomposes a matrix into vectors and values. SVD decomposes a matrix into the following parts: singular vectors and singular values. SVD is widely used in signal processing, computer vision, **natural language processing** (**NLP**), and machine learning—for example, topic modeling and recommender systems where SVD is widely accepted and implemented in real-life business solutions. Have a look at the following:

$$A = U\Sigma V^T$$

Here, A is a m x n left singular matrix, Σ is a $n\ x\ n$ diagonal matrix, V is a $m\ x\ n$ right singular matrix, and V^T is the transpose of the V. The `numpy.linalg` subpackage offers the `svd()` function to decompose a matrix. Let's see an example of SVD, as follows:

```
# import required libraries
import numpy as np
from scipy.linalg import svd

# Create a matrix
mat=np.array([[5, 3, 1],[5, 3, 0],[1, 0, 5]])

# Perform matrix decomposition using SVD
U, Sigma, V_transpose = svd(mat)

print("Left Singular Matrix:",U)
print("Diagonal Matrix: ", Sigma)
print("Right Singular Matrix:", V_transpose)
```

This results in the following output:

```
Left Singular Matrix: [[-0.70097269 -0.06420281 -0.7102924 ]
                       [-0.6748668  -0.26235919  0.68972636]
                       [-0.23063411  0.9628321   0.14057828]]

Diagonal Matrix: [8.42757145 4.89599358 0.07270729]

Right Singular Matrix: [[-0.84363943 -0.48976369 -0.2200092]
                        [-0.13684207 -0.20009952 0.97017237]
                        [ 0.51917893 -0.84858218 -0.10179157]]
```

In the preceding code block, we have decomposed the given matrix into three parts: `Left Singular Matrix`, `Diagonal Matrix`, and `Right Singular Matrix` using the `svd()` function of the `scipy.linalg` subpackage.

Eigenvectors and Eigenvalues using NumPy

Eigenvectors and Eigenvalues are the tools required to understand linear mapping and transformation. Eigenvalues are solutions to the equation Ax = λx. Here, A is the square matrix, x is the eigenvector, and λ is eigenvalues. The `numpy.linalg` subpackage provides two functions, `eig()` and `eigvals()`. The `eig()` function returns a tuple of eigenvalues and eigenvectors, and `eigvals()` returns the eigenvalues.

Eigenvectors and eigenvalues are the core fundamentals of linear algebra. Eigenvectors and eigenvalues are used in SVD, spectral clustering, and PCA.

Let's compute the eigenvectors and eigenvalues of a matrix, as follows:

- Create the matrix using the NumPy `mat()` function, like this:

```
# Import numpy
import numpy as np

# Create matrix using NumPy
mat=np.mat([[2,4],[5,7]])
print("Matrix:\n",mat)
```

This results in the following output:

```
Matrix: [[2 4]
         [5 7]]
```

- Compute eigenvectors and eigenvalues using the `eig()` function, like this:

```
# Calculate the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(mat)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:", eigenvectors)
```

This results in the following output:

```
Eigenvalues: [-0.62347538 9.62347538]

Eigenvectors: [[-0.83619408 -0.46462222]
 [ 0.54843365 -0.885509 ]]
```

In the preceding two blocks, we have created a 2*2 matrix and computed eigenvectors and eigenvalues using the `eig()` function of the `numpy.linalg` subpackage.

- Compute eigenvalues using the `eigvals()` function, like this:

```
# Compute eigenvalues
eigenvalues= np.linalg.eigvals(mat)
print("Eigenvalues:", eigenvalues)
```

This results in the following output:

```
Eigenvalues: [-0.62347538 9.62347538]
```

In the preceding code snippet, we have computed the eigenvalues using the `eigvals()` function of the `numpy.linalg` subpackage. After performing eigendecomposition, we will see how to generate random numbers and a matrix.

Generating random numbers

Random numbers offer a variety of applications such as Monte Carlo simulation, cryptography, initializing passwords, and stochastic processes. It is not easy to generate real random numbers, so in reality, most applications use pseudo-random numbers. Pseudo numbers are adequate for most purposes except for some rare cases. Random numbers can be generated from discrete and continuous data. The `numpy.random()` function will generate a random number matrix for the given input size of the matrix.

The core random number generator is based on the Mersenne Twister algorithm (refer to `https://en.wikipedia.org/wiki/Mersenne_twister`).

Let's see one example of generating random numbers, as follows:

```
# Import numpy
import numpy as np

# Create an array with random values
random_mat=np.random.random((3,3))
print("Random Matrix: \n",random_mat)
```

This results in the following output:

```
Random Matrix: [[0.90613234 0.83146869 0.90874706]
```

```
                 [0.59459996 0.46961249 0.61380679]
                 [0.89453322 0.93890312 0.56903598]]
```

In the preceding example, we have generated a 3*3 random matrix using the `numpy.random.random()` function. Let's try other distributions for random number generation, such as binomial and normal distributions.

Binomial distribution

Binomial distribution models the number of repeated trials with the same probability on each trial. Here, each trial is independent and has two possible outcomes—success and failure—that can occur on each client. The following formula represents the binomial distribution:

$$P(X) = \frac{n!}{(n-X)!\,X!} \cdot (p)^X \cdot (q)^{n-X}$$

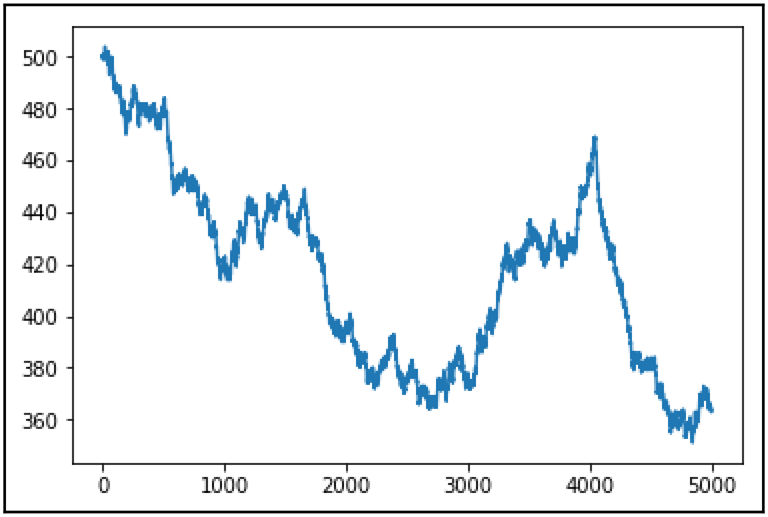
Here, p and q are the probabilities of success and failure, n is the number of trials, and X is the number of the desired output.

The `numpy.random` subpackage provides a `binomial()` function that generates samples based on the binomial distribution for certain parameters, number of trials, and the probability of success.

Let's consider a 17th-century gambling house where you can bet on eight tossing pieces and nine coins being flipped. If you get five or more heads then you win, otherwise you will lose. Let's write code for this simulation for 1,000 coins using the `binomial()` function, as follows:

```
# Import required libraries
import numpy as np
import matplotlib.pyplot as plt

# Create an numpy vector of size 5000 with value 0
cash_balance = np.zeros(5000)

cash_balance[0] = 500

# Generate random numbers using Binomial
samples = np.random.binomial(9, 0.5, size=len(cash_balance))

# Update the cash balance
for i in range(1, len(cash_balance)):
if samples[i] < 5:
    cash_balance[i] = cash_balance[i - 1] - 1
else:
    cash_balance[i] = cash_balance[i - 1] + 1

# Plot the updated cash balance
plt.plot(np.arange(len(cash_balance)), cash_balance)
plt.show()
```

This results in the following output:



In the preceding code block, we first created the `cash_balance` array of size 500 with zero values and updated the first value with 500. Then, we generated values between 0 to 9 using the `binomial()` function. After this, we updated the `cash_balance` array based on the results of coin tosses and plotted the cash balance using the Matplotlib library.

In each execution, the code will generate different results or random walks. If you want to make walking constant, you need to use the seed value in the `binomial()` function. Let's try another form of distribution for the random number generator: normal distribution.

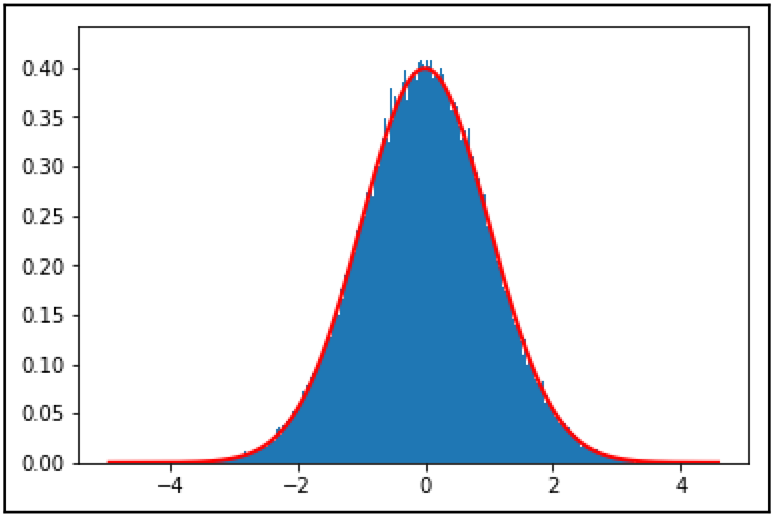
Normal distribution

Normal distributions occur frequently in real-life scenarios. A normal distribution is also known as a bell curve because of its characteristic shape. The probability density function models continuous distribution. The `numpy.random` subpackage offers lots of continuous distributions such as beta, gamma, logistic, exponential, multivariate normal, and normal distribution. The `normal()` functions find samples from Gaussian or normal distribution.

Let's write code for visualizing the normal distribution using the `normal()` function, as follows:

```
# Import required library
import numpy as np
import matplotlib.pyplot as plt

sample_size=225000

# Generate random values sample using normal distribution
sample = np.random.normal(size=sample_size)

# Create Histogram
n, bins, patch_list = plt.hist(sample, int(np.sqrt(sample_size)),
density=True)

# Set parameters
mu, sigma=0,1

x= bins
y= 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (bins - mu)**2 / (2 *
sigma**2) )

# Plot line plot(or bell curve)
plt.plot(x,y,color='red',lw=2)
plt.show()
```

This results in the following output:



Here, we have generated random values using the `normal()` function of the `numpy.random` subpackage and displayed the values using a histogram and line plot or bell curve or theoretical **probability density function** (**PDF**) with mean 0 and standard deviation of 1.

Testing normality of data using SciPy

A normal distribution is commonly used at a wide scale in scientific and statistical operations. As per the central limit theorem, as sample size increases, the sample distribution approaches a normal distribution. The normal distribution is well known and easy to use. In most cases, it is recommended to confirm the normality of data, especially in parametric methods, assuming that the data is Gaussian-distributed. There are lots of normality tests that exist in the literature such as the Shapiro-Wilk test, the Anderson-Darling test, and the D'Agostino-Pearson test. The `scipy.stats` package offers most of the tests for normality.

In this section, we will learn how to apply normality tests on data. We are using three samples of small-, medium-, and large-sized random data. Let's generate the data samples for all three samples using the `normal()` function, as follows:

```
# Import required library
import numpy as np

# create small, medium, and large samples for normality test
```

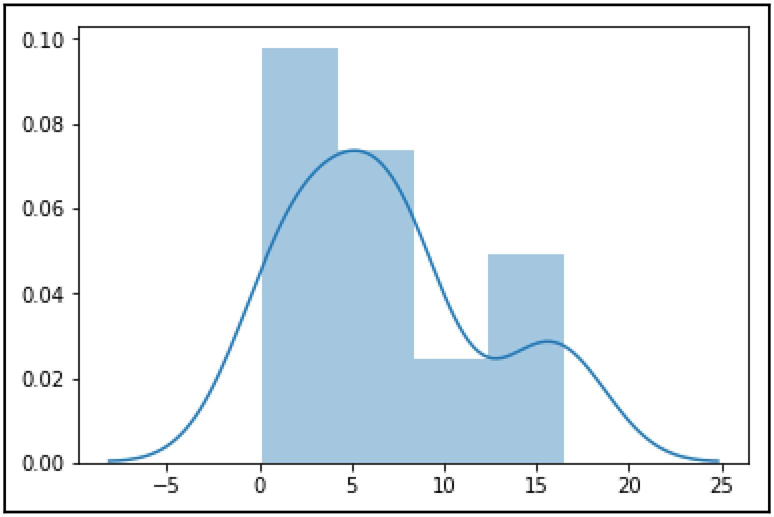
```
small_sample = np.random.normal(loc=100, scale=60, size=15)
medium_sample = np.random.normal(loc=100, scale=60, size=100)
large_sample = np.random.normal(loc=100, scale=60, size=1000)
```

We will now explore various techniques to check the normality of the data:

1. **Using a histogram:** A histogram is the easiest and fastest method to check the normality of the data. It divides the data into bins and counts the observation into each bin. Finally, it visualizes the data. Here, we are using `distplot()` from the `seaborn` library to plot the histogram and kernel density estimation. Let's see an example of a histogram for a small sample, as follows:

```
# Histogram for small
import seaborn as sns
import matplotlib.pyplot as plt

# Create distribution plot
sns.distplot(small_sample)

sns.distplot(small_sample)

plt.show()
```

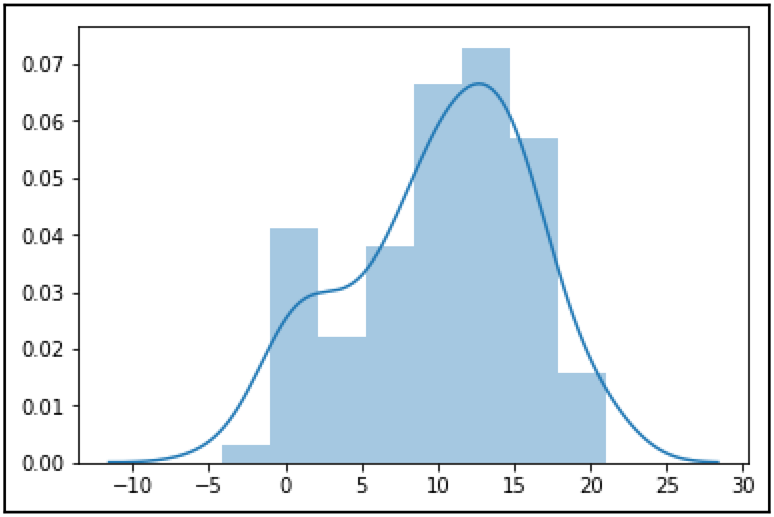
This results in the following output:



Let's see an example of the histogram for a medium sample, as follows:

```
# Histogram for medium
import seaborn as sns
import matplotlib.pyplot as plt

# Create distribution plot
sns.distplot(medium_sample)

plt.show()
```

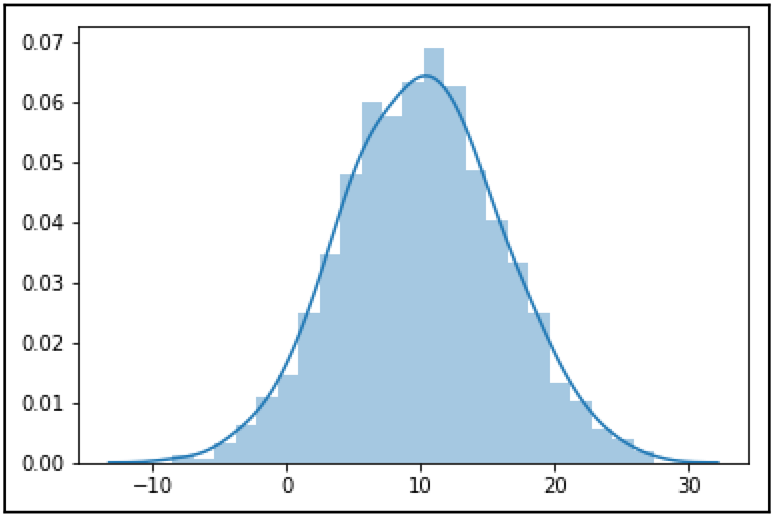
This results in the following output:



Let's see an example of the histogram for a large sample, as follows:

```
# Histogram for large
import seaborn as sns
import matplotlib.pyplot as plt

# Create distribution plot
sns.distplot(large_sample)

plt.show()
```

This results in the following output:



In the preceding three plots, we can observe that as the sample size increases, the curve becomes a normal curve. Histograms can be a good tool to test the normality of data.

2. **Shapiro-Wilk test:** This test is used to assess the normality of data. In Python, the `shapiro()` function of the `scipy.stats` subpackage can be used to assess normality. The `shapiro()` function will return tuples of two values: test statistics and p-value. Let's see the following example:

```
# Import shapiro function
from scipy.stats import shapiro

# Apply Shapiro-Wilk Test
print("Shapiro-Wilk Test for Small Sample: ",shapiro(small_sample))
print("Shapiro-Wilk Test for Medium Sample:
",shapiro(medium_sample))
print("Shapiro-Wilk Test for Large Sample: ",shapiro(large_sample))
```

This results in the following output:

```
Shapiro-Wilk Test for Small Sample: (0.9081739783287048,
0.2686822712421417)
Shapiro-Wilk Test for Medium Sample: (0.9661878347396851,
0.011379175819456577)
Shapiro-Wilk Test for Large Sample: (0.9991633892059326,
0.9433153867721558)
```

In the preceding code block, you can see that the small and large datasets have p-values greater than 0.05, so as the null hypothesis has failed to reject it, this means that the sample looks like a Gaussian or normal distribution; while for the medium dataset, the p-value is less than 0.05, so the null hypothesis has rejected it, which means the sample does not look like a Gaussian or normal distribution.

Similarly, we can try the Anderson-Darling test and the D'Agostino-Pearson test for normality using the `anderson()` and `normaltest()` functions of the `scipy.stats` subpackage. I will leave this for you as an activity. In visualization, we can also try the box plot and **quantile-quantile** (**QQ**) plot techniques to assess the normality of data. We will learn the box plot technique in the upcoming chapter, `Chapter 5`, *Data Visualization*. Let's move on to the concept of a masked array.

Creating a masked array using the numpy.ma subpackage

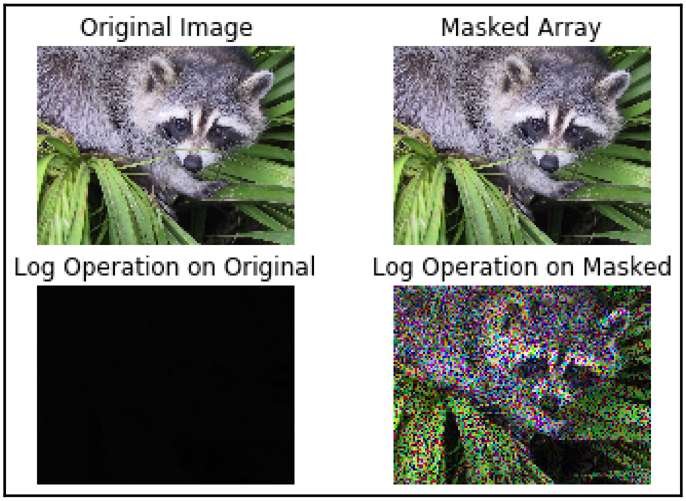
In most situations, real-life data is noisy and messy. It contains lots of gaps or missing characters in the data. Masked arrays are helpful in such cases and handle the issue. Masked arrays may contain invalid and missing values. The `numpy.ma` subpackage offers all the masked array-required functionality. In this section of the chapter, we will use the face image as the original image source and perform log mask operations.

Have a look at the following code block:

```
# Import required library
import numpy as np
from scipy.misc import face
import matplotlib.pyplot as plt

face_image = face()
mask_random_array = np.random.randint(0, 3, size=face_image.shape)

fig, ax = plt.subplots(nrows=2, ncols=2)
```

```
# Display the Original Image
plt.subplot(2,2,1)
plt.imshow(face_image)
plt.title("Original Image")
plt.axis('off')

# Display masked array
masked_array = np.ma.array(face_image, mask=mask_random_array)
plt.subplot(2,2,2)
plt.title("Masked Array")
plt.imshow(masked_array)
plt.axis('off')

# Log operation on original image
plt.subplot(2,2,3)
plt.title("Log Operation on Original")
plt.imshow(np.ma.log(face_image).astype('uint8'))
plt.axis('off')

# Log operation on masked array
plt.subplot(2,2,4)
plt.title("Log Operation on Masked")
plt.imshow(np.ma.log(masked_array).astype('uint8'))
plt.axis('off')

# Display the subplots
plt.show()
```

This results in the following output:



In the preceding code block, we first loaded the face image from the `scipy.misc` subpackage and created a random mask using the `randint()` function. Then, we applied the random mask on the face image. After this, we applied the log operation on the original face image and masked face image. Finally, we displayed all the images in 2*2 subplots. You can also try a range of mask operations on the image from the `numpy.ma` subpackage. Here, we are only focusing on the log operation of the masked array. That is all about basic linear algebra concepts. It's time to move on to data visualization concepts, in the next chapter.

Summary

Finally, we can conclude that mathematical subjects such as linear algebra are the backbone for all machine learning algorithms. Throughout the chapter, we have focused on essential linear algebra concepts to improve you as a data professional. In this chapter, you learned a lot about linear algebra concepts using the NumPy and SciPy subpackages. Our main focus was on polynomials, determinant, matrix inverse; solving linear equations; eigenvalues and eigenvectors; SVD; random numbers; binomial and normal distributions; normality tests; and masked arrays.

The next chapter, `Chapter 5`, *Data Visualization*, is about the important topic of visualizing data with Python. Visualization is something we often do when we start analyzing data. It helps to display relations between variables in the data. By visualizing the data, we can also get an idea about its statistical properties.

2
Section 2: Exploratory Data Analysis and Data Cleaning

The main objective of this section is to develop **Exploratory Data Analysis** (**EDA**) and data cleaning skills for the learner. These skills comprise data visualization, data extraction, and preprocessing. This section will mostly be using matplotlib, seaborn, Bokeh, pandas, scikit-learn, NumPy, and SciPy. It also focuses on signal processing and time series analysis.

This section includes the following chapters:

- `Chapter 5`, *Data Visualization*
- `Chapter 6`, *Retrieving, Processing, and Storing Data*
- `Chapter 7`, *Cleaning Messy Data*
- `Chapter 8`, *Signal Processing and Time Series*

5
Data Visualization

Data visualization is the initial move in the data analysis system toward easily understanding and communicating information. It represents information and data in graphical form using visual elements such as charts, graphs, plots, and maps. It helps analysts to understand patterns, trends, outliers, distributions, and relationships. Data visualization is an efficient way to deal with a large number of datasets.

Python offers various libraries for data visualization, such as Matplotlib, Seaborn, and Bokeh. In this chapter, we will first focus on Matplotlib, which is the basic Python library for visualization. After Matplotlib, we will explore Seaborn, which uses Matplotlib and offers high-level and advanced statistical plots. In the end, we will work on interactive data visualization using Bokeh. We will also explore `pandas` plotting. The following is a list of topics that will be covered in this chapter:

- Visualization using Matplotlib
- Advanced visualization using the Seaborn package
- Interactive visualization with Bokeh

Technical requirements

This chapter has the following technical requirements:

- You can find the code and the datasets at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter05`.
- All the code blocks are available in the `ch5.ipynb` file.
- This chapter uses only one CSV file (`HR_comma_sep.csv`) for practice purposes.
- In this chapter, we will use the Matplotlib, `pandas`, Seaborn, and Bokeh Python libraries.

Visualization using Matplotlib

As we know, a picture speaks a thousand words. Humans understand visual things better. Visualization helps to present things to any kind of audience and can easily explain a complex phenomenon in layman's terms. Python offers a couple of visualization libraries, such as Matplotlib, Seaborn, and Bokeh.

Matplotlib is the most popular Python module for data visualization. It is a base library for most of the advanced Python visualization modules, such as Seaborn. It offers flexible and easy-to-use built-in functions for creating figures and graphs.

In Anaconda, Matplotlib is already installed. If you still find an error, you can install it in the following ways.

We can install Matplotlib with `pip` as follows:

```
pip install matplotlib
```

For Python 3, we can use the following command:

```
pip3 install matplotlib
```

You can also simply install Matplotlib from your terminal or Command Prompt using the following command:

```
conda install matplotlib
```

To create a very basic plot in Matplotlib, we need to invoke the `plot()` function in the `matplotlib.pyplot` subpackage. This function produces a two-dimensional plot for a single list or multiple lists of points with known *x* and *y* coordinates.

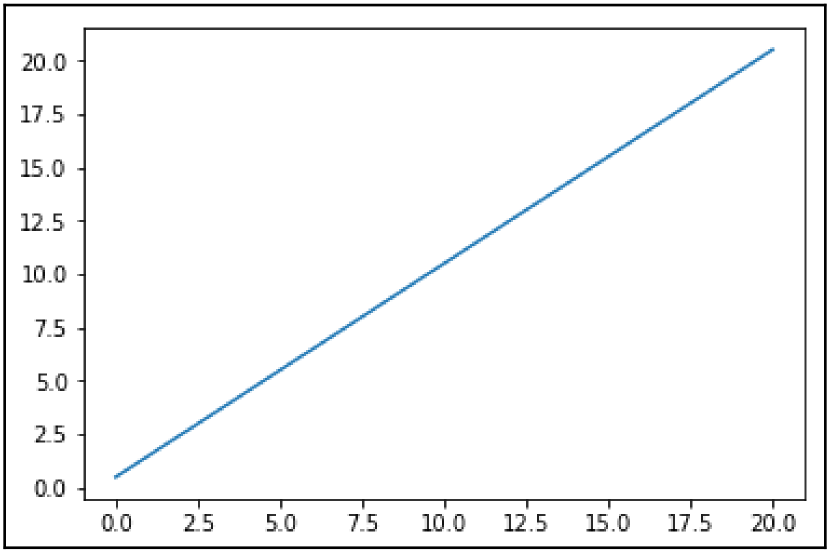
The following demo code is in the `ch5.ipynb` file in this book's code bundle, which you can find at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/blob/master/Chapter05/Ch5.ipynb`.

Let's see a small demo code for visualizing the line plot:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt
import numpy as np

# create the data
a = np.linspace(0, 20)

# Draw the plot
plt.plot(a, a + 0.5, label='linear')
```

```
# Display the chart
plt.show()
```

This results in the following output:



In the preceding code block, first, we are importing the Matplotlib and NumPy modules. After this, we are creating the data using the `linespace()` function of NumPy and plotting this data using the `plot()` function of Matplotlib. Finally, we are displaying the figure using the `show()` function.

There are two basic components of a plot: the figure and the axes. The figure is a container on which everything is drawn. It contains components such as plots, subplots, axes, titles, and a legend. In the next section, we will focus on these components, which act like accessories for charts.

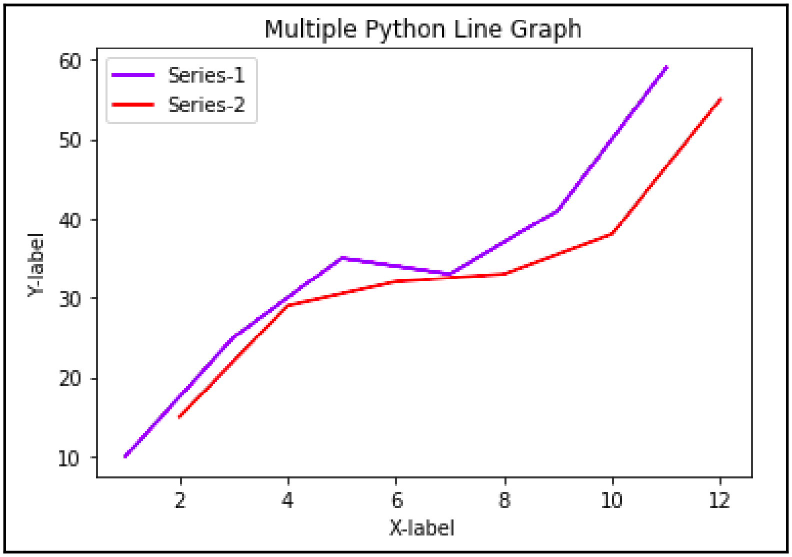
Accessories for charts

In the `matplotlib` module, we can add titles and axes labels to a graph. We can add a title using `plt.title()` and labels using `plt.xlabel()` and `plt.ylabel()`.

Multiple graphs mean multiple objects, such as line, bar, and scatter. Points of different series can be shown on a single graph. Legends or graph series reflect the *y* axis. A legend is a box that appears on either the right or left side of a graph and shows what each element of the graph represents. Let's see an example where we see how to use these accessories in our charts:

```
# Add the required libraries
import matplotlib.pyplot as plt

# Create the data
x = [1,3,5,7,9,11]
y = [10,25,35,33,41,59]

# Let's plot the data
plt.plot(x, y,label='Series-1', color='blue')

# Create the data
x = [2,4,6,8,10,12]
y = [15,29,32,33,38,55]

# Plot the data
plt.plot(x, y, label='Series-2', color='red')

# Add X Label on X-axis
plt.xlabel("X-label")

# Add X Label on X-axis
plt.ylabel("Y-label")

# Append the title to graph
plt.title("Multiple Python Line Graph")

# Add legend to graph
plt.legend()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding graph, two lines are shown on a single graph. We have used two extra parameters – `label` and `color` – in the `plot()` function. The `label` parameter defines the name of the series and `color` defines the color of the line graph. In the upcoming sections, we will focus on different types of plots. We will explore a scatter plot in the next section.

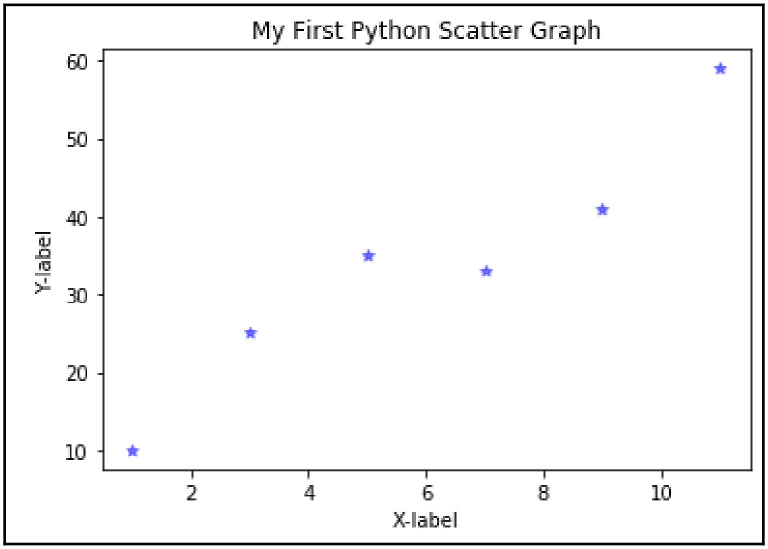
Scatter plot

Scatter plots draw data points using Cartesian coordinates to show the values of numerical values. They also represent the relationship between two numerial values. We can create a scatter plot in Matplotlib using the `scatter()` function, as follows:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data
x = [1,3,5,7,9,11]
y = [10,25,35,33,41,59]

# Draw the scatter chart
plt.scatter(x, y, c='blue', marker='*',alpha=0.5)

# Append the label on X-axis
plt.xlabel("X-label")
```

```
# Append the label on X-axis
plt.ylabel("Y-label")

# Add the title to graph
plt.title("Scatter Chart Sample")

# Display the chart
plt.show()
```

This results in the following output:



In the preceding scatter plot, the `scatter()` function takes x-axis and y-axis values. In our example, we are plotting two lists: `x` and `y`. We can also use optional parameters such as `c` for color, `alpha` for the transparency of the markers, ranging between 0 and 1, and `marker` for the shape of the points in the scatter plot, such as `*`, `o`, or any other symbol. In the next section, we will focus on the line plot.

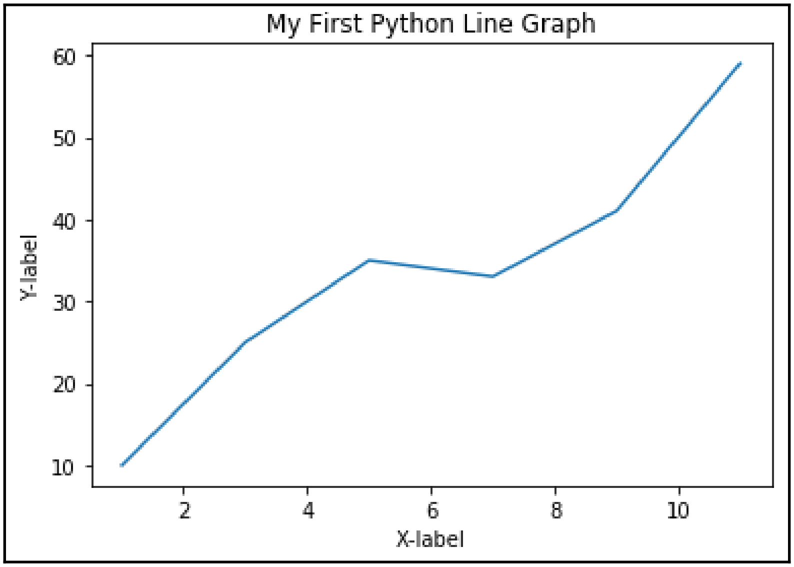
Line plot

A line plot is a chart that displays a line between two variables. It has a sequence of data points joined by a segment:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt
```

```
# create the data
x = [1,3,5,7,9,11]
y = [10,25,35,33,41,59]

# Draw the line chart
plt.plot(x, y)

# Append the label on X-axis
plt.xlabel("X-label")

# Append the label on X-axis
plt.ylabel("Y-label")

# Append the title to chart
plt.title("Line Chart Sample")

# Display the chart
plt.show()
```

This results in the following output:



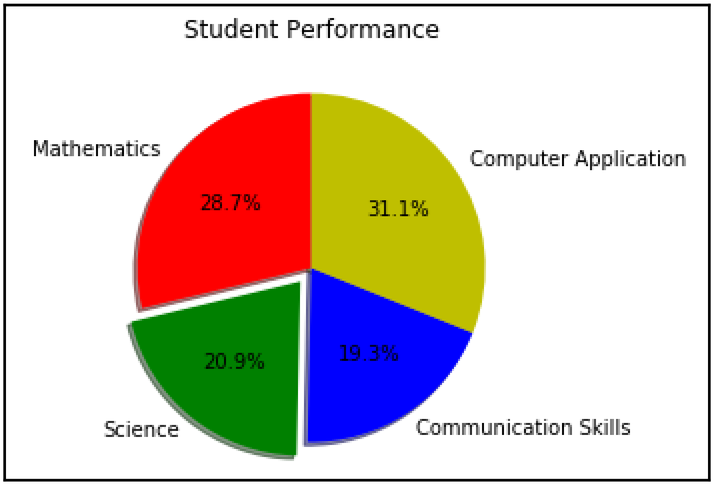
In the preceding line plot program, the `plot()` function takes x-axis and y-axis values. In the next section, we will learn how to plot a pie chart.

Pie plot

A pie plot is a circular graph that is split up into wedge-shaped pieces. Each piece is proportionate to the value it represents. The total value of the pie is 100 percent:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data
subjects = ["Mathematics","Science","Communication Skills","Computer
Application"]
scores = [85,62,57,92]

# Plot the pie plot
plt.pie(scores,
        labels=subjects,
        colors=['r','g','b','y'],
        startangle=90,
        shadow= True,
        explode=(0,0.1,0,0),
        autopct='%1.1f%%')

# Add title to graph
plt.title("Student Performance")

# Draw the chart
plt.show()
```

This results in the following output:



In the preceding code of the pie chart, we specified values, labels, colors, startangle, shadow, explode, and autopct. In our example, values is the scores of the student in four subjects and labels is the list of subject names. We can also specify the color list for the individual subject scores. The startangle parameter specifies the first value angle, which is 90 degrees; this means the first line is vertical.

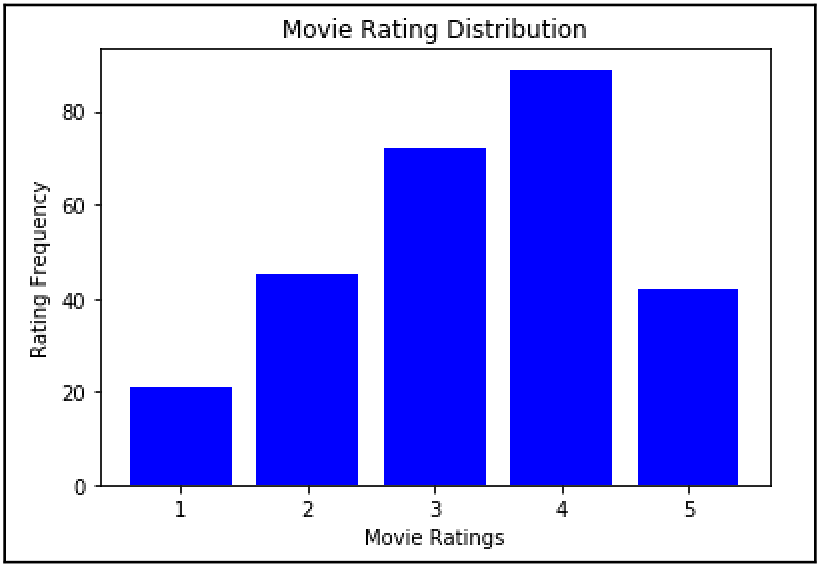
Optionally, we can also use the shadow parameter to specify the shadow of the pie slice and the explode parameter to pull out a pie slice list of the binary value. If we want to pull out a second pie slice, then a tuple of values would be (0, 0.1, 0, 0). Let's now jump to the bar plot.

Bar plot

A bar plot is a visual tool to compare the values of various groups. It can be drawn horizontally or vertically. We can create a bar graph using the bar() function:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data
movie_ratings = [1,2,3,4,5]
rating_counts = [21,45,72,89,42]

# Plot the data
plt.bar(movie_ratings, rating_counts, color='blue')

# Add X Label on X-axis
plt.xlabel("Movie Ratings")

# Add X Label on X-axis
plt.ylabel("Rating Frequency")

# Add a title to graph
plt.title("Movie Rating Distribution")

# Show the plot
plt.show()
```

This results in the following output:



In the preceding bar chart program, the `bar()` function takes *x*-axis values, *y*-axis values, and a color. In our example, we are plotting movie ratings and their frequency. Movie ratings are on the *x* axis and the rating frequency is on the *y* axis. We can also specify the color of the bars in the bar graph using the `color` parameter. Let's see another variant of bar plot in the next subsection.

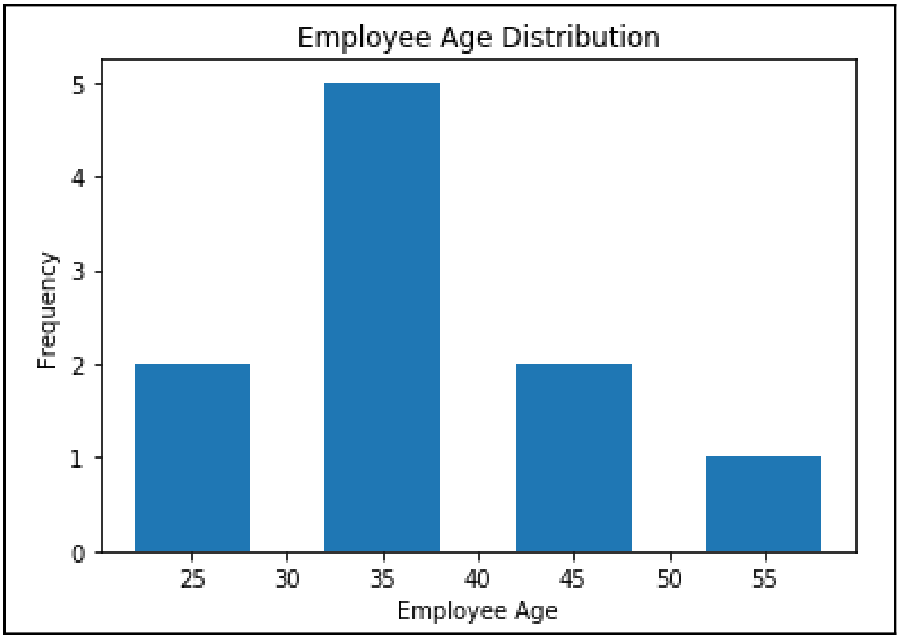
Histogram plot

A histogram shows the distribution of a numeric variable. We create a histogram using the `hist()` method. It shows the probability distribution of a continuous variable. A histogram only works on a single variable while a bar graph works on two variables:

```
# Add the essential library
import matplotlib.pyplot as plt

# Create the data
employee_age = [21,28,32,34,35,35,37,42,47,55]

# Create bins for histogram
bins = [20,30,40,50,60]

# Plot the histogram
```

```
plt.hist(employee_age, bins, rwidth=0.6)

# Add X Label on X-axis
plt.xlabel("Employee Age")

# Add X Label on X-axis
plt.ylabel("Frequency")

# Add title to graph
plt.title("Employee Age Distribution")

# Show the plot
plt.show()
```

This results in the following output:



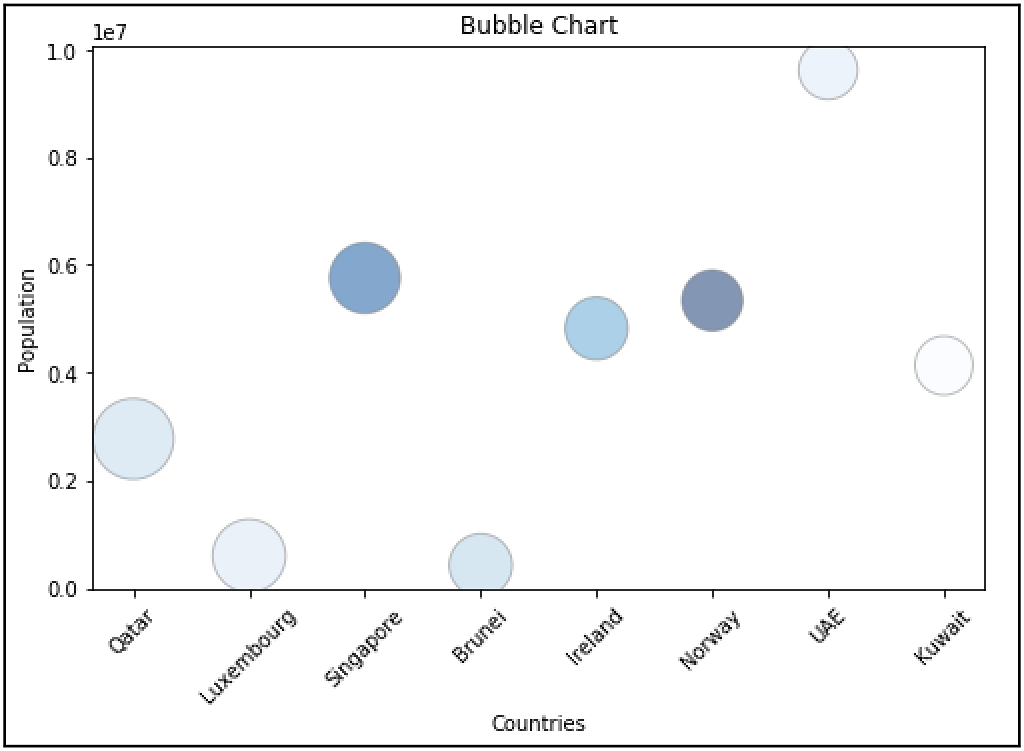
In the preceding histogram, the `hist()` function takes `values`, `bins`, and `rwidth`. In our example, we are plotting the age of the employee and using a bin of 10 years. We are starting our bin from 20 to 60 with a 10 years bin size. We are using a relative bar width of 0.6, but you can choose any size for thicker and thinner width. Now it's time to jump to the bubble plot, which can handle multiple variables in a two-dimensional plot.

Bubble plot

A bubble plot is a type of scatter plot. It not only draws data points using Cartesian coordinates but also creates bubbles on data points. Bubble shows the third dimension of a plot. It shows three numerical values: two values are on the *x* and *y* axes and the third one is the size of data points (or bubbles):

```
# Import the required modules
import matplotlib.pyplot as plt
import numpy as np

# Set figure size
plt.figure(figsize=(8,5))

# Create the data
countries =
['Qatar','Luxembourg','Singapore','Brunei','Ireland','Norway','UAE','Kuwait
']
populations = [2781682,
604245,5757499,428963,4818690,5337962,9630959,4137312]
gdp_per_capita = [130475, 106705, 100345, 79530, 78785, 74356,69382, 67000]

# scale GDP per capita income to shoot the bubbles in the graph
scaled_gdp_per_capita = np.divide(gdp_per_capita, 80)

colors = np.random.rand(8)

# Draw the scatter diagram
plt.scatter(countries, populations, s=scaled_gdp_per_capita, c=colors,
cmap="Blues",edgecolors="grey", alpha=0.5)

# Add X Label on X-axis
plt.xlabel("Countries")

# Add Y Label on X-axis
plt.ylabel("Population")

# Add title to graph
plt.title("Bubble Chart")

# rotate x label for clear visualization
plt.xticks(rotation=45)

# Show the plot
plt.show()
```

This results in the following output:



In the preceding plot, a bubble chart is created using the scatter function. Here, the important thing is the `s` (size) parameter of the scatter function. We assigned a third variable, `scaled_gdp_per_capita`, to the `size` parameters. In the preceding bubble plot, countries are on the *x* axis, the population is on the *y* axis, and GDP per capita is shown by the size of the scatter point or bubble. We also assigned a random color to the bubbles to make it attractive and more understandable. From the bubble size, you can easily see that Qatar has the highest GDP per capita and Kuwait has the lowest GDP per capita. In all the preceding sections, we have focused on most of the Matplotlib plots and charts. Now, we will see how we can plot the charts using the `pandas` module.

pandas plotting

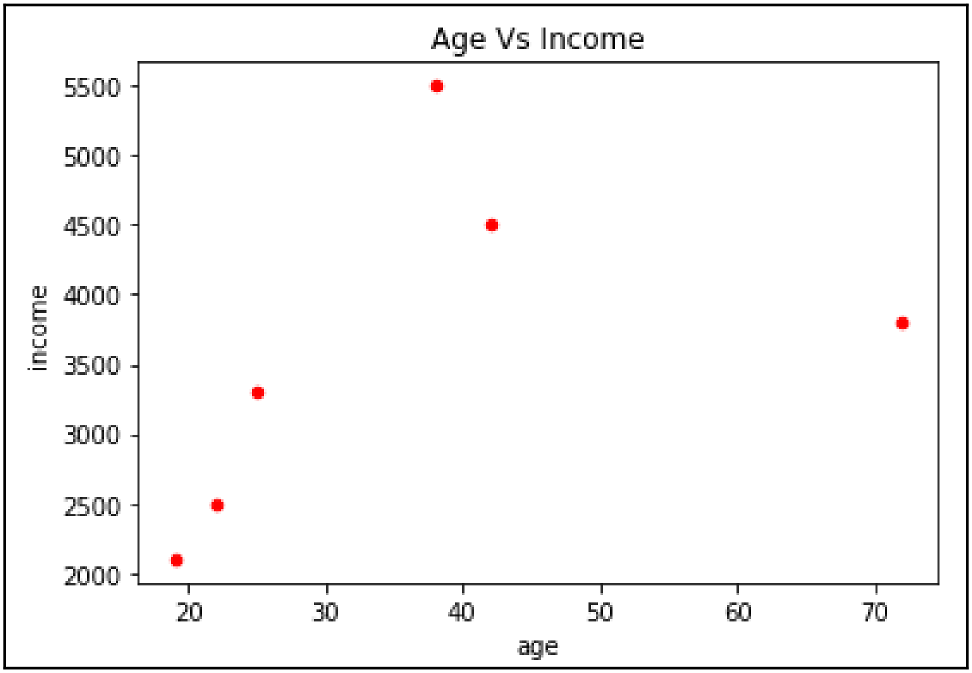
The pandas library offers the plot() method as a wrapper around the Matplotlib library. The plot() method allows us to create plots directly on pandas DataFrames. The following plot() method parameters are used to create the plots:

- kind: A string parameter for the type of graph, such as line, bar, barh, hist, box, KDE, pie, area, or scatter.
- figsize: This defines the size for a figure in a tuple of (width, height).
- title: This defines the title for the graph.
- grid: Boolean parameter for the axis grid line.
- legend: This defines the legend.
- xticks: This defines the sequence of x-axis ticks.
- yticks: This defines the sequence of y-axis ticks.

Let's create a scatter plot using the pandas plot() function:

```
# Import the required modules
import pandas as pd
import matplotlib.pyplot as plt

# Let's create a Dataframe
df = pd.DataFrame({
            'name':['Ajay','Malala','Abhijeet','Yming','Desilva','Lisa'],
            'age':[22,72,25,19,42,38],
            'gender':['M','F','M','M','M','F'],
'country':['India','Pakistan','Bangladesh','China','Srilanka','UK'],
            'income':[2500,3800,3300,2100,4500,5500]
        })

# Create a scatter plot
df.plot(kind='scatter', x='age', y='income', color='red', title='Age Vs
Income')

# Show figure
plt.show()
```

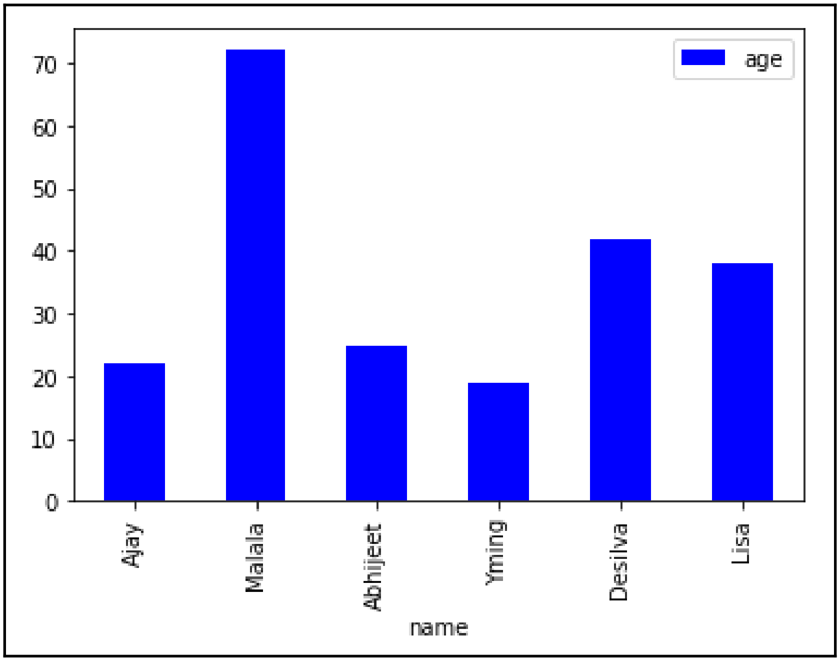
This results in the following output:



In the preceding plot, the `plot()` function takes `kind`, `x`, `y`, `color`, and `title` values. In our example, we are plotting the scatter plot between age and income using the `kind` parameter as `'scatter'`. The `age` and `income` columns are assigned to the `x` and `y` parameters. The scatter point color and the title of the plot are assigned to the `color` and `title` parameters:

```
import matplotlib.pyplot as plt
import pandas as pd

# Create bar plot
df.plot(kind='bar',x='name', y='age', color='blue')

# Show figure
plt.show()
```

This results in the following output:



In the preceding plot, the `plot()` function takes `kind`, `x`, `y`, `color`, and `title` values. In our example, we are plotting the bar plot between age and income using the `kind` parameter as `'bar'`. The `name` and `age` columns are assigned to the `x` and `y` parameters. The scatter point color is assigned to the `color` parameter. This is all about `pandas` plotting. Now, from the next section onward, we will see how to visualize the data using the Seaborn library.

Advanced visualization using the Seaborn package

Visualization can be helpful to easily understand complex patterns and concepts. It represents the insights in pictorial format. In the preceding sections, we have learned about Matplotlib for visualization. Now, we will explore the new Seaborn library for high-level and advanced statistical plots. Seaborn is an open source Python library for high-level interactive and attractive statistical visualization. Seaborn uses Matplotlib as a base library and offers more simple, easy-to-understand, interactive, and attractive visualizations.

In the Anaconda software suite, you can install the Seaborn library in the following way:

Install Seaborn with pip:

```
pip install seaborn
```

For Python 3, use the following command:

```
pip3 install seaborn
```

You can simply install Seaborn from your terminal or Command Prompt using the following:

```
conda install seaborn
```

If you are installing it into the Jupyter Notebook, then you need to put the ! sign before the pip command. Here is an example:

```
!pip install seaborn
```

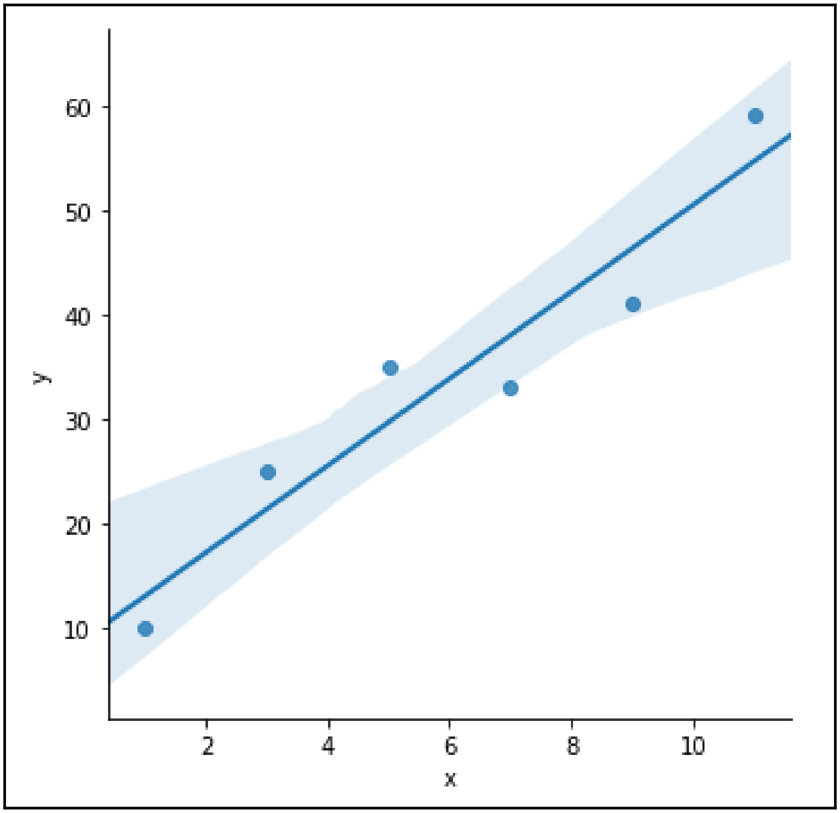
Let's jump to the lm plot of Seaborn.

lm plots

The lm plot plots the scatter and fits the regression model on it. A scatter plot is the best way to understand the relationship between two variables. Its output visualization is a joint distribution of two variables. lmplot() takes two column names – x and y – as a string and DataFrame variable. Let's see the following example:

```
# Import the required libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Create DataFrame
df=pd.DataFrame({'x':[1,3,5,7,9,11],'y':[10,25,35,33,41,59]})

# Create lmplot
sns.lmplot(x='x', y='y', data=df)

# Show figure
plt.show()
```

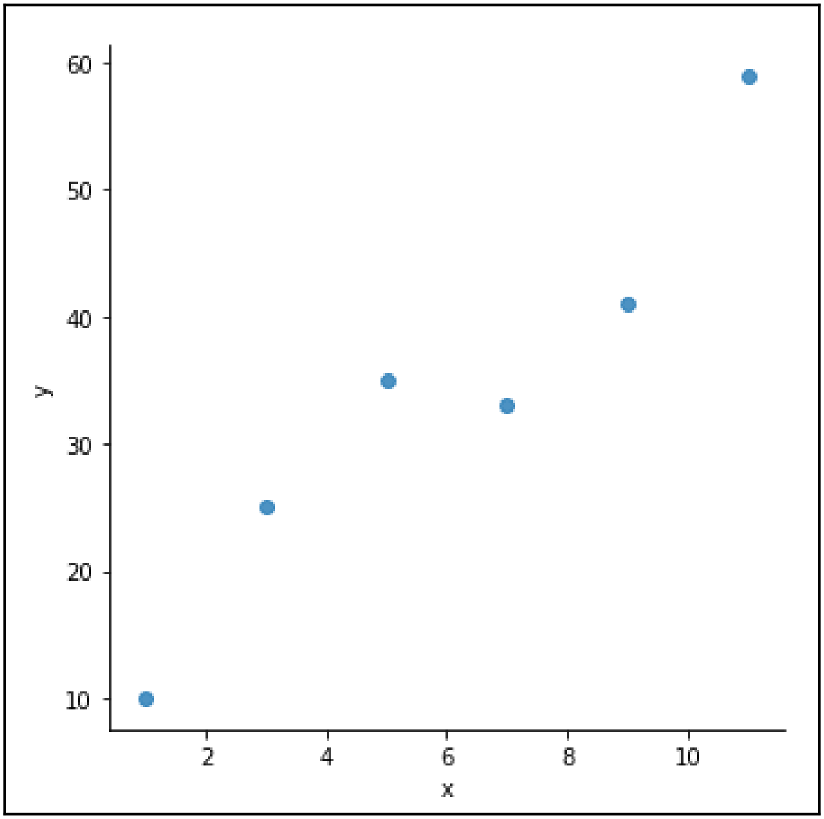
This results in the following output:



By default, `lmplot()` fits the regression line. We can also remove this by setting the `fit_reg` parameter as `False`:

```
# Create lmplot
sns.lmplot(x='x', y='y', data=df, fit_reg=False)

# Show figure
plt.show()
```

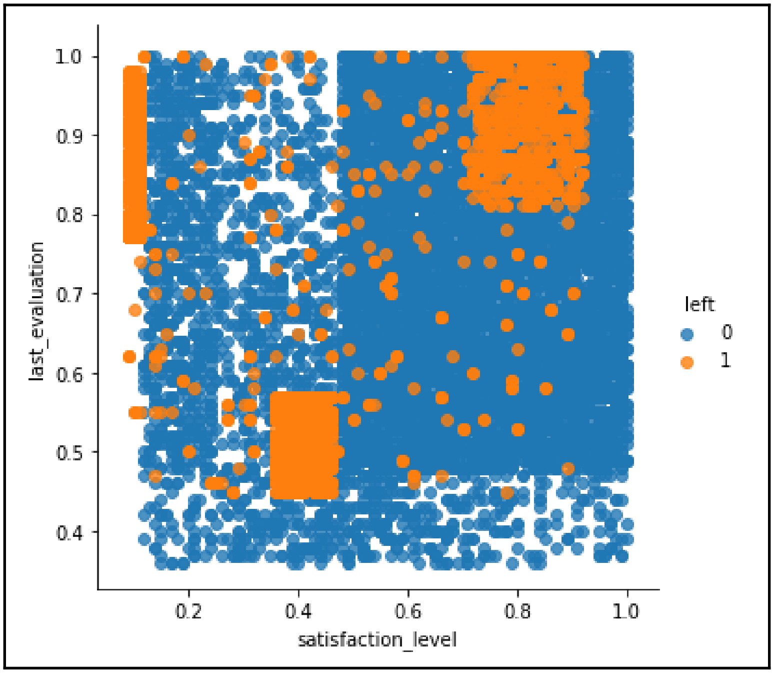
This results in the following output:



Let's take a dataset of HR Analytics and try to plot `lmplot()`:

```
# Load the dataset
df=pd.read_csv("HR_comma_sep.csv")

# Create lmplot
sns.lmplot(x='satisfaction_level', y='last_evaluation', data=df,
fit_reg=False, hue='left')

# Show figure
plt.show()
```

This results in the following output:



In the preceding example, `last_evaluation` is the evaluated performance of the employee, `satisfaction_level` is the employee's satisfaction level in the company, and `left` means whether the employee left the company or not. `satisfaction_level` and `last_evaluation` were drawn on the *x* and *y* axes, respectively. The third variable left is passed in the `hue` parameter. The `hue` property is used for color shade. We are passing a `left` variable as `hue`. We can clearly see in the diagram that employees that have left are scattered into three groups. Let's now jump to bar plots.

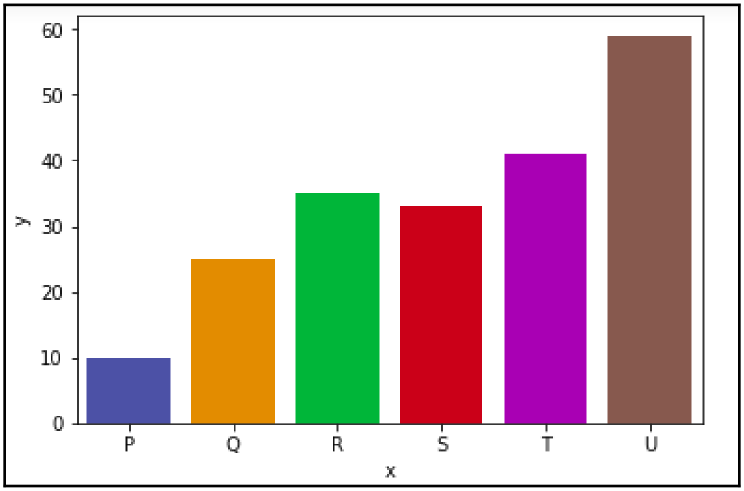
Bar plots

`barplot()` offers the relationship between a categorical and a continuous variable. It uses rectangular bars with variable lengths:

```
# Import the required libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Create DataFrame
df=pd.DataFrame({'x':['P','Q','R','S','T','U'],'y':[10,25,35,33,41,59]})

# Create lmplot
sns.barplot(x='x', y='y', data=df)

# Show figure
plt.show()
```

This results in the following output:



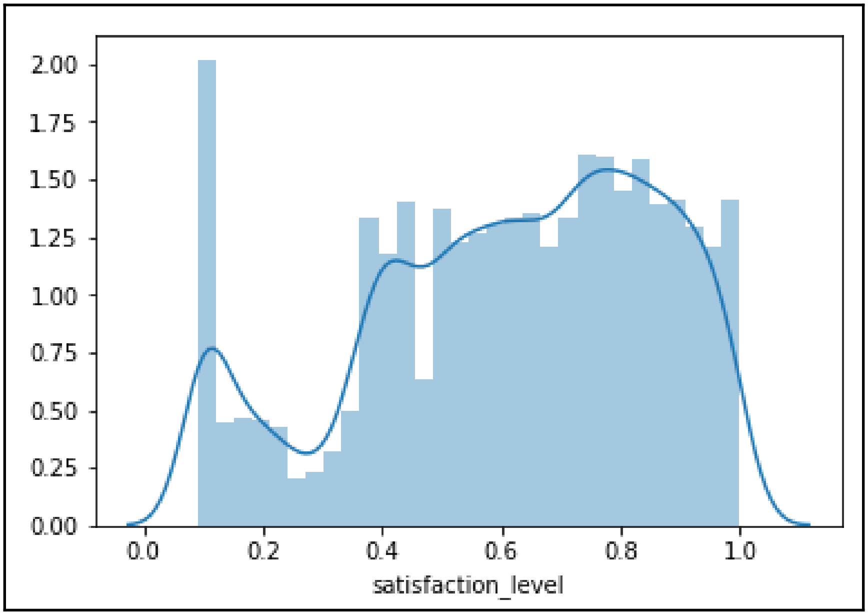
In the preceding example, the bar plot is created using the `bar()` function. It takes two columns – `x` and `y` – and a DataFrame as input. In the next section, we will see how to plot a distribution plot.

Distribution plots

This plots a univariate distribution of variables. It is a combination of a histogram with the default bin size and a **Kernel Density Estimation** (**KDE**) plot. In our example, `distplot()` will take the `satisfaction_level` input column and plot the distribution of it. Here, the distribution of `satisfaction_level` has two peaks:

```
# Create a distribution plot (also known as Histogram)
sns.distplot(df.satisfaction_level)

# Show figure
plt.show()
```

This results in the following output:



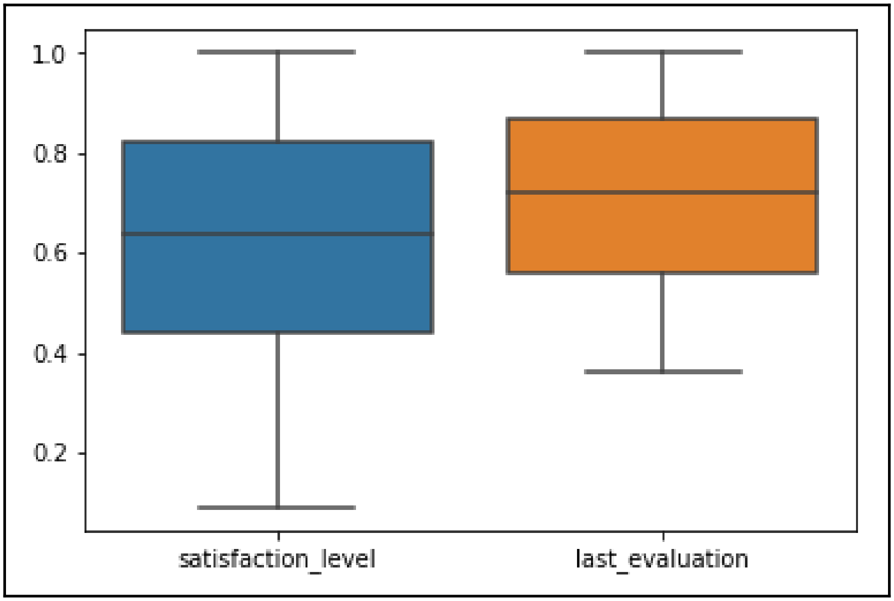
In the preceding code block, we have created the distribution plot using `distplot()`. It's time to jump to the box plot diagram.

Box plots

Box plot, aka box-whisker plot, is one of the best plots to understand the distribution of each variable with its quartiles. It can be horizontal or vertical. It shows quartile distribution in a box, which is known as a whisker. It also shows the minimum and maximum and outliers in the data. We can easily create a box plot using Seaborn:

```
# Create boxplot
sns.boxplot(data=df[['satisfaction_level','last_evaluation']])

# Show figure
plt.show()
```

This results in the following output:



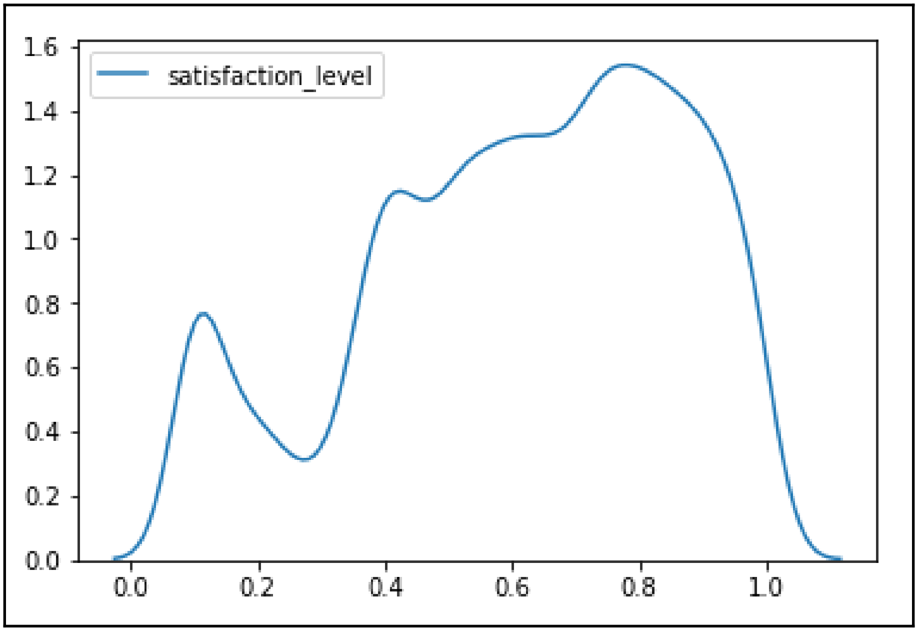
In the preceding example, we have used two variables for the box plot. Here, the box plot indicates that the range of `satisfaction_level` is higher than `last_evaluation` (performance). Let's jump to the KDE plot in Seaborn.

KDE plots

The `kde()` function plots the probability density estimation of a given continuous variable. It is a non-parametric kind of estimator. In our example, the `kde()` function takes one parameter, `satisfaction_level`, and plots the KDE:

```
# Create density plot
sns.kdeplot(df.satisfaction_level)

# Show figure
plt.show()
```

This results in the following output:



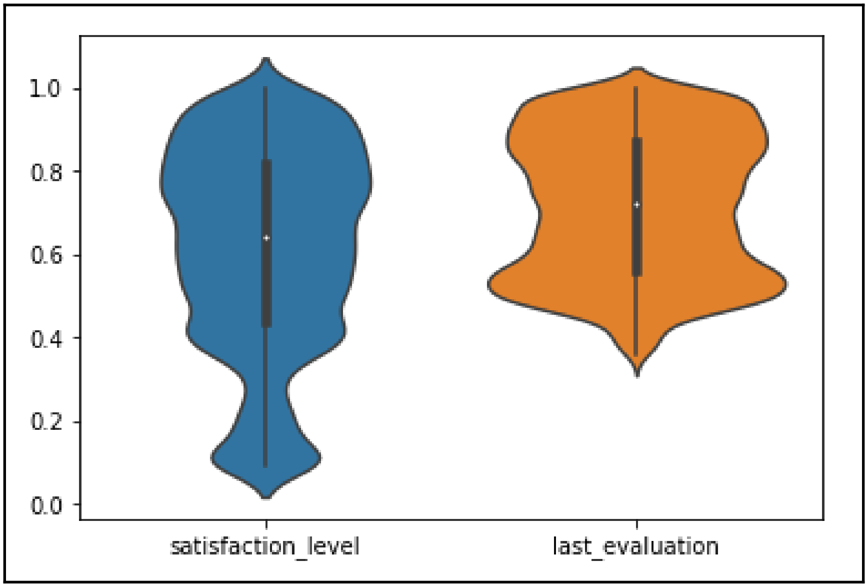
In the preceding code block, we have created a density plot using `kdeplot()`. In the next section, we will see another distribution plot, which is a combination of a density and box plot, known as a violin plot.

Violin plots

Violin plots are a combined form of box plots and KDE, which offer easy-to-understand analysis of the distribution:

```
# Create violin plot
sns.violinplot(data=df[['satisfaction_level','last_evaluation']])

# Show figure
plt.show()
```

This results in the following output:



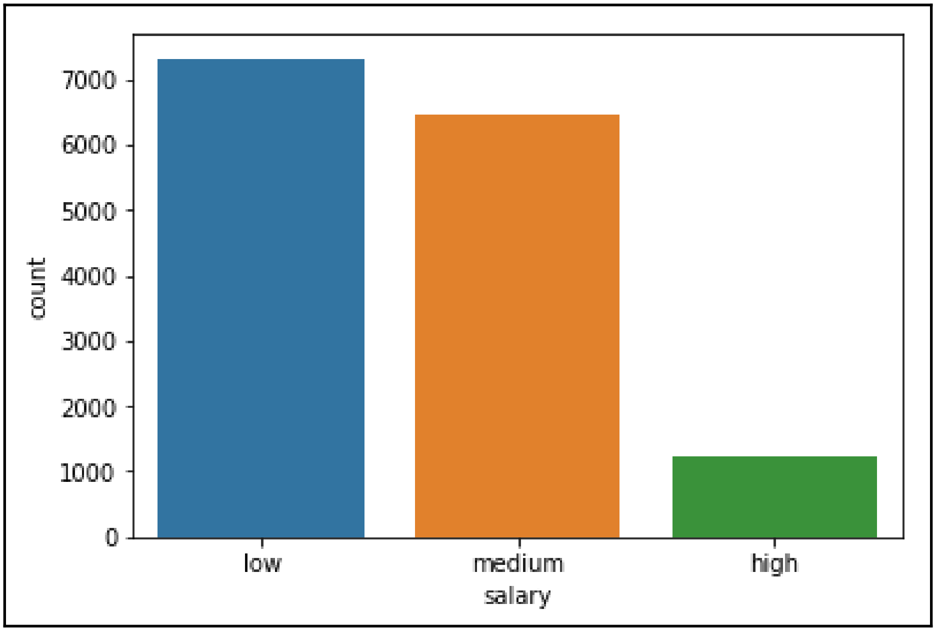
In the preceding example, we have used two variables for the violin plot. Here, we can conclude that the range of `satisfaction_level` is higher than `last_evaluation` (performance) and both the variables have two peaks in the distribution. After working on distribution plots, we will see how we can combine the `groupby` operation and box plot into a single plot using a count plot.

Count plots

`countplot()` is a special type of bar plot. It shows the frequency of each categorical variable. It is also known as a histogram for categorical variables. It makes operations very simple compared to Matplotlib. In Matplotlib, to create a count plot, first we need to group by the category column and count the frequency of each class. After that, this count is consumed by Matplotlib's bar plot. But the Seaborn count plot offers a single line of code to plot the distribution:

```
# Create count plot (also known as Histogram)
sns.countplot(x='salary', data=df)

# Show figure
plt.show()
```

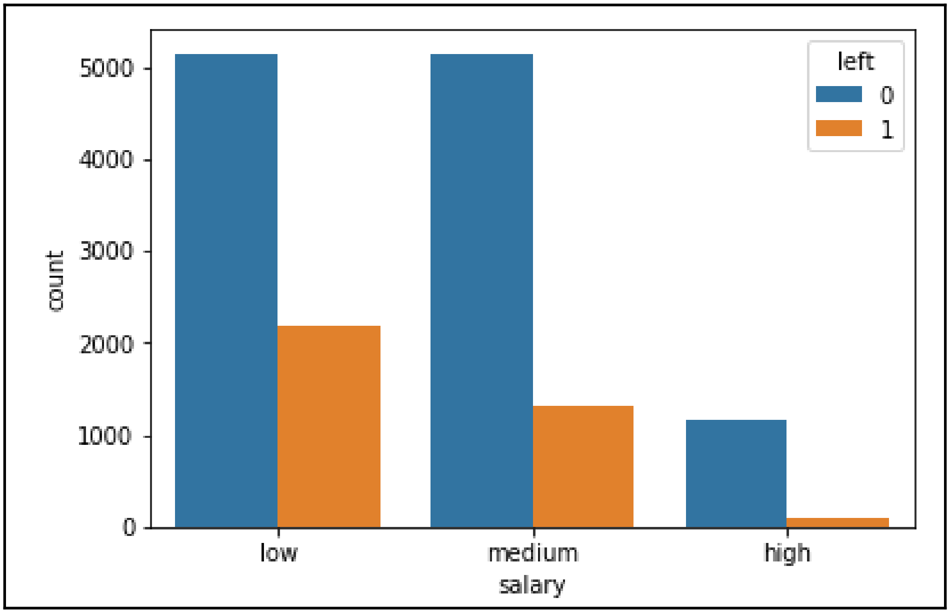
This results in the following output:



In the preceding example, we are counting the `salary` variable. The `count()` function takes a single column and DataFrame. So, we can easily conclude from the graph that most of the employees have low and medium salaries. We can also use `hue` as the second variable. Let's see the following example:

```
# Create count plot (also known as Histogram)
sns.countplot(x='salary', data=df, hue='left')

# Show figure
plt.show()
```

This results in the following output:



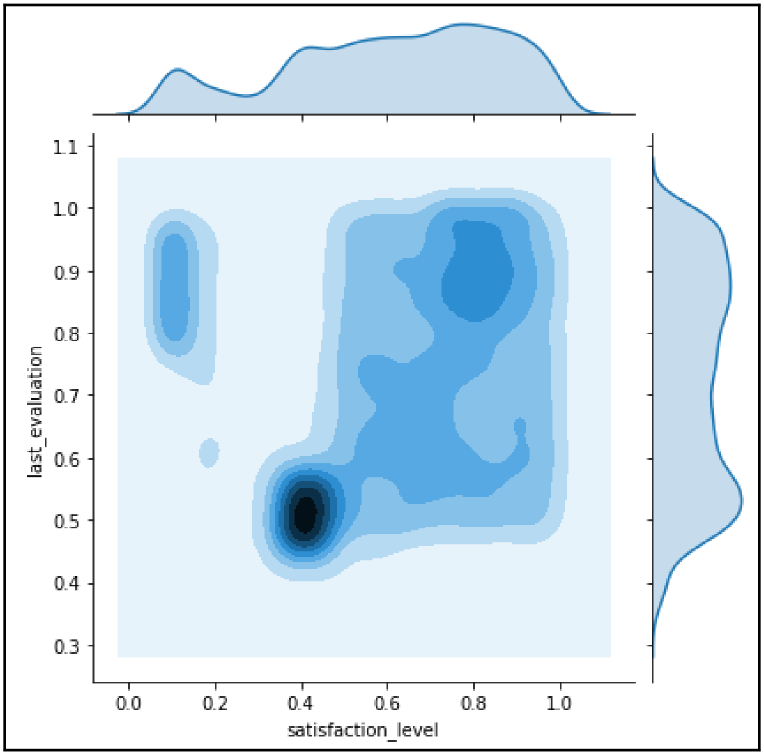
In the preceding example, we can see that `left` is used as the hue or color shade. This indicates that most of the employees with the lowest salary left the company. Let's see another important plot for visualizing the relationship and distribution of two variables.

Joint plots

The joint plot is a multi-panel visualization; it shows the bivariate relationship and distribution of individual variables in a single graph. We can also plot a KDE using the `kind` parameter of `jointplot()`. By setting the `kind` parameter as `"kde"`, we can draw the KDE plot. Let's see the following example:

```
# Create joint plot using kernel density estimation(kde)
sns.jointplot(x='satisfaction_level', y='last_evaluation', data=df,
kind="kde")

# Show figure
plt.show()
```

This results in the following output:



In the preceding plot, we have created the joint plot using `jointplot()` and also added the `kde` plot using a `kind` parameter as `"kde"`. Let's jump to heatmaps for more diverse visualization.

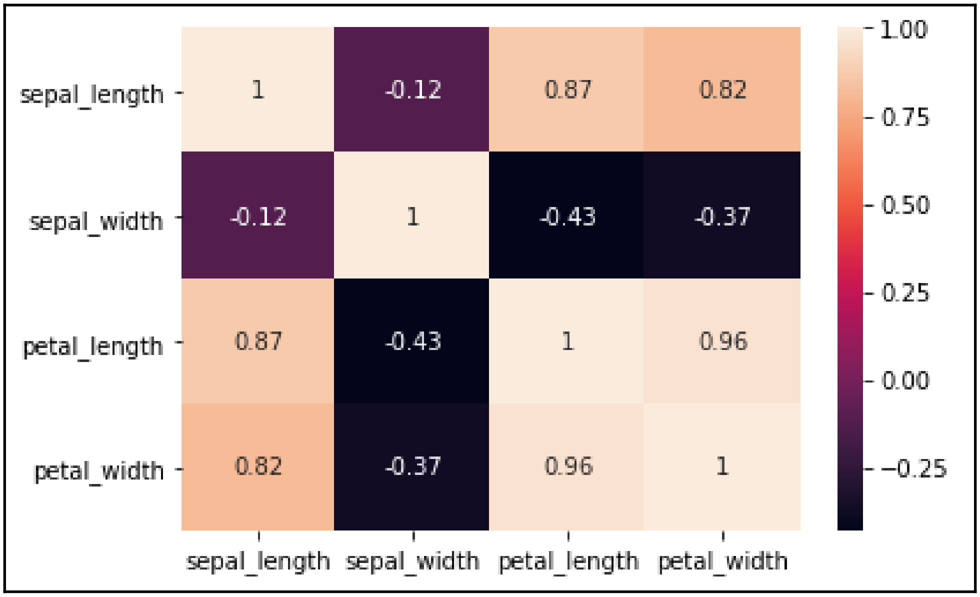
Heatmaps

Heatmap offers two-dimensional grid representation. The individual cell of the grid contains a value of the matrix. The heatmap function also offers annotation on each cell:

```
# Import required library
import seaborn as sns

# Read iris data using load_dataset() function
data = sns.load_dataset("iris")

# Find correlation
```

```
cor_matrix=data.corr()

# Create heatmap
sns.heatmap(cor_matrix, annot=True)

# Show figure
plt.show()
```

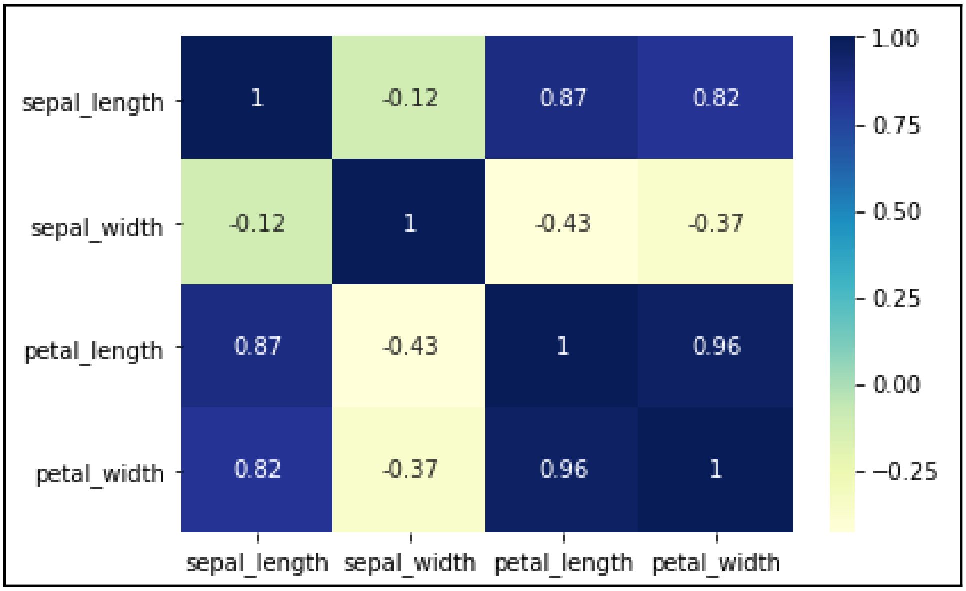
This results in the following output:



In the preceding example, the Iris dataset is loaded using `load_dataset()` and the correlation is calculated using the `corr()` function. The `corr()` function returns the correlation matrix. This correlation matrix is plotted using the `heatmap()` function for the grid view of the correlation matrix. It takes two parameters: the correlation matrix and `annot`. The `annot` parameter is passed as `True`. In the plot, we can see a symmetric matrix, and all the values on the diagonal are ones, which indicates a perfect correlation of a variable with itself. We can also set a new color map using the `cmap` parameter for different colors:

```
# Create heatmap
sns.heatmap(cor_matrix, annot=True, cmap="YlGnBu")

# Show figure
plt.show()
```

This results in the following output:



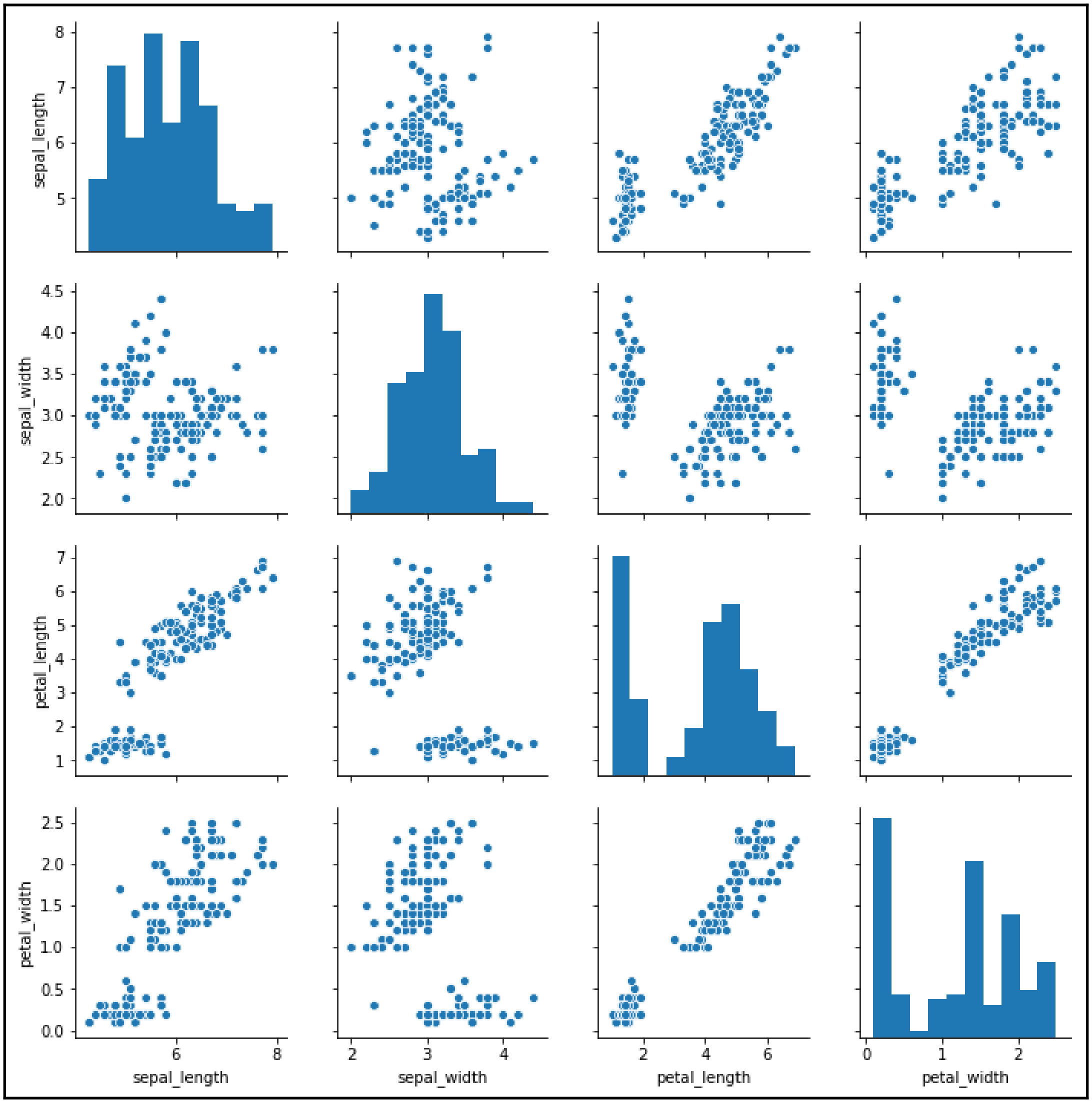
In the preceding heatmap, we have changed the color map using the `cmap` parameter for different colors. Here, we are using the `YlGnBu` (yellow, green, and blue) combination for `cmap`. Now, we will move on to the pair plot for faster exploratory analysis.

Pair plots

Seaborn offers quick exploratory data analysis with relationships and individual distribution using a pair plot. A pair plot offers a single distribution using a histogram and joint distribution using a scatter plot:

```
# Load iris data using load_dataset() function
data = sns.load_dataset("iris")

# Create a pair plot
sns.pairplot(data)

# Show figure
plt.show()
```

This results in the following output:



In the preceding example, the Iris dataset is loaded using `load_dataset()` and that dataset is passed into the `pairplot()` function. In the plot, it creates an *n* by *n* matrix or a grid of graphs. The diagonal shows the distribution of the columns, and the non-diagonal elements of the grid show the scatter plot to understand the relationship among all the variables.

In the preceding few sections, we have seen how to use the Seaborn plots. Now, we will jump to another important visualization library, which is Bokeh. In the upcoming sections, we will draw interactive and versatile plots using the Bokeh library.

Interactive visualization with Bokeh

Bokeh is an interactive, high-quality, versatile, focused, and more powerful visualization library for large-volume and streaming data. It offers interactive, rich charts, plots, layouts, and dashboards for modern web browsers. Its output can be mapped to a notebook, HTML, or server.

Both the Matplotlib and Bokeh libraries have different intentions. Matplotlib focuses on static, simple, and fast visualization while Bokeh focuses on highly interactive, dynamic, web-based, and quality visualization. Matplotlib is generally used for publication images while Bokeh is for a web audience. In the remaining sections of this chapter, we will learn basic plotting using Bokeh. We can create more interactive visuals for data exploration using Bokeh.

The simplest way to install the Bokeh library is with the Anaconda distribution package. To install Bokeh, use the following command:

```
conda install bokeh
```

We can also install it using `pip`. To install Bokeh using `pip`, use the following command:

```
pip install bokeh
```

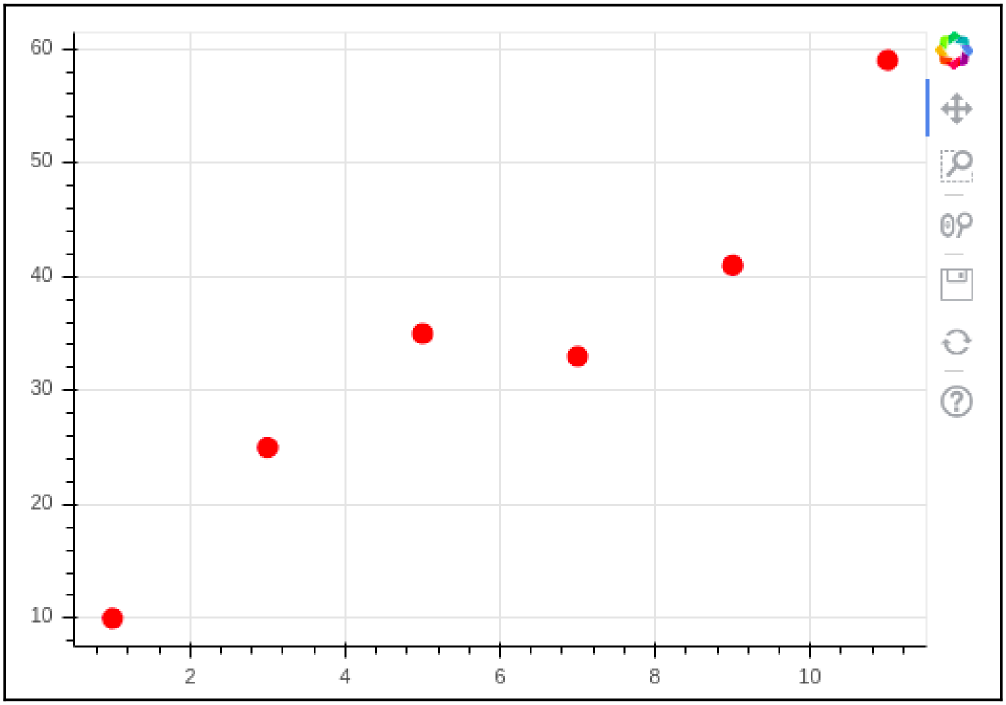
Plotting a simple graph

Let's plot a first and simple plot using Bokeh. First, we need to import the basic `bokeh.plotting` module. The `output_notebook()` function defines that the plot will render on the Jupyter Notebook. The `figure` object is used as one of the core objects to draw charts and graphs. The `figure` object focuses on the plot title, size, label, grids, and style. The `figure` object also deals with plot style, title, axes labels, axes, grids, and various methods for adding data:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show

# Create the data
```

```
x = [1,3,5,7,9,11]
y = [10,25,35,33,41,59]

# Output to notebook
output_notebook()

# Instantiate a figure
fig= figure(plot_width = 500, plot_height = 350)

# Create scatter circle marker plot by rendering the circles
fig.circle(x, y, size = 10, color = "red", alpha = 0.7)

# Show the plot
show(fig)
```

This results in the following output:



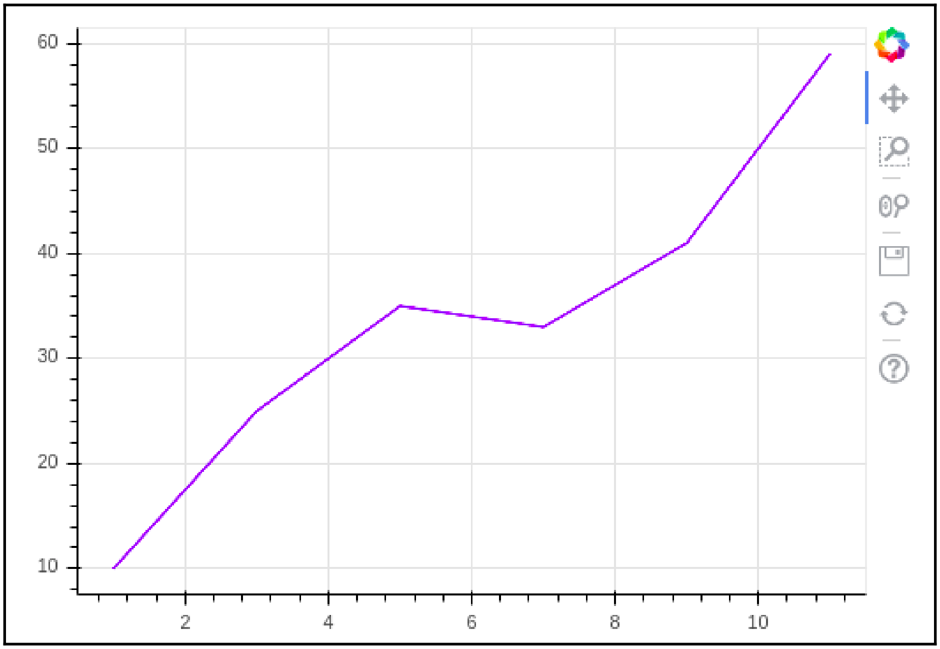
After setting up the figure object, we will create a scatter circle markers plot using a circle function. The circle() function will take x and y values. It also takes size, color, and alpha parameters. The show() function will finally plot the output once all the features and data are added to the plot.

Glyphs

Bokeh uses a visual glyph, which refers to the circles, lines, triangles, squares, bars, diamonds, and other shape graphs. The glyph is a unique symbol that is used to convey information in pictorial form. Let's create a line plot using the line() function:

```
# Import the required modules
from bokeh.plotting import figure, output_notebook, show

# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show

# Create the data
x_values = [1,3,5,7,9,11]
y_values = [10,25,35,33,41,59]

# Output to notebook
output_notebook()

# Instantiate a figure
p = figure(plot_width = 500, plot_height = 350)

# create a line plot
p.line(x_values, y_values, line_width = 1, color = "blue")

# Show the plot
show(p)
```

This results in the following output:



In the preceding example, the `line()` function takes the *x*- and *y*-axis values. It also takes the `line_width` and `color` values of the line. In the next section, we will focus on the layouts for multiple plots.

Layouts

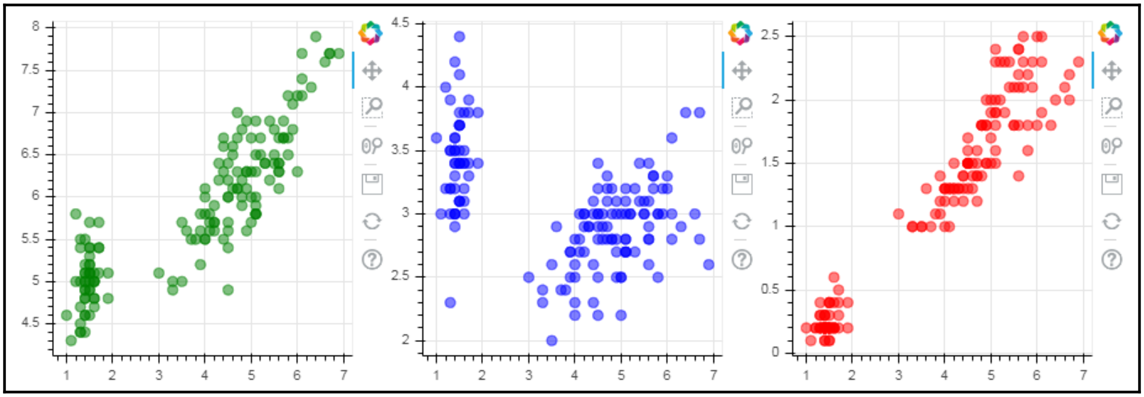
Bokeh offers layouts for organizing plots and widgets. Layouts organize more than one plot in a single panel for interactive visualizations. They also allow setting the sizing modes for resizing the plots and widgets based on panel size. The layout can be of the following types:

- **Row layout**: This organizes all the plots in a row or in a horizontal fashion.
- **Column layout**: This organizes all the plots in a column or in a vertical fashion.
- **Nested layout**: This is a combination of row and column layouts.
- **Grid layout**: This offers you a grid of matrices for arranging the plots in.

Let's see a row layout example:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook, show
from bokeh.layouts import row, column

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Instantiate a figure
fig1 = figure(plot_width = 300, plot_height = 300)
fig2 = figure(plot_width = 300, plot_height = 300)
fig3 = figure(plot_width = 300, plot_height = 300)


# Create scatter marker plot by render the circles
fig1.circle(df['petal_length'], df['sepal_length'], size=8, color =
"green", alpha = 0.5)
fig2.circle(df['petal_length'], df['sepal_width'], size=8, color = "blue",
alpha = 0.5)
fig3.circle(df['petal_length'], df['petal_width'], size=8, color = "red",
alpha = 0.5)

# Create row layout
row_layout = row(fig1, fig2, fig3)

# Show the plot
show(row_layout)
```

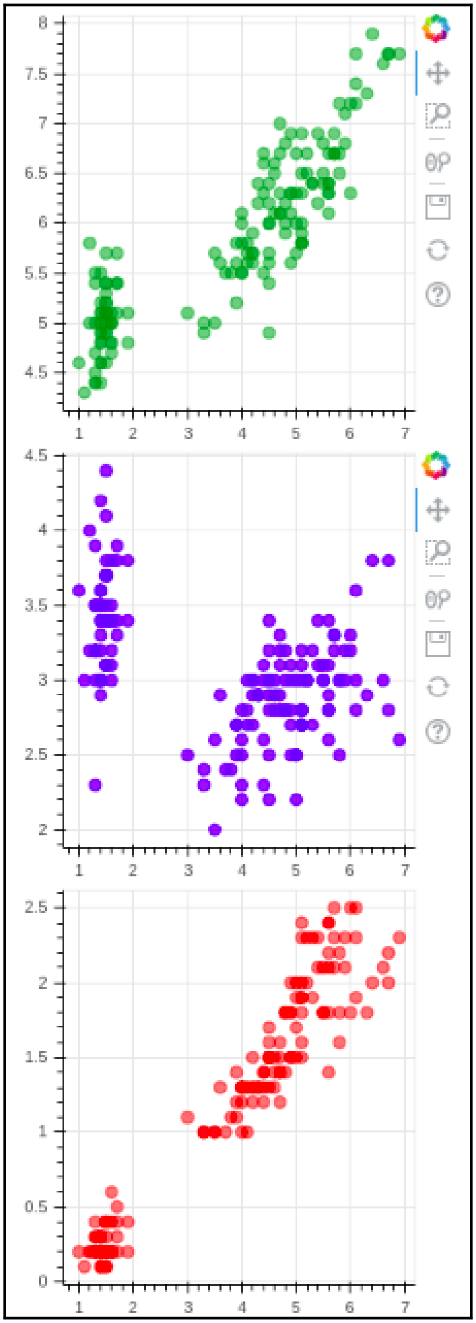
This results in the following output:



In this layout plot, we have imported the row and column layouts, loaded the Iris data from Bokeh sample data, instantiated the three `figure` objects with plot width and height, created the three scatter circle markers on each figure object, and created the row layout. This row layout will take the `figure` objects as input and is drawn using the `show()` function. We can also create a column layout by creating a column layout in place of a row layout, as shown:

```
# Create column layout
col_layout = column(fig1, fig2, fig3)

# Show the plot
show(col_layout)
```

This results in the following output:



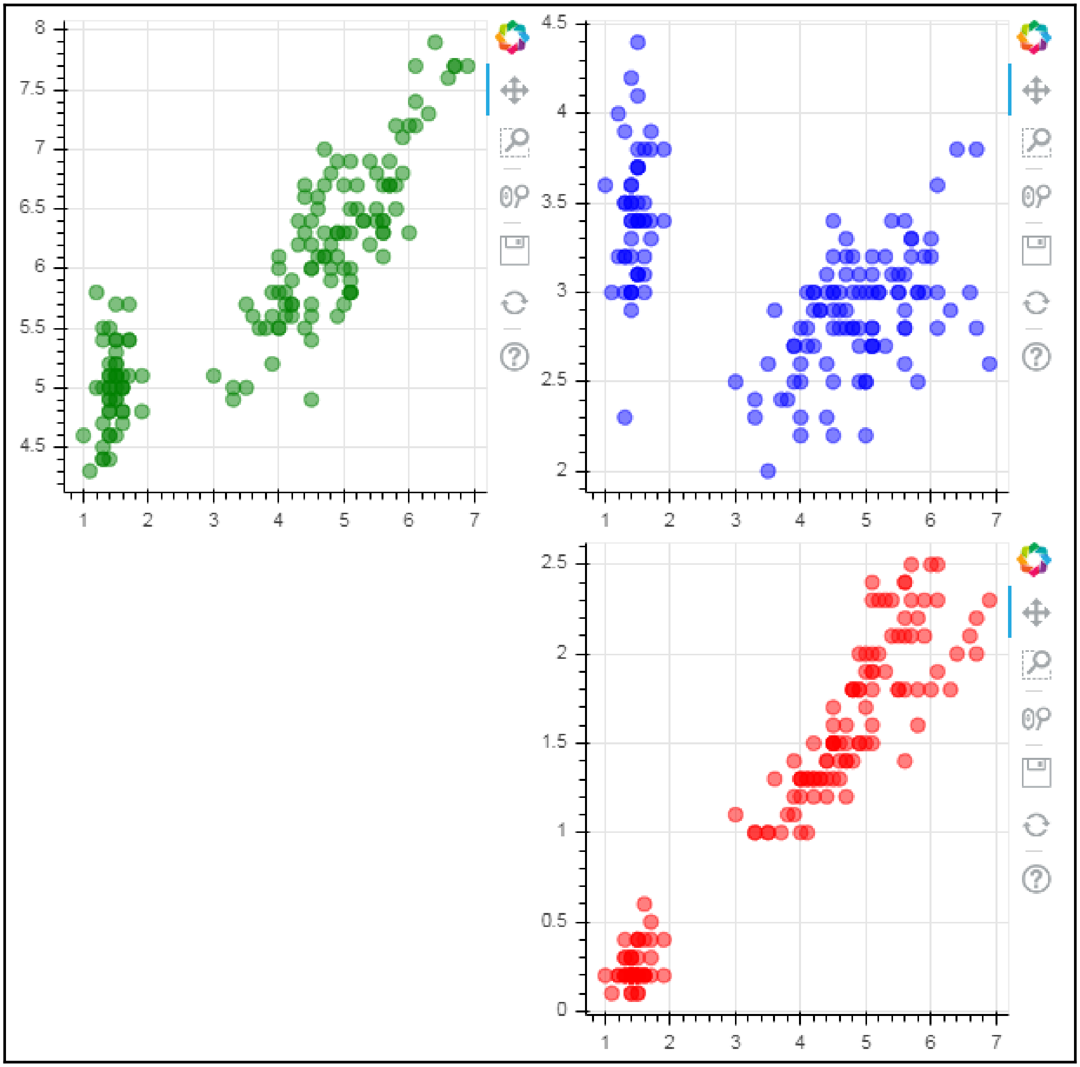
In the preceding plot, we have created the column layout of three plots. Let's jump to the nested layouts for more powerful visualizations.

Nested layout using row and column layouts

A nested layout is the combination of multiple row and column layouts. Let's see the example given here for a better practical understanding:

```
# Import the required modules
from bokeh.plotting import figure, output_notebook, show

# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.layouts import row, column


# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Instantiate a figure
fig1 = figure(plot_width = 300, plot_height = 300)
fig2 = figure(plot_width = 300, plot_height = 300)
fig3 = figure(plot_width = 300, plot_height = 300)

# Create scatter marker plot by render the circles
fig1.circle(df['petal_length'], df['sepal_length'], size=8, color =
"green", alpha = 0.5)
fig2.circle(df['petal_length'], df['sepal_width'], size=8, color = "blue",
alpha = 0.5)
fig3.circle(df['petal_length'], df['petal_width'], size=8, color = "red",
alpha = 0.5)

# Create nested layout
nasted_layout = row(fig1, column(fig2, fig3))

# Show the plot
show(nasted_layout)
```

This results in the following output:



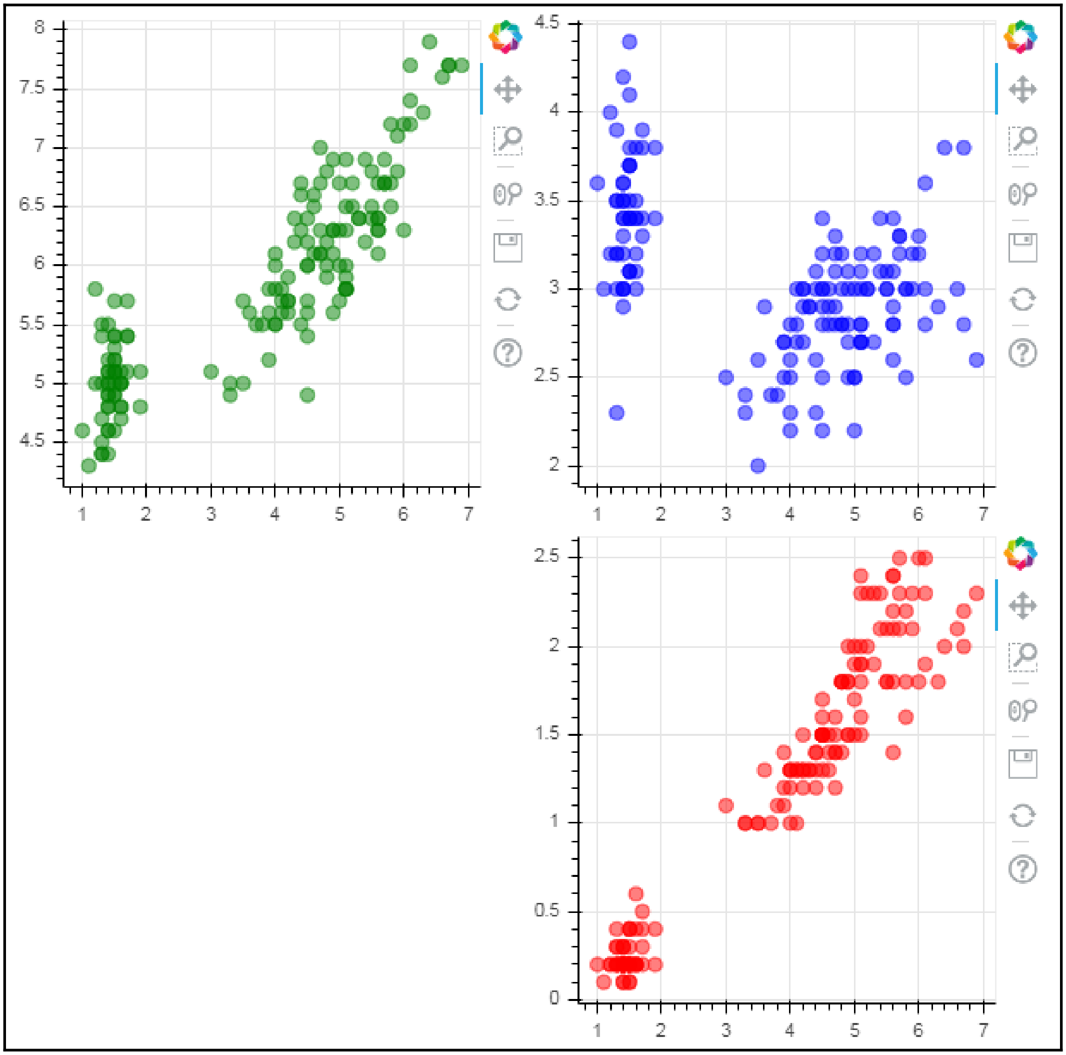
Here, you can see the row layout has two rows. In the first, `fig1` is assigned and the second row has the column layout of `fig2` and `fig3`. So, this layout becomes a 2*2 layout, in which the first column has only one component and the second column has two components.

Multiple plots

Multiple plots and objects can also be created using a grid layout. A grid layout arranges the plots and widget objects in a row-column matrix fashion. It takes a list of figure objects for each row. We can also use None as a placeholder:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.layouts import gridplot

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Instantiate a figure
fig1 = figure(plot_width = 300, plot_height = 300)
fig2 = figure(plot_width = 300, plot_height = 300)
fig3 = figure(plot_width = 300, plot_height = 300)

# Create scatter marker plot by render the circles
fig1.circle(df['petal_length'], df['sepal_length'], size=8, color =
"green", alpha = 0.5)
fig2.circle(df['petal_length'], df['sepal_width'], size=8, color = "blue",
alpha = 0.5)
fig3.circle(df['petal_length'], df['petal_width'], size=8, color = "red",
alpha = 0.5)

# Create a grid layout
grid_layout = gridplot([[fig1, fig2], [None,fig3]])

# Show the plot
show(grid_layout)
```

This results in the following output:



The preceding layout is similar to the nested layout. Here, we have imported `gridplot()`. It arranges the components in rows and columns. The grid plot has taken a list of row figures. The first items in the list are `fig1` and `fig2`. The second items are `None` and `fig3`. Each item is a row in the grid matrix. The `None` placeholder is used to leave the cell empty or without components.

Sizing modes can help us to configure figures with resizing options. Bokeh offers the following sizing modes:

- `fixed`: This retains the same original width and height.
- `stretch_width`: This stretches to the available width based on the type of the other component. It doesn't maintain the aspect ratio.
- `stretch_height`: This stretches to the available height based on the type of the other component. It doesn't maintain the aspect ratio.
- `stretch_both`: This stretches both the width and height based on the type of the other component without maintaining the original aspect ratio.
- `scale_width`: This stretches to the available width based on the type of the other component while maintaining the original aspect ratio.
- `scale_height`: This stretches to the available height based on the type of the other component while maintaining the original aspect ratio.
- `scale_both`: This stretches both the width and height based on the type of the other component while maintaining the original aspect ratio.

After learning about layouts and multiple plots, it's time to learn about interactions for interactive visualizations.

Interactions

Bokeh offers interactive legends for runtime-actionable graphs. Legends can be hidden or muted by clicking on glyph plots. We can activate these modes by activating the `click_policy` property and clicking on the legend entry.

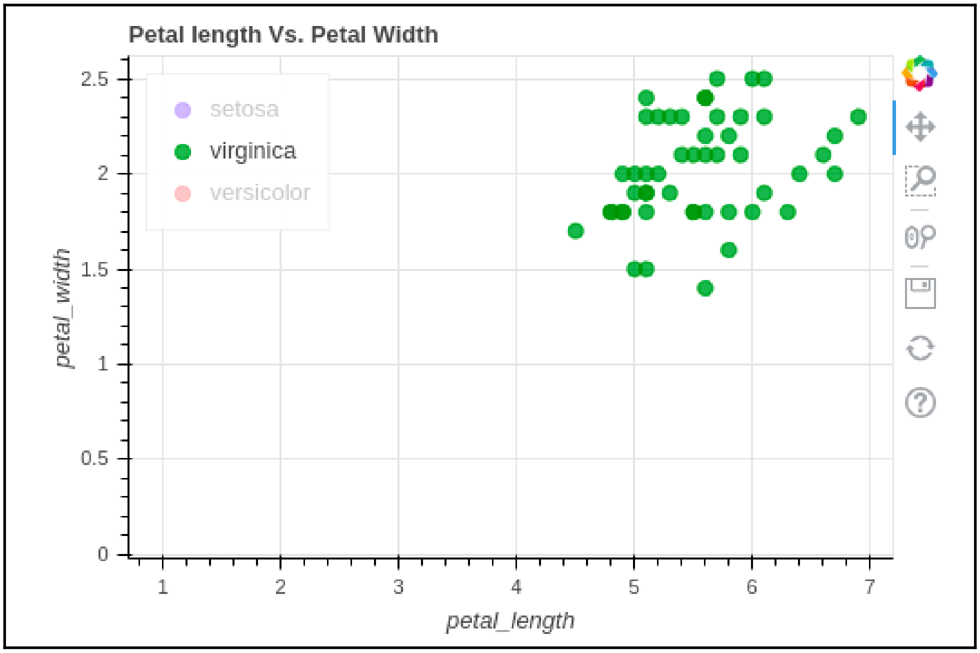
Hide click policy

Hide click policy hides the desirable glyphs by clicking on the legend entry. Let's see an example of a hide click policy:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models import CategoricalColorMapper

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df
```

```
# Output to notebook
output_notebook()


# Instantiate a figure object
fig = figure(plot_width = 500, plot_height = 350, title="Petal length Vs.
Petal Width",
           x_axis_label='petal_length', y_axis_label='petal_width')

# Create scatter marker plot by render the circles
for specie, color in zip(['setosa', 'virginica','versicolor'], ['blue',
'green', 'red']):
    data = df[df.species==specie]
    fig.circle('petal_length', 'petal_width', size=8, color=color, alpha =
0.7, legend_label=specie, source=data)

# Set the legend location and click policy
fig.legend.location = 'top_left'
fig.legend.click_policy="hide"

# Show the plot
show(fig)
```

This results in the following output:



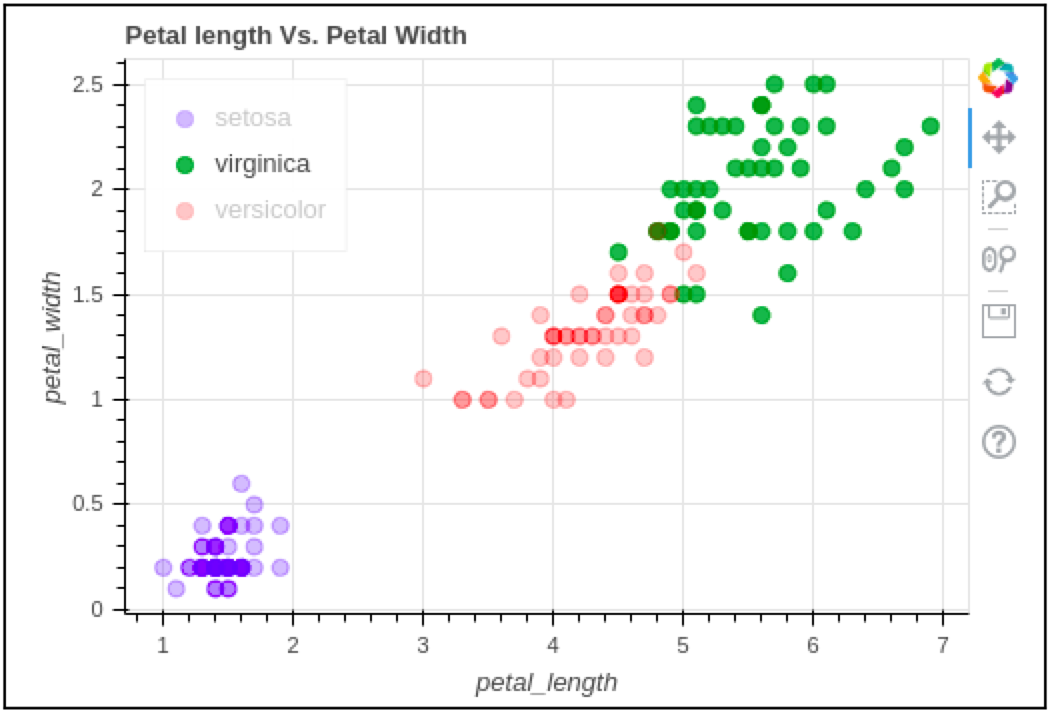
Here, we can set the click policy with the `legend.click_policy` parameter of the `figure` object. Also, we need to run a `for` loop of each type of glyph or legend element on which you click. In our example, we are running a `for` loop for types of species and colors. On the click of any species in the legend, it will filter the data and hide that glyph.

Mute click policy

Mute click policy mutes the glyph by clicking on a legend entry. Here, the following code shows the desirable glyph with high intensity and uninteresting glyphs using lower intensity instead of hiding the whole glyph. Let's see an example of a mute click policy:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models import CategoricalColorMapper

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()


# Instantiate a figure object
fig = figure(plot_width = 500, plot_height = 350, title="Petal length Vs.
Petal Width",
           x_axis_label='petal_length', y_axis_label='petal_width')

# Create scatter marker plot by render the circles
for specie, color in zip(['setosa', 'virginica','versicolor'], ['blue',
'green', 'red']):
    data = df[df.species==specie]
    fig.circle('petal_length', 'petal_width', size=8, color=color, alpha =
0.7,legend_label=specie,source=data,
            muted_color=color, muted_alpha=0.2)

# Set the legend location and click policy
fig.legend.location = 'top_left'
fig.legend.click_policy="mute"

# Show the plot
show(fig)
```

This results in the following output:



Here, we can set the mute click policy with the `legend.click_policy` parameter to mute figure objects. Also, we need to run the `for` loop of each type of glyph or legend element on which you click. In our example, we are running a `for` loop for types of species and colors. On the click of any species in the legend, it will filter the data and hide that glyph. In addition to that, we need to add a `muted_color` and `muted_alpha` parameter to the scatter circle marker.

Annotations

Bokeh offers several annotations for supplementary information for visualizations. It helps the viewer by adding the following information:

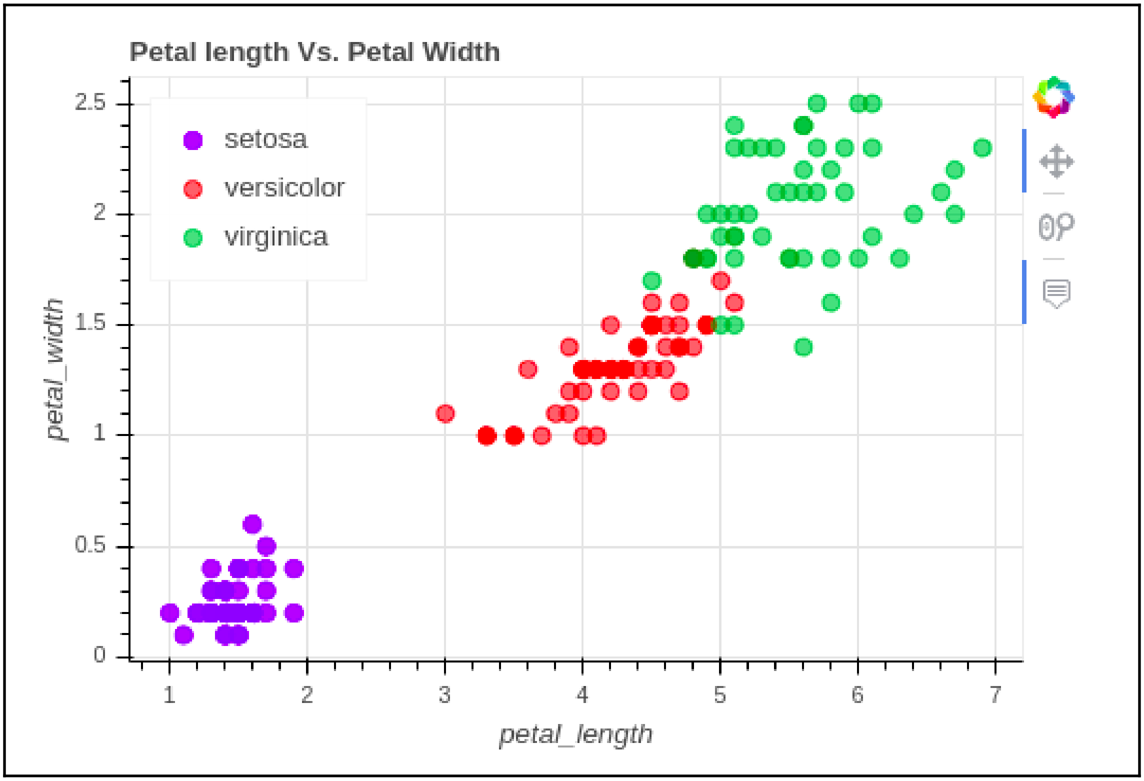
- **Titles**: This annotation provides a name to the plot.
- **Axis labels**: This annotation provides labels to the axis. It helps us to understand what the *x* and *y* axes represent.

- **Legends**: This annotation represents the third variable via color or shape and helps us to link features for easy interpretations.
- **Color bars**: Color bars are created using ColorMapper with the color palette.

Let's see an annotation example:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models import CategoricalColorMapper

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Create color mapper for categorical column
color_mapper = CategoricalColorMapper(factors=['setosa', 'virginica',
'versicolor'], palette=['blue', 'green', 'red'])

color_dict={'field': 'species','transform': color_mapper }

# Instantiate a figure object
p = figure(plot_width = 500, plot_height = 350, title="Petal length Vs.
Petal Width",
           x_axis_label='petal_length', y_axis_label='petal_width')

# Create scatter marker plot by render the circles
p.circle('petal_length', 'petal_width', size=8, color=color_dict, alpha =
0.5, legend_group='species', source=df)

# Set the legend location
p.legend.location = 'top_left'

# Show the plot
show(p)
```

This results in the following output:



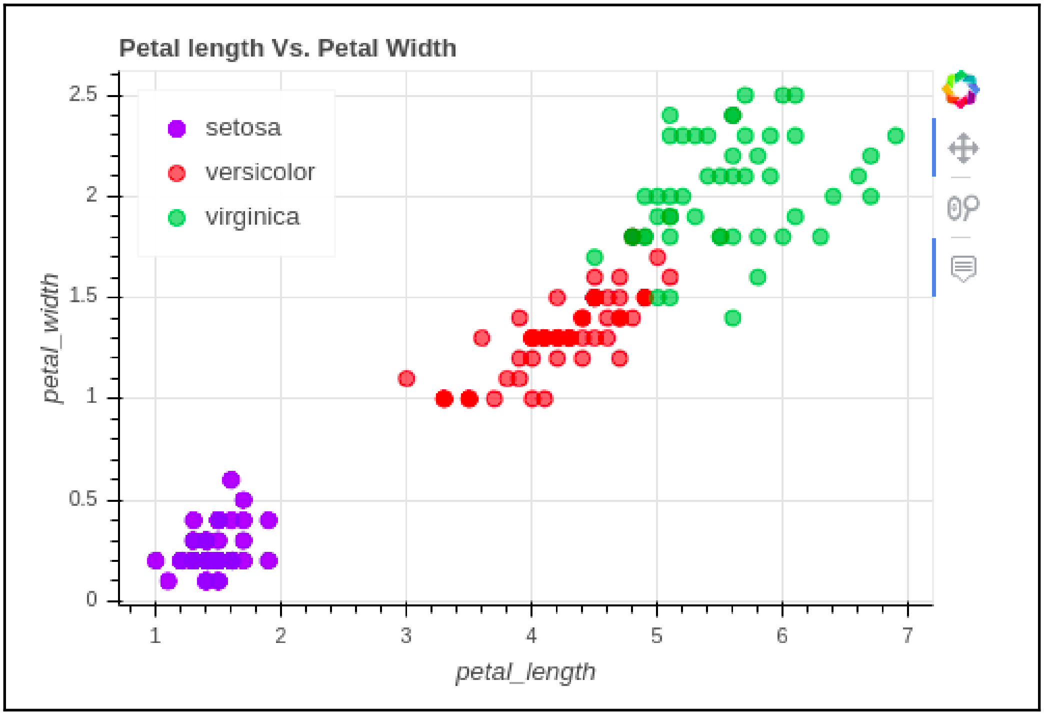
In the preceding example, `CategoricalColorMapper` is imported and objects are created by defining factors or unique items in iris species and their respective colors. A color dictionary is created by defining the `field` and `transform` parameters for the mapper. We need to define the figure title; `x_axis_label` and `y_axis_label` were defined inside the `figure` object. The legend is defined in the circle scatter marker function with the species column and its location is defined using the location attribute of the `figure` object with `top_left`.

Hover tool

The hover tool shows the related information whenever the mouse pointer is placed over a particular area. Let's see examples to understand the hovering plots:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models import CategoricalColorMapper
from bokeh.models import HoverTool

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Create color mapper for categorical column
mapper = CategoricalColorMapper(factors=['setosa', 'virginica',
'versicolor'],
                                palette=['blue', 'green', 'red'])

color_dict={'field': 'species','transform': mapper}

# Create hovertool and specify the hovering information
hover = HoverTool(tooltips=[('Species type','@species'),
                            ('IRIS Petal Length','@petal_length'),
                            ('IRIS Petal Width', '@petal_width')])

# Instantiate a figure object
p = figure(plot_width = 500, plot_height = 350, title="Petal length Vs.
Petal Width",
           x_axis_label='petal_length', y_axis_label='petal_width',
           tools=[hover, 'pan', 'wheel_zoom'])

# Create scatter marker plot by render the circles
p.circle('petal_length', 'petal_width', size=8, color=color_dict, alpha =
0.5,legend_group='species',source=df)

# Set the legend location
p.legend.location = 'top_left'

# Show the plot
show(p)
```

This results in the following output:



In the preceding example, we have imported `HoverTool` from `bokeh.models` and created its object by defining the information that will be shown on mouse hover. In our example, we have defined information in the list of tuples. Each tuple has two arguments. The first is for the string label and the second is for the actual value (preceded with `@`). This hover object is passed into the `figure` object's `tools` parameter.

Widgets

Widgets offer real-time interaction on the frontend. Widgets have the capability to modify and update plots at runtime. They can run either a Bokeh server or a standalone HTML application. For using widgets, you need to specify the functionality. It can be nested inside the layout. There are two approaches to add the functionality of widgets into the program:

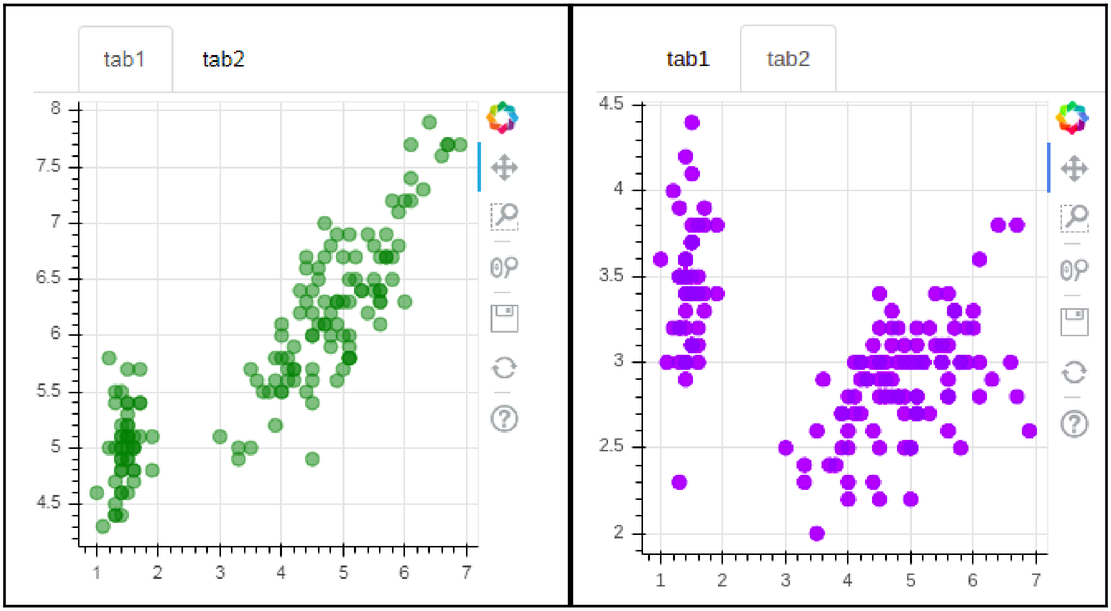
- CustomJS callback
- With the Bokeh server and setup event handler, such as `onclick` or `onchange` `event`

Tab panel

Tab panes allow us to create multiple plots and layouts in a single window. Let's see an example of a tab panel:

```
# Import the required modules
from bokeh.plotting import figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models.widgets import Tabs
from bokeh.models.widgets import Panel

# Import iris flower dataset as pandas DataFrame
from bokeh.sampledata.iris import flowers as df

# Output to notebook
output_notebook()

# Instantiate a figure
fig1 = figure(plot_width = 300, plot_height = 300)
fig2 = figure(plot_width = 300, plot_height = 300)

# Create scatter marker plot by render the circles
fig1.circle(df['petal_length'], df['sepal_length'], size=8, color =
"green", alpha = 0.5)
fig2.circle(df['petal_length'], df['sepal_width'], size=8, color = "blue",
alpha = 0.5)

# Create panels
tab1 = Panel(child=fig1, title='tab1')
tab2 = Panel(child=fig2, title='tab2')

# Create tab by putting panels into it
tab_layout = Tabs(tabs=[tab1,tab2])

# Show the plot
show(tab_layout)
```

This results in the following output:



In the preceding code, we have created the two panels by passing `figure` objects to a child parameter and `title` to a `title` parameter to `Panel`. Both panels are combined into a list and passed to the `Tabs` layout object. This `Tabs` object is shown by the `show()` function. You can change the tab by just clicking on it.

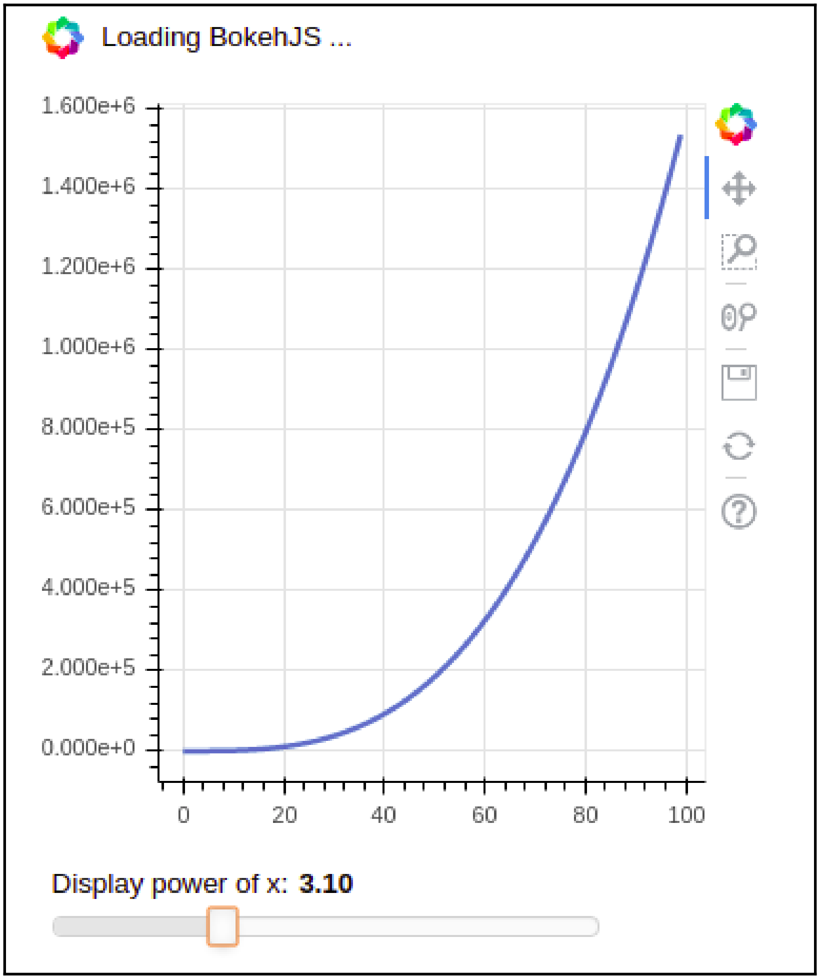
Slider

A slider is a graphical track bar that controls the value by moving it on a horizontal scale. We can change the values of the graph without affecting its formatting. Let's see an example of a slider:

```
# Import the required modules
from bokeh.plotting import Figure
from bokeh.plotting import output_notebook
from bokeh.plotting import show
from bokeh.models import CustomJS
from bokeh.models import ColumnDataSource
from bokeh.models import Slider
from bokeh.layouts import column

# Show output in notebook
```

```
output_notebook()

# Create list of data
x = [x for x in range(0, 100)]
y = x

# Create a DataFrame
df = ColumnDataSource(data={"x_values":x, "y_values":y})

# Instantiate the Figure object
fig = Figure(plot_width=350, plot_height=350)

# Create a line plot
fig.line('x_values', 'y_values', source=df, line_width=2.5, line_alpha=0.8)

# Create a callback using CustomJS
callback = CustomJS(args=dict(source=df), code="""
    var data = source.data;
    var f = cb_obj.value
    var x_values = data['x_values']
    var y_values = data['y_values']
    for (var i = 0; i < x_values.length; i++) {
        y_values[i] = Math.pow(x_values[i], f)
    }
    source.change.emit();
""")

slider_widget = Slider(start=0.0, end=10, value=1, step=.1, title="Display
power of x")

slider_widget.js_on_change('value', callback)

# Create layout
slider_widget_layout = column(fig,slider_widget)

# Display the layout
show(slider_widget_layout)
```

This results in the following output:



In the preceding code, the Bokeh `slider()` function takes `start`, `end`, `value`, `step`, `title`, and CustomJS callback as input. In our example, we are creating a line graph and changing its `y` variable by the power of the `x` variable using the slide bar. We can create the slider by passing `start`, `end`, `value`, `step`, `title`, and a CustomJS callback to the `Slider` object. We need to focus on the CustomJS callback. It takes the source DataFrame, gets the value of the slider using `cb_obj.value`, and updates its values using the `change.emit()` function. We are updating `y_value` in the `for` loop by finding its power using the slider value.

Summary

In this chapter, we discussed visualizing data using plotting with Matplotlib, `pandas`, Seaborn, and Bokeh. We covered various plots, such as line plots, pie plots, bar plots, histograms, scatter plots, box plots, bubble charts, heatmaps, KDE plots, violin plots, count plots, joint plots, and pair plots. We focused on accessories for charts, such as titles, labels, legends, layouts, subplots, and annotations. Also, we learned about interactive visualization using Bokeh layouts, interactions, hover tools, and widgets.

The next chapter, `Chapter 6`, *Retrieving, Processing, and Storing Data*, will teach us skills of data reading and writing from various sources such as files, objects, and relational and NoSQL databases. Although some people don't consider these skills for data analysis, an independent or assistant data analyst must know how they can fetch data from various file formats and databases for analysis purposes.

6

Retrieving, Processing, and Storing Data

Data can be found everywhere, in all shapes and forms. We can get it from the web, IoT sensors, emails, FTP, and databases. We can also collect it ourselves in a lab experiment, election polls, marketing polls, and social surveys. As a data professional, you should know how to handle a variety of datasets as that is a very important skill. We will discuss retrieving, processing, and storing various types of data in this chapter. This chapter offers an overview of how to acquire data in various formats, such as CSV, Excel, JSON, HDF5, Parquet, and `pickle`.

Sometimes, we need to store or save the data before or after the data analysis. We will also learn how to access data from relational and **NoSQL** (**Not Only SQL**) databases such as `sqlite3`, MySQL, MongoDB, Cassandra, and Redis. In the world of the21st-century web, NoSQL databases are undergoing substantial growth in big data and web applications. They provide a more flexible, faster, and schema-free database. NoSQL databases can store data in various formats, such as document style, column-oriented, objects, graphs, tuples, or a combination.

The topics covered in this chapter are listed as follows:

- Reading and writing CSV files with NumPy
- Reading and writing CSV files with pandas
- Reading and writing data from Excel
- Reading and writing data from JSON
- Reading and writing data from HDF5
- Reading and writing data from HTML tables
- Reading and writing data from Parquet
- Reading and writing data from a `pickle pandas` object
- Lightweight access with `sqllite3`
- Reading and writing data from MySQL

- Reading and writing data from MongoDB
- Reading and writing data from Cassandra
- Reading and writing data from Redis
- PonyORM

Technical requirements

This chapter has the following technical requirements:

- You can find the code and the dataset at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter06`.
- All the code blocks are available in the `ch6.ipynb` file.
- This chapter uses CSV files (`demo.csv`, `product.csv`, `demo_sample_df.csv`, `my_first_demo.csv`, and `employee.csv`), Excel files (`employee.xlsx`, `employee_performance.xlsx`, and `new_employee_details.xlsx`), JSON files (`employee.json` and `employee_demo.json`), an HTML file (`country.html`), a `pickle` file (`demo_obj.pkl`), an HDF5 file (`employee.h5`), and a Parquet file (`employee.parquet`) for practice purposes.
- In this chapter, we will use the `pandas`, `pickle`, `pyarrow`, `sqlite3`, `pymysql`, `mysql-connector`, `pymongo`, `cassandra-driver`, and `redis` Python libraries.

Reading and writing CSV files with NumPy

In `Chapter 2`, *NumPy and pandas*, we looked at the NumPy library in detail and explored lots of functionality. NumPy also has functions to read and write CSV files and get output in a NumPy array. The `genfromtxt()` function will help us to read the data and the `savetxt()` function will help us to write the data into a file. The `genfromtxt()` function is slow compared to other functions due to its two-stage operation. In the first stage, it reads the data in a string type, and in the second stage, it converts the string type into suitable data types. `genfromtxt()` has the following parameters:

- `fname`: String; filename or path of the file.
- `delimiter`: String; optional, separate string value. By default, it takes consecutive white spaces.
- `skip_header`: Integer; optional, number of lines you want to skip from the start of the file.

Let's see an example of reading and writing CSV files:

```
# import genfromtxt function
from numpy import genfromtxt

# Read comma separated file
product_data = genfromtxt('demo.csv', delimiter=',')

# display initial 5 records
print(product_data)
```

This results in the following output:

```
[[14. 32. 33.]
 [24. 45. 26.]
 [27. 38. 39.]]
```

In the preceding code example, we are reading the `demo.csv` file using the `genfromtxt()` method of the NumPy module:

```
# import numpy
import numpy as np

# Create a sample array
sample_array = np.asarray([ [1,2,3], [4,5,6], [7,8,9] ])

# Write sample array to CSV file
np.savetxt("my_first_demo.csv", sample_array, delimiter=",")
```

In the preceding code example, we are writing the `my_first_demo.csv` file using the `savetxt()` method of the NumPy module.

Let's see how can we read CSV files using the `pandas` module in the next section.

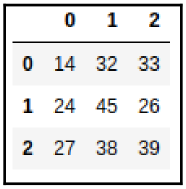
Reading and writing CSV files with pandas

The `pandas` library provides a variety of file reading and writing options. In this section, we will learn about reading and writing CSV files. In order to read a CSV file, we will use the `read_csv()` method. Let's see an example:

```
# import pandas
import pandas as pd

# Read CSV file
sample_df=pd.read_csv('demo.csv', sep=',' , header=None)
```

```
# display initial 5 records
sample_df.head()
```

This results in the following output:



	0	1	2
0	14	32	33
1	24	45	26
2	27	38	39

We can now save the dataframe as a CSV file using the following code:

```
# Save DataFrame to CSV file
sample_df.to_csv('demo_sample_df.csv')
```

In the preceding sample code, we have read and saved the CSV file using the `read_csv()` and `to_csv(0)` methods of the `pandas` module.

The `read_csv()` method has the following important arguments:

- `filepath_or_buffer`: Provides a file path or URL as a string to read a file.
- `sep`: Provides a separator in the string, for example, comma as `','` and semicolon as `';'`. The default separator is a comma `','`.
- `delim_whitespace`: Alternative argument for a white space separator. It is a Boolean variable. The default value for `delim_whitespace` is `False`.
- `header`: This is used to identify the names of columns. The default value is `infer`.
- `names`: You can pass a list of column names. The default value for `names` is `None`.

In `pandas`, a DataFrame can also be exported in a CSV file using the `to_csv()` method. CSV files are comma-separated values files. This method can run with only a single argument (filename as a string):

- `path_or_buf`: The file path or location where the file will export.
- `sep`: This is a delimiter used for output files.
- `header`: To include column names or a list of column aliases (default value: `True`).
- `index`: To write an index to the file (default value: `True`).

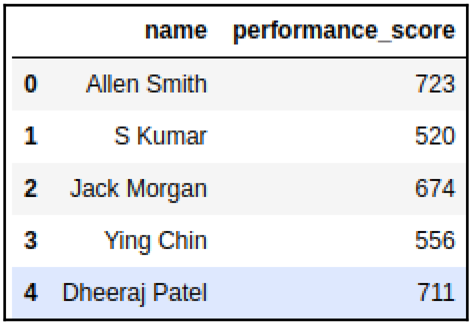
For more parameters and detailed descriptions, visit https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html. Let's see how can we read Excel files using the pandas module in the next section.

Reading and writing data from Excel

Excel files are widely used files in the business domain. Excel files can be easily read in Python's pandas using the read_excel() function. The read_excel() function takes a file path and sheet_name parameters to read the data:

```
# Read excel file
df=pd.read_excel('employee.xlsx',sheet_name='performance')

# display initial 5 records
df.head()
```

This results in the following output:



	name	performance_score
0	Allen Smith	723
1	S Kumar	520
2	Jack Morgan	674
3	Ying Chin	556
4	Dheeraj Patel	711

DataFrame objects can be written on Excel sheets. We can use the to_excel() function to export DataFrame objects into an Excel sheet. Mostly, the to_excel() function arguments are the same as to_csv() except for the sheet_name argument:

```
df.to_excel('employee_performance.xlsx')
```

In the preceding code example, we have exported a single DataFrame into an Excel sheet. We can also export multiple DataFrames in a single file with different sheet names. We can also write more than one DataFrame in a single Excel file (each DataFrame on different sheets) using ExcelWriter, as shown:

```
# Read excel file
emp_df=pd.read_excel('employee.xlsx',sheet_name='employee_details')
```

```
# write multiple dataframes to single excel file
with pd.ExcelWriter('new_employee_details.xlsx') as writer:
    emp_df.to_excel(writer, sheet_name='employee')
    df.to_excel(writer, sheet_name='perfromance')
```

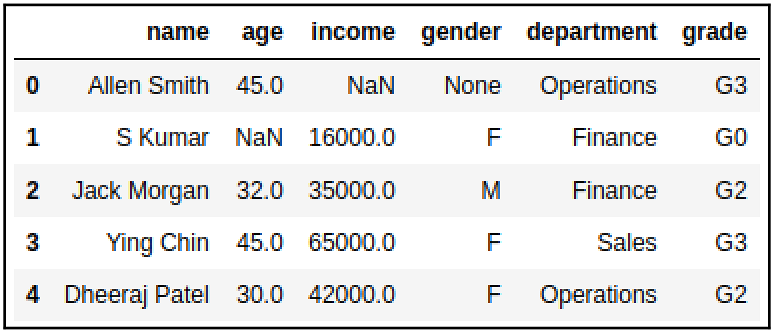
In the preceding code example, we have written multiple DataFrames to a single Excel file. Here, each DataFrame store on a different sheet using the `ExcelWriter` function. Let's see how can we read the JSON files using the `pandas` module in the next section.

Reading and writing data from JSON

JSON (**JavaScript Object Notation**) files are a widely used format for interchanging data among web applications and servers. It acts as a data interchanger and is more readable compared to XML. `pandas` offers the `read_json` function for reading JSON data and `to_json()` for writing JSON data:

```
# Reading JSON file
df=pd.read_json('employee.json')

# display initial 5 records
df.head()
```

This results in the following output:



	name	age	income	gender	department	grade
0	Allen Smith	45.0	NaN	None	Operations	G3
1	S Kumar	NaN	16000.0	F	Finance	G0
2	Jack Morgan	32.0	35000.0	M	Finance	G2
3	Ying Chin	45.0	65000.0	F	Sales	G3
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2

In the preceding code example, we have read the JSON file using the `read_json()` method. Let's see how to write a JSON file:

```
# Writing DataFrame to JSON file
df.to_json('employee_demo.json',orient="columns")
```

In the preceding code example, we have written the JSON file using the `to_json()` method. In the `to_json()` method, the `orient` parameter is used to handle the output string format. `orient` offers record, column, index, and value kind of formats. You can explore it in more detail on the official web page, at `https://pandas.pydata.org/pandas-docs/version/0.24.2/reference/api/pandas.DataFrame.to_json.html`. It's time to jump into HDF5 files. In the next section, we will see how to read and write HDF5 files using the `pandas` module.

Reading and writing data from HDF5

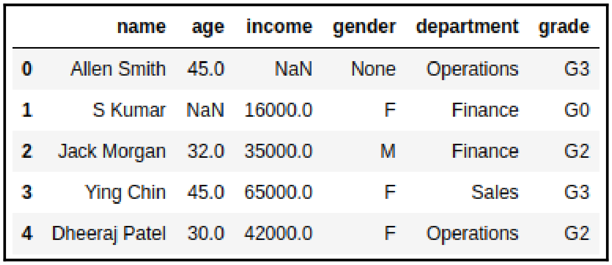
HDF stands for **Hierarchical Data Format**. HDF is designed to store and manage large amounts of data with high performance. It offers fast I/O processing and storage of heterogeneous data. There are various HDF file formats available, such as HDF4 and HDF5. HDF5 is the same as a dictionary object that reads and writes `pandas` DataFrames. It uses the PyTables library's `read_hdf()` function for reading the HDF5 file and the `to_hdf()` function for writing:

```
# Write DataFrame to hdf5
df.to_hdf('employee.h5', 'table', append=True)
```

In the preceding code example, we have written the HDF file format using the `to_hdf()` method. `'table'` is a format parameter used for the table format. Table format may perform slower but offers more flexible operations, such as searching and selecting. The `append` parameter is used to append input data onto the existing data file:

```
# Read a hdf5 file
df=pd.read_hdf('employee.h5', 'table')

# display initial 5 records
df.head()
```

This results in the following output:



	name	age	income	gender	department	grade
0	Allen Smith	45.0	NaN	None	Operations	G3
1	S Kumar	NaN	16000.0	F	Finance	G0
2	Jack Morgan	32.0	35000.0	M	Finance	G2
3	Ying Chin	45.0	65000.0	F	Sales	G3
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2

In the preceding code example, we have read the HDF file format using the `read_hdf()` method. Let's see how to read and write HTML tables from a website in the next section.

Reading and writing data from HTML tables

HTML tables store rows in the `<tr>...</tr>` tag and each row has corresponding `<td>...</td>` cells for holding values. In `pandas`, we can also read the HTML tables from a file or URL. The `read_html()` function reads an HTML table from a file or URL and returns HTML tables into a list of `pandas` DataFrames:

```
# Reading HTML table from given URL
table_url =
'https://en.wikipedia.org/wiki/List_of_sovereign_states_and_dependent_terri
tories_in_North_America'
df_list = pd.read_html(table_url)

print("Number of DataFrames:",len(df_list))
```

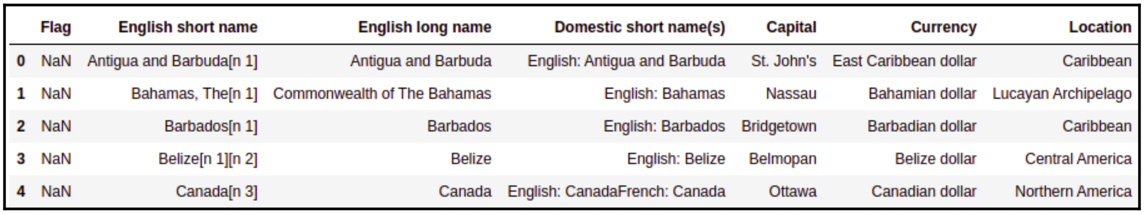
This results in the following output:

```
Number of DataFrames: 7
```

In the preceding code example, we have read the HTML table from a given web page using the `read_html()` method. `read_html()` will return all the tables as a list of DataFrames. Let's check one of the DataFrames from the list:

```
# Check first DataFrame
df_list[0].head()
```

This results in the following output:



	Flag	English short name	English long name	Domestic short name(s)	Capital	Currency	Location
0	NaN	Antigua and Barbuda[n 1]	Antigua and Barbuda	English: Antigua and Barbuda	St. John's	East Caribbean dollar	Caribbean
1	NaN	Bahamas, The[n 1]	Commonwealth of The Bahamas	English: Bahamas	Nassau	Bahamian dollar	Lucayan Archipelago
2	NaN	Barbados[n 1]	Barbados	English: Barbados	Bridgetown	Barbadian dollar	Caribbean
3	NaN	Belize[n 1][n 2]	Belize	English: Belize	Belmopan	Belize dollar	Central America
4	NaN	Canada[n 3]	Canada	English: CanadaFrench: Canada	Ottawa	Canadian dollar	Northern America

In the preceding code example, we have shown the initial five records of the first table available on the given web page. Similarly, we can also write DataFrame objects as HTML tables using `to_html()`. `to_html()` renders the content as an HTML table:

```
# Write DataFrame to raw HTML
df_list[1].to_html('country.html')
```

With the preceding code example, we can convert any DataFrame into an HTML page that contains the DataFrame as a table.

Reading and writing data from Parquet

The Parquet file format provides columnar serialization for `pandas` DataFrames. It reads and writes DataFrames efficiently in terms of storage and performance and shares data across distributed systems without information loss. The Parquet file format does not support duplicate and numeric columns.

There are two engines used to read and write Parquet files in `pandas`: `pyarrow` and the `fastparquet` engine. `pandas`'s default Parquet engine is `pyarrow`; if `pyarrow` is unavailable, then it uses `fastparquet`. In our example, we are using `pyarrow`. Let's install `pyarrow` using `pip`:

```
pip install pyarrow
```

You can also install the `pyarrow` engine in the Jupyter Notebook by putting an `!` before the `pip` keyword. Here is an example:

```
!pip install pyarrow
```

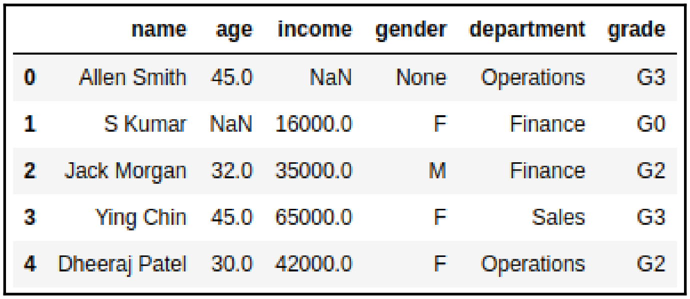
Let's write a file using the `pyarrow` engine:

```
# Write to a parquet file.
df.to_parquet('employee.parquet', engine='pyarrow')
```

In the preceding code example, we have written the `using to_parquet()` Parquet file and the `pyarrow` engine:

```
# Read parquet file
employee_df = pd.read_parquet('employee.parquet', engine='pyarrow')

# display initial 5 records
employee_df.head()
```

This results in the following output:



	name	age	income	gender	department	grade
0	Allen Smith	45.0	NaN	None	Operations	G3
1	S Kumar	NaN	16000.0	F	Finance	G0
2	Jack Morgan	32.0	35000.0	M	Finance	G2
3	Ying Chin	45.0	65000.0	F	Sales	G3
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2

In the preceding code example, we have read the Parquet file using `read_parquet()` and the `pyarrow` engine. `read_parquet()` helps to read the Parquet file formats. Let's see how to read and write the data using `pickle` files in the next section.

Reading and writing data from a pickle pandas object

In the data preparation step, we will use various data structures such as dictionaries, lists, arrays, or DataFrames. Sometimes, we might want to save them for future reference or send them to someone else. Here, a `pickle` object comes into the picture. `pickle` serializes the objects to save them and can be loaded again any time. `pandas` offer two functions: `read_pickle()` for loading `pandas` objects and `to_pickle()` for saving Python objects:

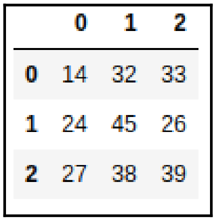
```
# import pandas
import pandas as pd

# Read CSV file
df=pd.read_csv('demo.csv', sep=',' , header=None)

# Save DataFrame object in pickle file
df.to_pickle('demo_obj.pkl')
```

In the preceding code, we read the `demo.csv` file using the `read_csv()` method with `sep` and `header` parameters. Here, we have assigned `sep` with a comma and `header` with `None`. Finally, we have written the dataset to a `pickle` object using the `to_pickle()` method. Let's see how to read `pickle` objects using the `pandas` library:

```
#Read DataFrame object from pickle file
pickle_obj=pd.read_pickle('demo_obj.pkl')
```

```
# display initial 5 records
pickle_obj.head()
```

This results in the following output:



	0	1	2
0	14	32	33
1	24	45	26
2	27	38	39

In the preceding code, we have read the `pickle` objects using the `read_pickle()` method.

Lightweight access with sqllite3

SQLite is an open-source database engine. It offers various features such as faster execution, lightweight processing, serverless architecture, ACID compliance, less administration, high stability, and reliable transactions. It is the most popular and widely deployed database in the mobile and computer world. It is also known as an embedded relational database because it runs as part of your application. SQLite is a lighter database and does not offer full-fledged features. It is mainly used for small data to store and process locally, such as mobile and desktop applications. The main advantages of SQLite are that it is easy to use, efficient, and light, and can be embedded into the application.

We can read and write data in Python from the `sqlite3` module. We don't need to download and install `sqlite3` as it is already available in all the standard Python distributions. With `sqlite3`, we can either store the database in a file or keep it in RAM. `sqlite3` allows us to write any database using SQL without any third-party application server. Let's look at the following example to understand database connectivity:

```
# Import sqlite3
import sqlite3

# Create connection. This will create the connection with employee
database. If the database does not exist it will create the database
conn = sqlite3.connect('employee.db')

# Create cursor
cur = conn.cursor()

# Execute SQL query and create the database table
```

```
cur.execute("create table emp(eid int,salary int)")

# Execute SQL query and Write the data into database
cur.execute("insert into emp values(105, 57000)")

# commit the transaction
con.commit()

# Execute SQL query and Read the data from the database
cur.execute('select * from emp')

# Fetch records
print(cur.fetchall())

# Close the Database connection
conn.close()

Output:
[(105, 57000)]
```

Here, we are using the `sqlite3` module. First, we import the module and create a connection using the `connect()` method. The `connect()` method will take the database name and path; if the database does not exist, it will create the database with the given name and on the given location path. Once you have established a connection with the database, then you need to create the `Cursor` object and execute the SQL query using the `execute()` method. We can create a table in the `execute()` method, as given in the example `emp` table, which is created in the employee database. Similarly, we can write the data using the `execute()` method with an `insert` query argument and commit the data into the database using the `commit()` method. Data can also be extracted using the `execute()` method by passing the `select` query as an argument and fetched using `fetchall()` and the `fetchone()` method. `fetchone()` extracts a single record and `fetchall()` extracts all the records from a database table.

Reading and writing data from MySQL

MySQL is a fast, open-source, and easy-to-use relational or tabular database. It is suitable for small and large business applications. It is very friendly with database-driven web development applications. There are lots of ways to access data in Python from MySQL. Connectors such as MySQLdb, `mysqlconnector`, and `pymysql` are available for MySQL database connectivity. For this connectivity purpose, you should install a MySQL relational database and the `mysql-python` connector. The MySQL setup details are available on its website: `https://www.mysql.com/downloads/`.

You can use the pymysql connector as the client library and it can be installed using pip:

```
pip install pymysql
```

We can establish a connection with the following steps:

1. Import the library.
2. Create a database connection.
3. Create a cursor object.
4. Execute the SQL query.
5. Fetch the records or response for the update or insert the record.
6. Close the connection.

In our examples, we are trying database connectivity using mysqlconnecter and pymysql. Before running the database connectivity script, the first step is to design and create a database and then create a table in MySQL.

Let's create a database using the following query:

```
>> create database employee
```

Change the database to the employee database:

```
>> use employee
```

Create a table in the database:

```
>> create table emp(eid int, salary int);
```

Now we can insert and fetch the records from a table in MySQL. Let's look at the following example to understand the database connectivity:

```
# import pymysql connector module
import pymysql

# Create a connection object using connect() method
connection = pymysql.connect(host='localhost', # IP address of the MySQL
database server
                             user='root', # user name
                             password='root',# password
                             db='emp', # database name
                             charset='utf8mb4', # character set
                             cursorclass=pymysql.cursors.DictCursor) #
cursor type

try:
    with connection.cursor() as cur:
        # Inject a record in database
        sql_query = "INSERT INTO `emp` (`eid`, `salary`) VALUES (%s, %s)"
        cur.execute(sql_query, (104,43000))

    # Commit the record insertion explicitly.
    connection.commit()

    with connection.cursor() as cur:
        # Read records from employee table
        sql_query = "SELECT * FROM `emp`"
        cur.execute(sql_query )
        table_data = cur.fetchall()
        print(table_data)
except:
    print("Exception Occurred")
finally:
    connection.close()
```

Here, we are using the pymysql module. First, we import the module and create a connection. The connect() function will take the host address, which is localhost, in our case (we can also use the IP address of the remote database), username, password, database name, character set, and cursor class.

After establishing the connection, we can read or write the data. In our example, we are writing the data using the insert SQL query and retrieving it using the select query. In the insert query, we are executing the query and passing the argument that we want to enter into the database, and committing the results into the database using the commit() method. When we read the records using the select query, we will get some number of records. We can extract those records using the fetchone() and fetchall() functions. The fetchone() method extracts only single records and the fetchall() method extracts multiple records from a database table.

One more thing; here, all the read-write operations are performed in a try block and the connection is closed in the final block. We can also try one more module mysql.connector for MySQL and Python connectivity. It can be installed using pip:

```
pip install mysql-connector-python
```

Let's look at the following example to understand the database connectivity:

```
# Import the required connector
import mysql.connector
import pandas as pd
```

```
# Establish a database connection to mysql
connection=mysql.connector.connect(user='root',password='root',host='localh
ost',database='emp')

# Create a cursor
cur=connection.cursor()

# Running sql query
cur.execute("select * from emp")

# Fetch all the records and print it one by one
records=cur.fetchall()
for i in records:
    print(i)

# Create a DataFrame from fetched records.
df = pd.DataFrame(records)

# Assign column names to DataFrame
df.columns = [i[0] for i in cur.description]

# close the connection
connection.close()
```

In the preceding code example, we are connecting to Python with the MySQL database using the `mysql.connector` module and the approach and steps for retrieving data are the same as with the `pymysql` module. We are also writing the extracted records into a `pandas` DataFrame by just passing fetched records into the DataFrame object and assigning column names from the cursor description.

Inserting a whole DataFrame into the database

In the preceding program, a single record is inserted using the `insert` command. If we want to insert multiple records, we need to run a loop to insert the multiple records into the database. We can also use the `to_sql()` function to insert multiple records in a single line of code:

```
# Import the sqlalchemy engine
from sqlalchemy import create_engine

# Instantiate engine object
en = create_engine("mysql+pymysql://{user}:{pw}@localhost/{db}"
                .format(user="root",
                        pw="root",
                        db="emp"))
```

```
# Insert the whole dataframe into the database
df.to_sql('emp', con=en, if_exists='append',chunksize=1000, index= False)
```

In the preceding code example, we will create an engine for a database connection with username, password, and database parameters. The `to_sql()` function writes multiple records from the DataFrame to a SQL database. It will take the table name, the `con` parameter for the connection engine object, the `if_exists` parameter for checking whether data will append to a new table or replace with a new table, and `chunksize` for writing data in batches.

Reading and writing data from MongoDB

MongoDB is a document-oriented non-relational (NoSQL) database. It uses JSON-like notation, **BSON** (**Binary Object Notation**) to store the data. MongoDB offers the following features:

- It is a free, open-source, and cross-platform database software.
- It is easy to learn, can build faster applications, supports flexible schemas, handles diverse data types, and has the capability to scale in a distributed environment.
- It works on concepts of documents.
- It has a database, collection, document, field, and primary key.

We can read and write data in Python from MongoDB using the `pymongo` connector. For this connectivity purpose, we need to install MongoDB and the `pymongo` connector. You can download MongoDB from its official web portal: `https://www.mongodb.com/download-center/community`. PyMongo is a pure Python MongoDB client library that can be installed using `pip`:

```
pip install pymongo
```

Let's try database connectivity using `pymongo`:

```
# Import pymongo
import pymongo

# Create mongo client
client = pymongo.MongoClient()

# Get database
db = client.employee
```

```
# Get the collection from database
collection = db.emp

# Write the data using insert_one() method
employee_salary = {"eid":114, "salary":25000}
collection.insert_one(employee_salary)

# Create a dataframe with fetched data
data = pd.DataFrame(list(collection.find()))
```

Here, we are trying to extract data from database collection in MongoDB by creating a Mongo client, inserting data, extracting collection details, and assigning it to the DataFrame. Let's see how to create a database connection with the columnar database Cassandra in the next section.

Reading and writing data from Cassandra

Cassandra is scalable, highly available, durable, and fault-tolerant, has lower admin overhead, has faster read-write, and is a resilient column-oriented database. It is easier to learn and configure. It provides solutions for quite complex problems. It also supports replication across multiple data centers. Plenty of big companies, for example, Apple, eBay, and Netflix use Cassandra.

We can read and write data in Python from Cassandra using the `cassandra-driver` connector. For this connectivity purpose, we need to install Cassandra and `cassandra-driver` connectors. You can download Cassandra from its official website: `http://cassandra.apache.org/download/`. `cassandra-driver` is a pure Python Cassandra client library that can be installed using `pip`:

```
pip install cassandra-driver
```

Let's try database connectivity using `cassandra-driver`:

```
# Import the cluster
from cassandra.cluster import Cluster

# Creating a cluster object
cluster = Cluster()

# Create connections by calling Cluster.connect():
conn = cluster.connect()

# Execute the insert query
conn.execute("""INSERT INTO employee.emp_details (eid, ename, age) VALUES
(%(eid)s, %(ename)s, %(age)s)""", {'eid':101, 'ename': "Steve Smith",
'age': 42})

# Execute the select query
rows = conn.execute('SELECT * FROM employee.emp_details')

# Print the results
for emp_row in rows:
    print(emp_row.eid, emp_row.ename, emp_row.age)

# Create a dataframe with fetched data
data = pd.DataFrame(rows)
```

Here, we are trying to extract data from the Cassandra database by creating a cluster object, creating a connection using the `connect()` method, executing an insert, and selecting query data. After running the query, we are printing the results and assigning the extracted records to the `pandas` DataFrame. Let's move on now to another NoSQL database: Redis.

Reading and writing data from Redis

Redis is an open-source NoSQL database. It is a key-value database, in-memory, extremely fast, and highly available. It can also be employed as a cache or act as a message broker. In-memory means it uses RAM for the storage of data and handles bigger-sized data using virtual memory. Redis offers a cache service or permanent storage. Redis supports a variety of data structures, such as string, set, list, bitmap, geospatial indexes, and hyperlogs. Redis can deal with geospatial, streaming, and time-series data. It is offered with cloud services such as AWS and Google Cloud.

We can read and write data in Python from Redis using the Redis connector. For this connectivity purpose, we need to install Redis and the Redis connector. You can download Redis from the following link: `https://github.com/rgl/redis/downloads`. Redis is a pure Python Redis client library that can be installed using `pip`:

```
pip install redis
```

Let's try database connectivity using Redis:

```
# Import module
import redis

# Create connection
r = redis.Redis(host='localhost', port=6379, db=0)

# Setting key-value pair
```

```
r.set('eid', '101')

# Get value for given key
value=r.get('eid')

# Print the value
print(value)
```

Here, we are trying to extract data from the Redis key-value database. First, we have created a connection with the database. We are setting the key-value pairs into the Redis database using the `set()` method and we have also extracted the value using the `get()` method with the given key parameter.

Finally, its time to shift to the last topic of this chapter, which is PonyORM for **object-relational mapping** (**ORM**).

PonyORM

PonyORM is a powerful ORM package that is written in pure Python. It is fast and easy to use and performs operations with minimum effort. It provides automatic query optimization and a GUI database schema editor. It also supports automatic transaction management, automatic caching, and composite keys. PonyORM uses Python generator expressions, which are translated in SQL. We can install it using `pip`:

```
$ pip install pony
```

Let's see an example of ORM using `pony`:

```
# Import pony module
from pony.orm import *

# Create database
db = Database()

# Define entities
class Emp(db.Entity):
    eid = PrimaryKey(int,auto=True)
    salary = Required(int)

# Check entity definition
show(Emp)

# Bind entities to MySQL database
db.bind('mysql', host='localhost', user='root', passwd='12345',
db='employee')
```

```
# Generate required mappings for entities
db.generate_mapping(create_tables=True)

# turn on the debug mode
sql_debug(True)

# Select the records from Emp entities or emp table
select(e for e in Emp)[:]

# Show the values of all the attribute
select(e for e in Emp)[:].show()

Output:
eid|salary
---+------
104|43000
104|43000
```

In the preceding code example, we are performing ORM. First, we have created a `Database` object and defined entities using an `Emp` class. After that, we have attached the entities to the database using `db.bind()`. We can bind it with four databases: `sqlite`, `mysql`, `postgresql`, and `oracle`. In our example, we are using MySQL and passing its credential details, such as username, password, and database name. We can perform the mapping of entities with data using `generate_mapping()`. The `create_tables=True` argument creates the tables if it does not exist. `sql_debug(True)` will turn on the debug mode. The `select()` function translates a Python generator into a SQL query and returns a `pony` object. This `pony` object will be converted into a list of entities using the slice operator (`[:]`) and the `show()` function will display all the records in a tabular fashion.

Summary

In this chapter, we learned about retrieving, processing, and storing data in different formats. We have looked at reading and writing data from various file formats and sources, such as CSV, Excel, JSON, HDF5, HTML, `pickle`, table, and Parquet files. We also learned how to read and write from various relational and NoSQL databases, such as SQLite3, MySQL, MongoDB, Cassandra, and Redis.

The next chapter, `Chapter 7`, *Cleaning Messy Data*, is about the important topic of data preprocessing and feature engineering with Python. The chapter starts with exploratory data analysis, and leads to filtering, handling missing values, and outliers. After cleaning, the focus will be on data transformation, such as encoding, scaling, and splitting.

7
Cleaning Messy Data

Data analysts and scientists spend most of their time cleaning data and pre-processing messy datasets. While this activity is less talked about, it is one of the most performed activities and one of the most important skills for any data professional. Mastering the skill of data cleaning is necessary for any aspiring data scientist. Data cleaning and pre-processing is the process of identifying, updating, and removing corrupt or incorrect data. Cleaning and pre-processing results in high-quality data for robust and error-free analysis. Quality data can beat complex algorithms and outperform simple and less complex algorithms. In this context, high quality means accurate, complete, and consistent data. Data cleaning is a set of activities such as handling missing values, removing outliers, feature encoding, scaling, transformation, and splitting.

This chapter focuses on data cleaning, manipulation, and wrangling. Data preparation, manipulation, wrangling, and munging are all terms for the same thing, and the main objective is to clean up the data in order to get valuable insights. We will start by exploring employee data and then start filtering the data and handling missing values and outliers. After cleaning, we will focus on performing data transformation activities such as encoding, scaling, and splitting. We will mostly be using `pandas` and `scikit-learn` in this chapter.

In this chapter, we will cover the following topics:

- Exploring data
- Filtering data to weed out the noise
- Handling missing values
- Handling outliers
- Feature encoding techniques
- Feature scaling
- Feature transformation
- Feature splitting

Let's get started!

Technical requirements

The following are the technical requirements for this chapter:

- You can find the code and the datasets that will be used in this chapter in this book's GitHub repository at `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter07`.
- All the code is available in the `ch7.ipynb` file.
- This chapter uses only one CSV file (`employee.csv`) for practice purposes.
- In this chapter, we will use the `pandas` and `scikit-learn` Python libraries, so please ensure you have them installed.

Exploring data

In this section, we will explore data by performing **Exploratory Data Analysis** (**EDA**). EDA is the most critical and most important component of the data analysis process. EDA offers the following benefits:

- It provides an initial glimpse of data and its context.
- It captures quick insights and identifies the potential drivers from the data for predictive analysis. It finds the queries and questions that can be answered for decision-making purposes.
- It assesses the quality of the data and helps us build the road map for data cleaning and preprocessing.
- It finds missing values, outliers, and the importance of features for analysis.
- EDA uses descriptive statistics and visualization techniques to explore data.

In EDA, the first step is to read the dataset. We can read the dataset using `pandas`. The `pandas` library offers various options for reading data. It can read files in various formats, such as CSV, Excel, JSON, parquet, HTML, and pickle. All these methods were covered in the previous chapter. After reading the data, we can explore the data. This initial exploration will help us understand the data and gain some domain insights. Let's start with the EDA process.

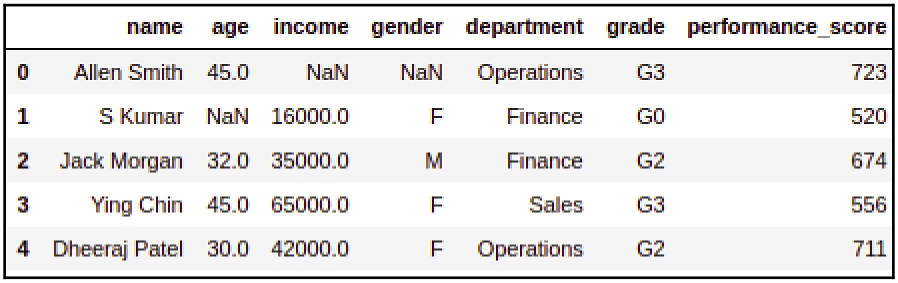
First, we will read the `employee.csv` file (you can find this file in the `Chapter-7` folder of this book's GitHub repository at `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/blob/master/Chapter07/employee.csv`):

```
# import pandas
import pandas as pd

# Read the data using csv
data=pd.read_csv('employee.csv')
```

Let's take a look at the first five records in the file using the `head()` method:

```
# See initial 5 records
data.head()
```

This results in the following output:

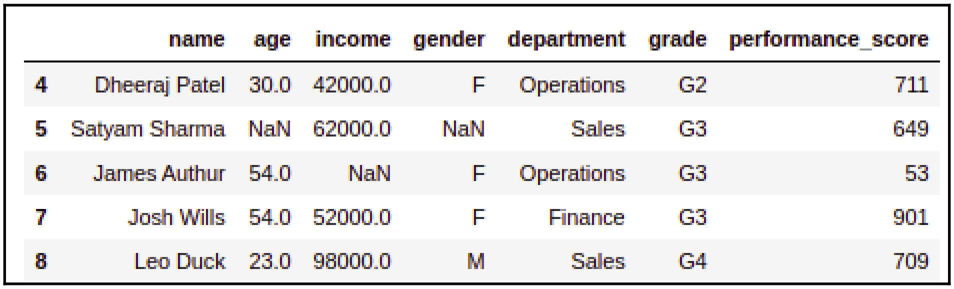


	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711

Similarly, let's look at the last five records using the `head()` method:

```
# See last 5 records
data.tail()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649
6	James Authur	54.0	NaN	F	Operations	G3	53
7	Josh Wills	54.0	52000.0	F	Finance	G3	901
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

We can check the list of columns using the `columns` attribute:

```
# Print list of columns in the data
print(data.columns)
```

This results in the following output:

```
Index(['name', 'age', 'income', 'gender', 'department', 'grade',
       'performance_score'], dtype='object')
```

Let's check out the list of columns by using the shape of the DataFrame by using the `shape` attribute:

```
# Print the shape of a DataFrame
print(data.shape)
```

This results in the following output:

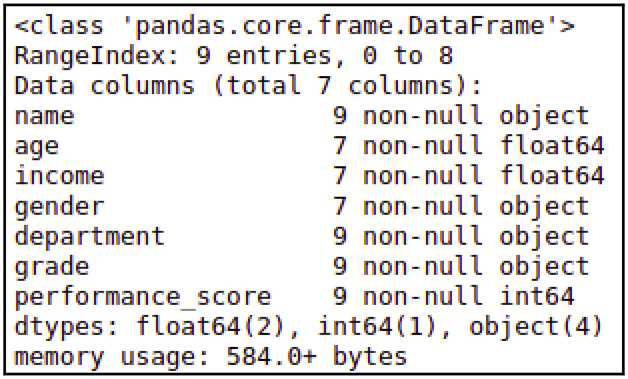
```
(9, 7)
```

As we can see, the dataset has `9` rows and `7` columns.

We can check the table schema, its columns, rows, data types, and missing values in the DataFrame by using the following code:

```
# Check the information of DataFrame
data.info()
```

This results in the following output:



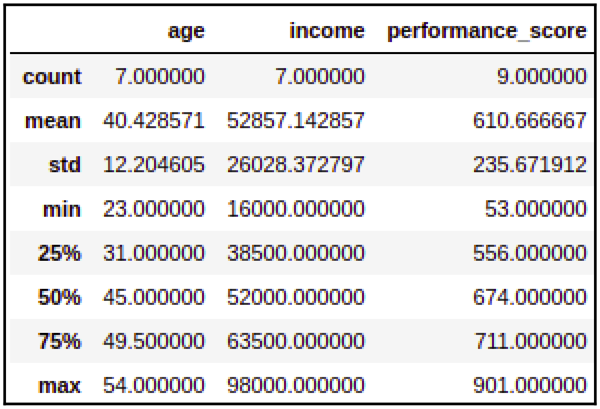
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9 entries, 0 to 8
Data columns (total 7 columns):
name                 9 non-null object
age                  7 non-null float64
income               7 non-null float64
gender               7 non-null object
department           9 non-null object
grade                9 non-null object
performance_score    9 non-null int64
dtypes: float64(2), int64(1), object(4)
memory usage: 584.0+ bytes
```

In the preceding output, you can see that there are 7 columns in the data. Out of these 7 columns, 3 columns (age, income, and gender) have missing values. Out of these 7 columns, 4 are objects, 2 are floats, and 1 is an integer.

Now, let's take a look at the descriptive statistics of the data by using the `describe` function. This function will describe numerical objects. In our example, the age, income, and performance scores will describe the count, mean, standard deviation, min-max, and the first, second, and third quartiles:

```
# Check the descriptive statistics
data.describe()
```

This results in the following output:



	age	income	performance_score
count	7.000000	7.000000	9.000000
mean	40.428571	52857.142857	610.666667
std	12.204605	26028.372797	235.671912
min	23.000000	16000.000000	53.000000
25%	31.000000	38500.000000	556.000000
50%	45.000000	52000.000000	674.000000
75%	49.500000	63500.000000	711.000000
max	54.000000	98000.000000	901.000000

In the preceding code block, we have checked the descriptive statistics values of the data using the `describe()` function. From these results, we can interpret that the employee's age is ranging from 23 to 54 years. Here, the mean age is 40 years and the median age is 45 years. Similarly, we can draw conclusions for income and performance scores. Now that we've described the data, let's learn how to filter noise from data.

Filtering data to weed out the noise

In the last two decades, the data size of companies and government agencies has increased due to digitalization. This also caused an increase in consistency, errors, and missing values. Data filtering is responsible for handling such issues and optimizing them for management, reporting, and predictions. The filtering process boosts the accuracy, relevance, completeness, consistency, and quality of the data by processing dirty, messy, or coarse datasets. It is a very crucial step for any kind of data management because it can make or break a competitive edge of business. Data scientists need to master the skill of data filtering. Different kinds of data need different kinds of treatment. That's why a systematic approach to data filtering needs to be taken.

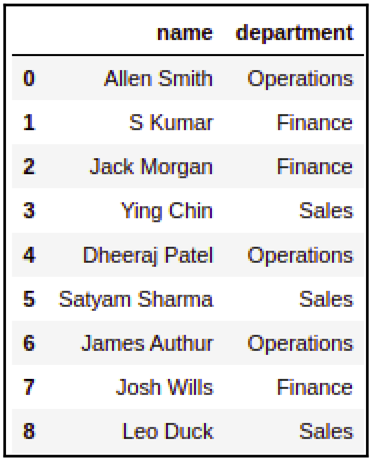
In the previous section, we learned about data exploration, while in this section, we will learn about data filtering. Data can be filtered either column-wise or row-wise. Let's explore them one by one.

Column-wise filtration

In this subsection, we will learnhow to filter column-wise data. We can filter columns using the `filter()` method. The `slicing []. filter()` method selects the columns when they're passed as a list of columns. Take a look at the following example:

```
# Filter columns
data.filter(['name', 'department'])
```

This results in the following output:



	name	department
0	Allen Smith	Operations
1	S Kumar	Finance
2	Jack Morgan	Finance
3	Ying Chin	Sales
4	Dheeraj Patel	Operations
5	Satyam Sharma	Sales
6	James Authur	Operations
7	Josh Wills	Finance
8	Leo Duck	Sales

Similarly, we can also filter columns using slicing. In slicing, a single column does not need a list, but when we are filtering multiple columns, then they should be on the list. The output of a single column is a pandas Series. If we want the output as a DataFrame, then we need to put the name of the single column into a list. Take a look at the following example:

```
# Filter column "name"
data['name']

0       Allen Smith
1           S Kumar
2       Jack Morgan
3         Ying Chin
4     Dheeraj Patel
```

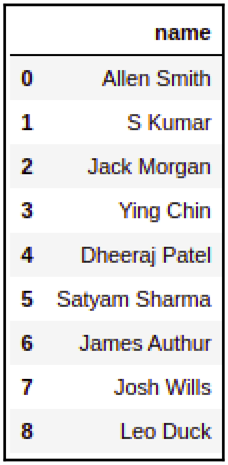
```
5     Satyam Sharma
6      James Authur
7        Josh Wills
8          Leo Duck
Name: name, dtype: object
```

In the preceding example, we have selected a single column without passing it into the list and the output is a pandas Series.

Now, let's select a single column using a Python list:

```
# Filter column "name"
data[['name']]
```

This results in the following output:



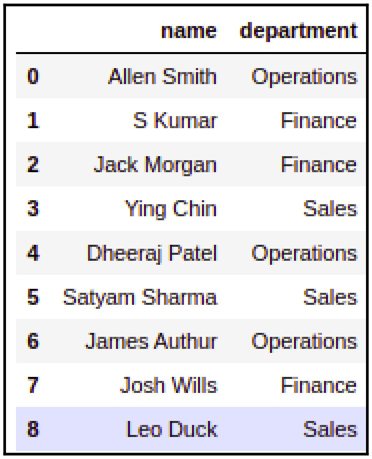
	name
0	Allen Smith
1	S Kumar
2	Jack Morgan
3	Ying Chin
4	Dheeraj Patel
5	Satyam Sharma
6	James Authur
7	Josh Wills
8	Leo Duck

As you can see, a single column can be selected using a Python list. The output of this filter is a pandas DataFrame with a single column.

Now, let's filter multiple columns from the pandas DataFrame:

```
# Filter two columns: name and department
data[['name','department']]
```

This results in the following output:



	name	department
0	Allen Smith	Operations
1	S Kumar	Finance
2	Jack Morgan	Finance
3	Ying Chin	Sales
4	Dheeraj Patel	Operations
5	Satyam Sharma	Sales
6	James Authur	Operations
7	Josh Wills	Finance
8	Leo Duck	Sales

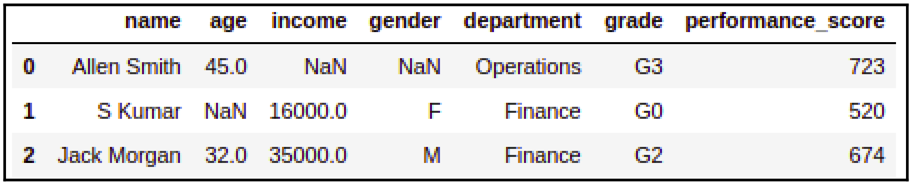
As you can see, we have filtered the two columns without using the `filter()` function.

Row-wise filtration

Now, let's filter row-wise data. We can filter data using indices, slices, and conditions. In indices, you have to pass the index of the record, while for slicing, we need to pass the slicing range. Take a look at the following example:

```
# Select rows for the specific index
data.filter([0,1,2],axis=0)
```

This results in the following output:



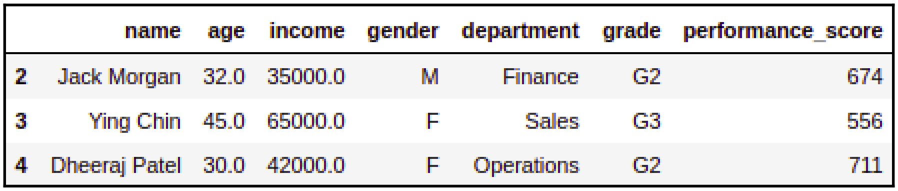
	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674

In the preceding example, we have filtered the data based on indexes.

The following is an example of filtering data by slicing:

```
# Filter data using slicing
data[2:5]
```

This results in the following output:

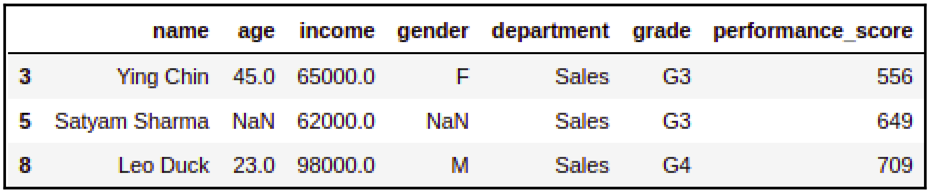


	name	age	income	gender	department	grade	performance_score
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711

In condition-based filtration, we have to pass some conditions in square brackets, `[ ]`, or brackets, `( )`. For a single value, we use the == (double equal to) condition, while for multiple values, we use the `isin()` function and pass the list of values. Let's take a look at the following example:

```
# Filter data for specific value
data[data.department=='Sales']
```

This results in the following output:

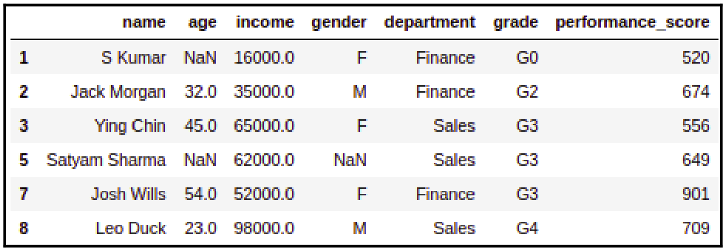


	name	age	income	gender	department	grade	performance_score
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

In the preceding code, we filtered the department sales in the first line of code using == (double equal to) as a condition. Now, let's filter multiple columns using the `isin()` function:

```
# Select data for multiple values
data[data.department.isin(['Sales','Finance'])]
```

This results in the following output:



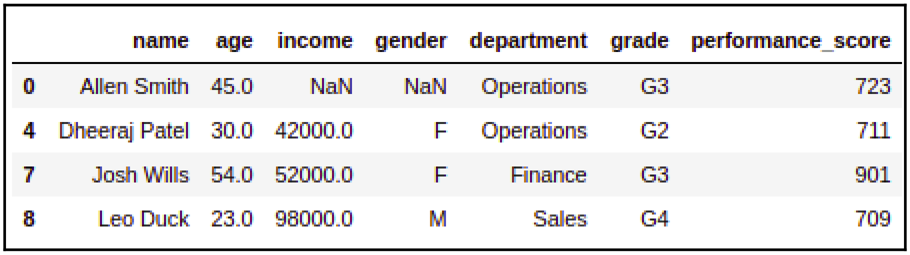
	name	age	income	gender	department	grade	performance_score
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649
7	Josh Wills	54.0	52000.0	F	Finance	G3	901
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

In the preceding example, we filtered the department sales and finance department using the `isin()` function.

Now, let's look at the >= and <= conditions for continuous variables. We can have single or multiple conditions. Let's take a look at the following example:

```
# Filter employee who has more than 700 performance score
data[(data.performance_score >=700)]
```

This results in the following output:

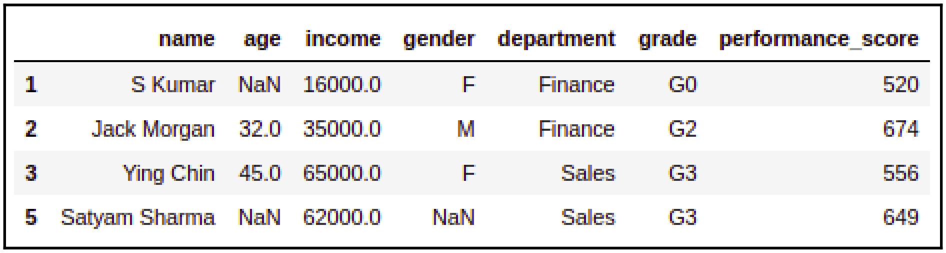


	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711
7	Josh Wills	54.0	52000.0	F	Finance	G3	901
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

In the preceding example, we filtered employees on the basis of their performance score (performance_score >=700). Now, let's filter data using multiple conditions:

```
# Filter employee who has more than 500 and less than 700 performance score
data[(data.performance_score >=500) & (data.performance_score < 700)]
```

This results in the following output:

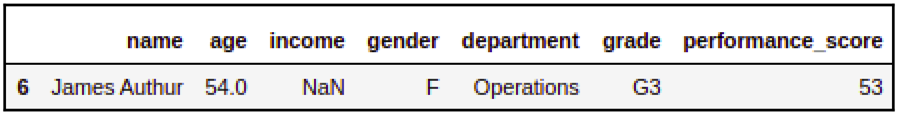


	name	age	income	gender	department	grade	performance_score
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649

We can also try the `query()` method. This method queries the columns using a boolean expression. Let's look at an example:

```
# Filter employee who has performance score of less than 500
data.query('performance_score<500')
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score
6	James Authur	54.0	NaN	F	Operations	G3	53

In the preceding example, we filtered the employees who have performance scores less than 500. Now, let's learn how to handle missing values.

Handling missing values

Missing values are the values that are absent from the data. Absent values can occur due to human error, privacy concerns, or the value not being filled in by the respondent filling in the survey. This is the most common problem in data science and the first step of data preprocessing. Missing values affect a machine learning model's performance. Missing values can be handled in the following ways:

- Drop the missing value records.
- Fill in the missing value manually.
- Fill in the missing values using the measures of central tendency, such as mean, median, and mode. The mean is used to impute the numeric feature, the median is used to impute the ordinal feature, and the mode or highest occurring value is used to impute the categorical feature.
- Fill in the most probable value using machine learning models such as regression, decision trees, KNNs.

It is important to understand that in some cases, missing values will not impact the data. For example, driving license numbers, social security numbers, or any other unique identification numbers will not impact the machine learning models because they can't be used as features in the model.

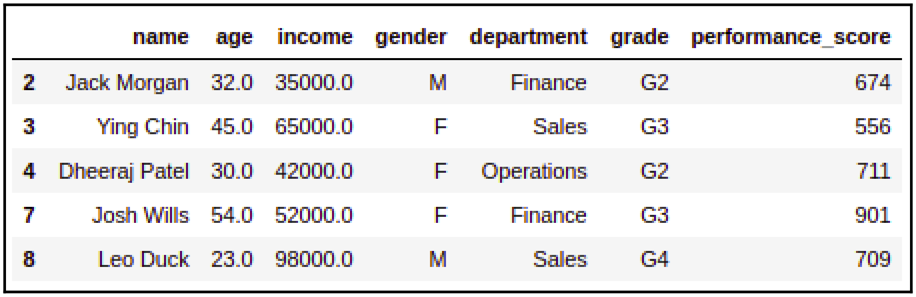
In the following subsections, we will look at how missing values can be handled in more detail. First, we'll learn how to drop missing values.

Dropping missing values

In Python, missing values can be dropped using the `dropna()` function. `dropna` takes one argument: `how`. `how` can take two values: `all` or `any`. `any` drops certain rows that contain NAN or missing values, while `all` drops all the rows contains NAN or missing values:

```
# Drop missing value rows using dropna() function
# Read the data

data=pd.read_csv('employee.csv')
data=data.dropna()
data
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711
7	Josh Wills	54.0	52000.0	F	Finance	G3	901
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

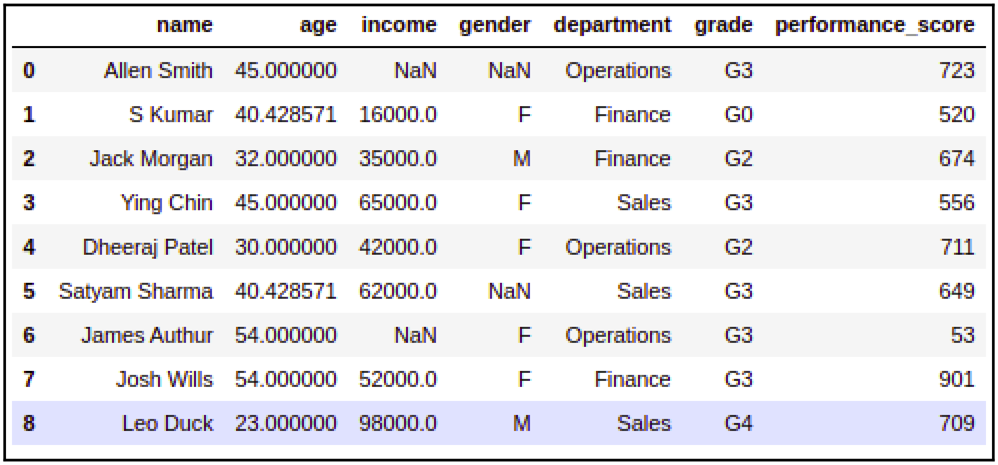
This summarizes the dataset as a dataframe.

Filling in a missing value

In Python, missing values can be dropped using the `fillna()` function. The `fillna()` function takes one value that we want to fill at the missing place. We can fill in the missing values using the mean, median, and mode:

```
# Read the data
data=pd.read_csv('employee.csv')

# Fill all the missing values in the age column with mean of the age column
data['age']=data.age.fillna(data.age.mean())
data
```

This results in the following output:

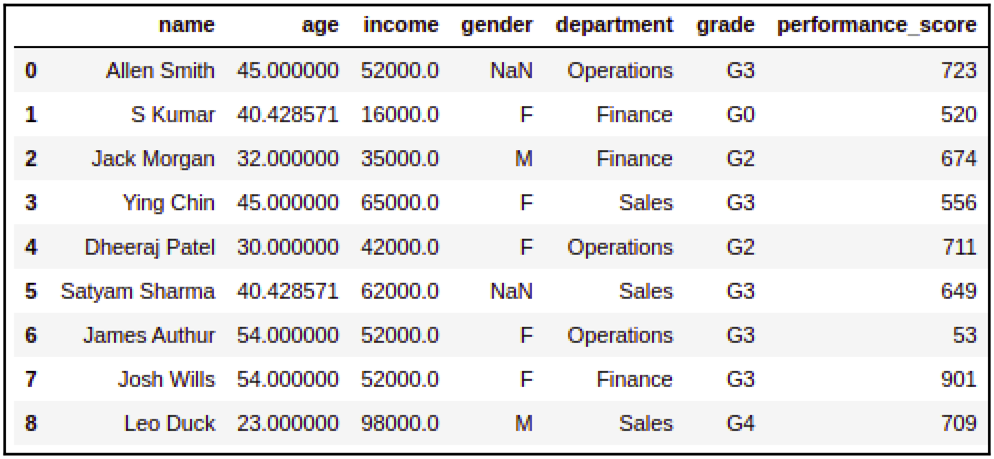


	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.000000	NaN	NaN	Operations	G3	723
1	S Kumar	40.428571	16000.0	F	Finance	G0	520
2	Jack Morgan	32.000000	35000.0	M	Finance	G2	674
3	Ying Chin	45.000000	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.000000	42000.0	F	Operations	G2	711
5	Satyam Sharma	40.428571	62000.0	NaN	Sales	G3	649
6	James Authur	54.000000	NaN	F	Operations	G3	53
7	Josh Wills	54.000000	52000.0	F	Finance	G3	901
8	Leo Duck	23.000000	98000.0	M	Sales	G4	709

In the preceding example, the missing values in the age column have been filled in with the mean value of the age column. Let's learn how to fill in the missing values using the median:

```
# Fill all the missing values in the income column with a median of the
income column
data['income']=data.income.fillna(data.income.median())
data
```

This results in the following output:

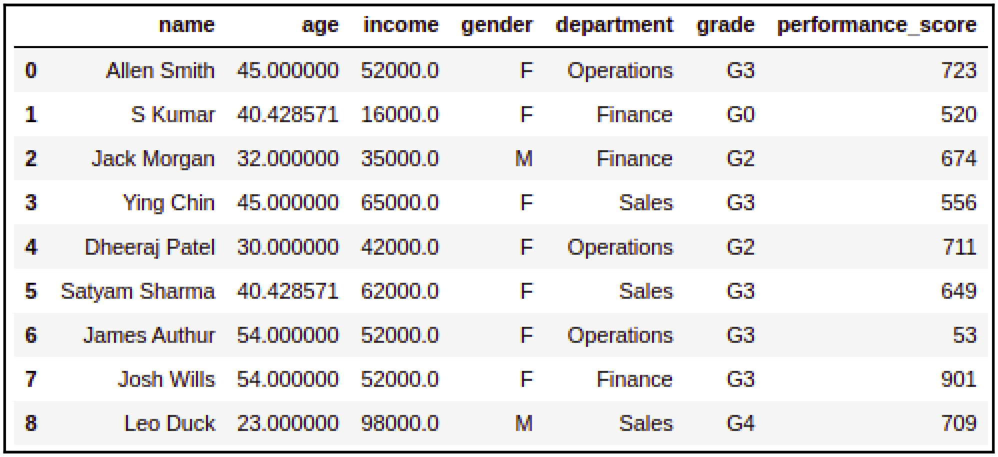


	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.000000	52000.0	NaN	Operations	G3	723
1	S Kumar	40.428571	16000.0	F	Finance	G0	520
2	Jack Morgan	32.000000	35000.0	M	Finance	G2	674
3	Ying Chin	45.000000	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.000000	42000.0	F	Operations	G2	711
5	Satyam Sharma	40.428571	62000.0	NaN	Sales	G3	649
6	James Authur	54.000000	52000.0	F	Operations	G3	53
7	Josh Wills	54.000000	52000.0	F	Finance	G3	901
8	Leo Duck	23.000000	98000.0	M	Sales	G4	709

In the preceding example, the missing values in the income column have been filled in with the median value of the income column. Let's learn how to fill in missing values using the mode:

```
# Fill all the missing values in the gender column(category column) with
the mode of the gender column
data['gender']=data['gender'].fillna(data['gender'].mode()[0])
data
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.000000	52000.0	F	Operations	G3	723
1	S Kumar	40.428571	16000.0	F	Finance	G0	520
2	Jack Morgan	32.000000	35000.0	M	Finance	G2	674
3	Ying Chin	45.000000	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.000000	42000.0	F	Operations	G2	711
5	Satyam Sharma	40.428571	62000.0	F	Sales	G3	649
6	James Authur	54.000000	52000.0	F	Operations	G3	53
7	Josh Wills	54.000000	52000.0	F	Finance	G3	901
8	Leo Duck	23.000000	98000.0	M	Sales	G4	709

In the preceding code example, the missing values in the gender column have been filled in with the mode value of the gender column. As you have seen, the mean, median, and mode help us handle missing values in pandas DataFrames. In the next section, we will focus on how to handle outliers.

Handling outliers

Outliers are those data points that are distant from most of the similar points – in other words, we can say the outliers are entities that are different from the crowd. Outliers cause problems when it comes to building predictive models, such as long model training times, poor accuracy, an increase in error variance, a decrease in normality, and a reduction in the power of statistical tests.

There are two types of outliers: univariate and multivariate. Univariate outliers can be found in single variable distributions, while multivariates can be found in n-dimensional spaces. We can detect and handle outliers in the following ways:

- **Box Plot**: We can use a box plot to create a bunch of data points through quartiles. It groups the data points between the first and third quartile into a rectangular box. The box plot also displays the outliers as individual points using the interquartile range.
- **Scatter Plot**: A scatter plot displays the points (or two variables) on the two-dimensional chart. One variable is placed on the x-axis, while the other is placed on the y-axis.
- **Z-Score**: The Z-score is a kind of parametric approach to detecting outliers. It assumes a normal distribution of the data. The outlier lies in the tail of the normal curve distribution and is far from the mean:

$$z = \frac{x-\mu}{\sigma}$$

- **Interquartile Range (IQR)**: IQR is a robust statistical measure of data dispersion. It is the difference between the third and first quartile. These quartiles can be visualized in a box plot. This is also known as the midspread, the middle 50%, or H-spread:

$$IQR = Q3 - Q1$$

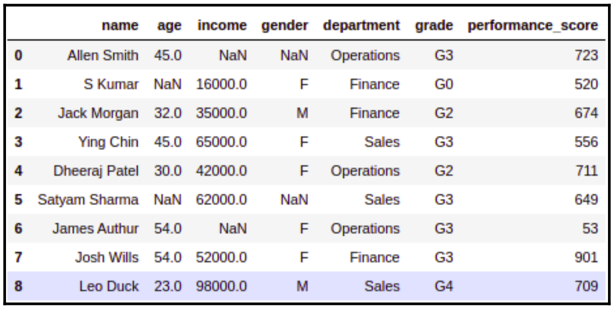
- **Percentile**: A percentile is a statistical measure that divides data into 100 groups of equal size. Its value indicates the percentage of the population below that value. For example, the 95th percentile means 95% of people fall under this category.

Let's drop some outliers using standard deviation and the mean:

```
# Dropping the outliers using Standard Deviation
# Read the data
data=pd.read_csv('employee.csv')

# Dropping the outliers using Standard Deviation
upper_limit= data['performance_score'].mean () + 3 *
data['performance_score'].std ()
lower_limit = data['performance_score'].mean () - 3 *
```

```
data['performance_score'].std ()
data = data[(data['performance_score'] < upper_limit) &
(data['performance_score'] > lower_limit)]
data
```

This results in the following output:

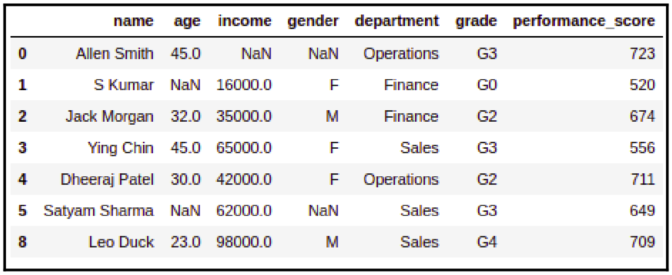


	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649
6	James Authur	54.0	NaN	F	Operations	G3	53
7	Josh Wills	54.0	52000.0	F	Finance	G3	901
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

In the preceding example, we are handling the outliers using standard deviation and the mean. We are using $mean - 3 * standard\ deviation$ as the upper limit and $mean + 3 * standard\ deviation$ as the lower limit for filtering the outliers. We can also try the percentile values to remove the outliers. Let's take a look at the following example:

```
# Read the data
data=pd.read_csv('employee.csv')

# Drop the outlier observations using Percentiles
upper_limit = data['performance_score'].quantile(.99)
lower_limit = data['performance_score'].quantile(.01)
data = data[(data['performance_score'] < upper_limit) &
(data['performance_score'] > lower_limit)]
data
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723
1	S Kumar	NaN	16000.0	F	Finance	G0	520
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674
3	Ying Chin	45.0	65000.0	F	Sales	G3	556
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711
5	Satyam Sharma	NaN	62000.0	NaN	Sales	G3	649
8	Leo Duck	23.0	98000.0	M	Sales	G4	709

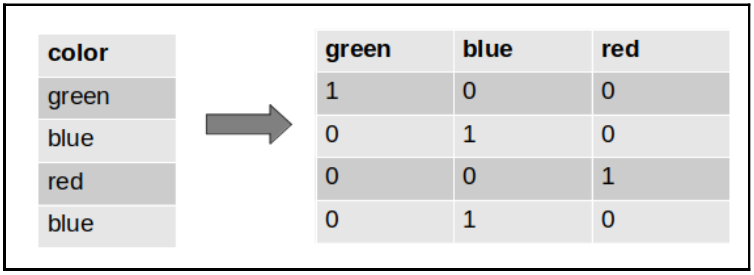
In the preceding code example, we handled the outliers using percentiles. We removed the outliers by using a percentile of 1 for the lower limit and by using a percentile of 99 for the upper limit. This helps us handle outliers in pandas DataFrames. In the next section, we will focus on how to perform feature encoding.

Feature encoding techniques

Machine learning models are mathematical models that required numeric and integer values for computation. Such models can't work on categorical features. That's why we often need to convert categorical features into numerical ones. Machine learning model performance is affected by what encoding technique we use. Categorical values range from 0 to N-1 categories.

One-hot encoding

One-hot encoding transforms the categorical column into labels and splits the column into multiple columns. The numbers are replaced by binary values such as 1s or 0s. For example, let's say that, in the color variable, there are three categories; that is, red, green, and blue. These three categories are labeled and encoded into binary columns, as shown in the following diagram:



color
green
blue
red
blue

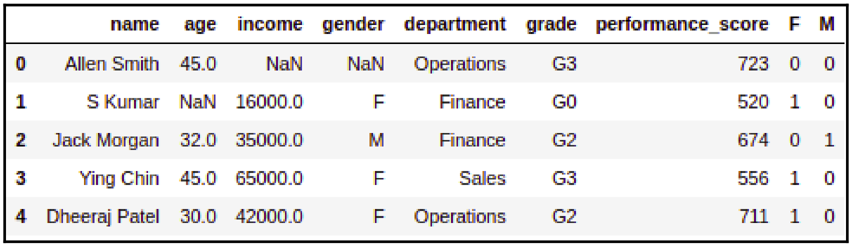
green	blue	red
1	0	0
0	1	0
0	0	1
0	1	0

One-hot encoding can also be performed using the get_dummies() function. Let's use the get_dummies() function as an example:

```
# Read the data
data=pd.read_csv('employee.csv')
# Dummy encoding
encoded_data = pd.get_dummies(data['gender'])

# Join the encoded _data with original dataframe
data = data.join(encoded_data)
```

```
# Check the top-5 records of the dataframe
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	F	M
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	0	0
1	S Kumar	NaN	16000.0	F	Finance	G0	520	1	0
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	0	1
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	1	0
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	1	0

Here, we can see two extra columns, F and M. Both columns are dummy columns that were added by the Boolean encoder. We can also perform the same task with `OneHotEncoder` from the `scikit-learn` module. Let's look at an example of using `OneHotEncoder`.

```
# Import one hot encoder
from sklearn.preprocessing import OneHotEncoder

# Initialize the one-hot encoder object
onehotencoder = OneHotEncoder()

# Fill all the missing values in income column(category column) with mode
of age column
data['gender']=data['gender'].fillna(data['gender'].mode()[0])

# Fit and transforms the gender column
onehotencoder.fit_transform(data[['gender']]).toarray()
```

This results in the following output:

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

In the preceding code example, we imported `OneHotEncoder`, initialized its object, and then fit and transformed the model on the gender column. We can see that the output array has two columns for female and male employees.

Label encoding

Label encoding is also known as integer encoding. Integer encoding replaces categorical values with numeric values. Here, the unique values in variables are replaced with a sequence of integer values. For example, let's say there are three categories: red, green, and blue. These three categories were encoded with integer values; that is, `red` is 0, `green` is 1, and `blue` is 2.

Let's take a look at the following label encoding example:

```
# Import pandas
import pandas as pd

# Read the data
data=pd.read_csv('employee.csv')

# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Instantiate the Label Encoder Object
label_encoder = LabelEncoder()

# Fit and transform the column
encoded_data = label_encoder.fit_transform(data['department'])

# Print the encoded
print(encoded_data)
```

This results in the following output:

```
[2 1 0 0 2 1 2 1 0 2]
```

In the preceding example, we performed simple label encoding.

In the following example, we are encoding the department column using the `LabelEncoder` class. First, we must import and initialize the `LabelEncoder` object and then fit and transform the column that we want to encode. Let's perform the inverse transformation on the encoded labels:

```
# Perform inverse encoding
inverse_encode=label_encoder.inverse_transform([0, 0, 1, 2])

# Print inverse encode
print(inverse_encode)
```

This results in the following output:

```
['Finance' 'Finance' 'Operations' 'Sales']
```

In the preceding example, we reversed the encoding of the encoded values using `inverse_transformation()`. We can also use one-hot encoding with numerical variables. Here, each unique numeric value is encoded into an equivalent binary variable.

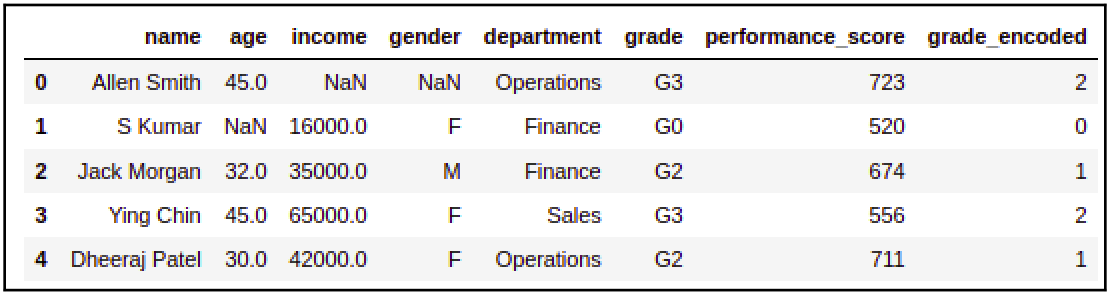
Ordinal encoder

Ordinal encoding is similar to label encoding, except there's an order to the encoding. The output encoding will start from 0 and end at one less than the size of the categories. Let's look at an example containing employee grades such as G0, G1, G2, G3, and G4. These five grades have been encoded with ordinal integer values; that is, `G0` is 0, `G1` is 1, `G2` is 2, `G3` is 3, and `G4` is 4. We can define the order of the values as a list and pass it to the category parameter. The ordinal encoder uses the integer or numeric values to encode. Here, the integer and numeric values are ordinal in nature. This encoding helps machine learning algorithms take advantage of this ordinal relationship.

Let's take a look at the following `OrdinalEncoder` example:

```
# Import pandas and OrdinalEncoder
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

# Load the data
data=pd.read_csv('employee.csv')

# Initialize OrdinalEncoder with order
order_encoder=OrdinalEncoder(categories=['G0','G1','G2','G3','G4'])

# fit and transform the grade
data['grade_encoded'] = label_encoder.fit_transform(data['grade'])

# Check top-5 records of the dataframe
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	grade_encoded
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	2
1	S Kumar	NaN	16000.0	F	Finance	G0	520	0
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	1
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	2
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	1

The preceding example is similar to the `LabelEncoder` example, except for the order of the values that were passed when the `OrdinalEncoder` object was initialized. In this example, the `categories` parameters were passed alongside the `grade` order at the time of initialization.

Feature scaling

In real life, most features have different ranges, magnitudes, and units, such as age being between 0-200 and salary being between 0 to thousands or millions. From a data analyst or data scientist's point of view, how can we compare these features when they are on different scales? High-magnitude features will weigh more on machine learning models than lower magnitude features. Thankfully, feature scaling or feature normalization can solve such issues.

Feature scaling brings all the features to the same level of magnitude. This is not compulsory for all kinds of algorithms; some algorithms clearly need scaled data, such as those that rely on Euclidean distance measures such as K-nearest neighbor and the K-means clustering algorithm.

Methods for feature scaling

Now, let's look at the various methods we can use for feature scaling:

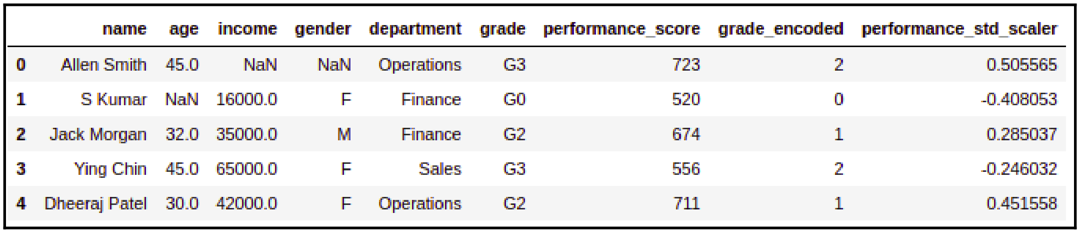
- **Standard Scaling or Z-Score Normalization**: This method computes the scaled values of a feature by using the mean and standard deviation of that feature. It is best suited for normally distributed data. Suppose μ is the mean and σ is the standard deviation of the feature column. This results in the following formula:

$$z_i = \frac{x_i - \mu}{\sigma}$$

Let's take a look at the following standard scaling example:

```
# Import StandardScaler(or z-score normalization)
from sklearn.preprocessing import StandardScaler

# Initialize the StandardScaler
scaler = StandardScaler()

# To scale data
scaler.fit(data['performance_score'].values.reshape(-1,1))
data['performance_std_scaler']=scaler.transform(data['performance_s
core'].values.reshape(-1,1))
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	grade_encoded	performance_std_scaler
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	2	0.505565
1	S Kumar	NaN	16000.0	F	Finance	G0	520	0	-0.408053
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	1	0.285037
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	2	-0.246032
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	1	0.451558

Here, we need to import and initialize the `StandardScaler` object. After initialization, we must perform fit and transform operations on the column that we want to scale.

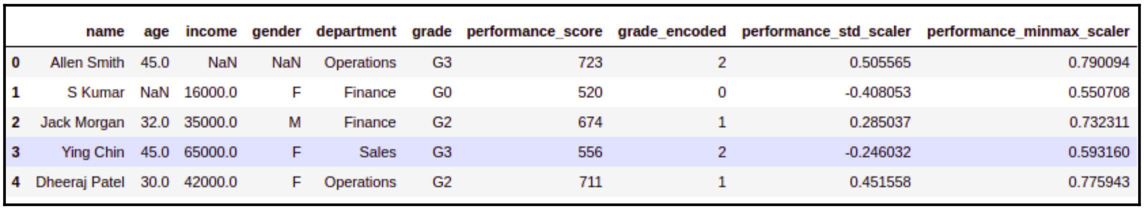
- **Min-Max Scaling**: This method linearly transforms the original data into the given range. It preserves the relationships between the scaled data and the original data. If the distribution is not normally distributed and the value of the standard deviation is very small, then the min-max scaler works better since it is more sensitive to outliers. Let's say that min_x is the minimum value and $max_x$is the maximum value of a feature column, while new_min_x and new_max_x are the new minimum and new maximum. This results in the following formula:

$$x'_i = \frac{x_i - min_x}{max_x - min_x}(new\_max_x - new\_min_x) + new\_min_x$$

Let's take a look at the following min-max scaling example:

```
# Import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

# Initialise the MinMaxScaler
scaler = MinMaxScaler()

# To scale data
scaler.fit(data['performance_score'].values.reshape(-1,1))
data['performance_minmax_scaler']=scaler.transform(data['performanc
e_score'].values.reshape(-1,1))
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	grade_encoded	performance_std_scaler	performance_minmax_scaler
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	2	0.505565	0.790094
1	S Kumar	NaN	16000.0	F	Finance	G0	520	0	-0.408053	0.550708
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	1	0.285037	0.732311
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	2	-0.246032	0.593160
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	1	0.451558	0.775943

Here, we need to import and initialize the `MinMaxScaler` object. After initialization, we, must perform the fit and transform operations on the column that we want to scale.

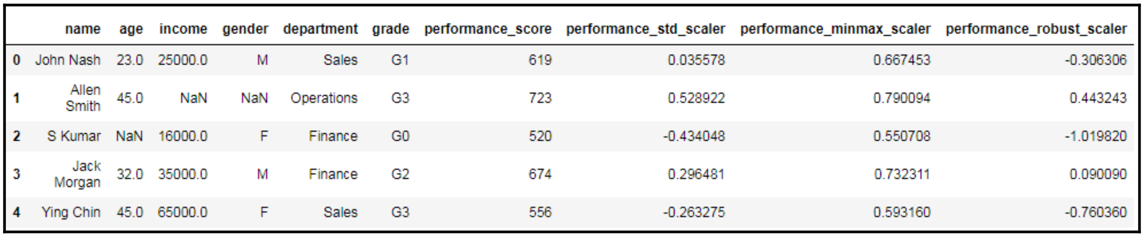
- **Robust Scaling**: This method is similar to the min-max scaler method. Instead of min-max, this method uses an interquartile range. That's why it is robust to outliers. Suppose $Q1_x$and $Q3_x$ are the first and third quartiles of column x. This results in the following formula:

$$x'_i = \frac{x_i - Q1_x}{Q3_x \ -Q1_x}$$

Let's take a look at the following robust scaling example:

```
# Import RobustScaler
from sklearn.preprocessing import RobustScaler

# Initialise the RobustScaler
scaler = RobustScaler()

# To scale data
scaler.fit(data['performance_score'].values.reshape(-1,1))
data['performance_robust_scaler']=scaler.transform(data['performanc
e_score'].values.reshape(-1,1))

# See initial 5 records
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	performance_std_scaler	performance_minmax_scaler	performance_robust_scaler
0	John Nash	23.0	25000.0	M	Sales	G1	619	0.035578	0.667453	-0.306306
1	Allen Smith	45.0	NaN	NaN	Operations	G3	723	0.528922	0.790094	0.443243
2	S Kumar	NaN	16000.0	F	Finance	G0	520	-0.434048	0.550708	-1.019820
3	Jack Morgan	32.0	35000.0	M	Finance	G2	674	0.296481	0.732311	0.090090
4	Ying Chin	45.0	65000.0	F	Sales	G3	556	-0.263275	0.593160	-0.760360

Here, we need to import and initialize the `RobustScaler` object. After initialization, we must fit and transform the column that we want to scale.

Feature transformation

Feature transformation alters features so that they're in the required form. It also reduces the effect of outliers, handles skewed data, and makes the model more robust. The following list shows the different kinds of feature transformation:

- Log transformation is the most common mathematical transformation used to transform skewed data into a normal distribution. Before applying the log transform, ensure that all the data values only contain positive values; otherwise, this will throw an exception or error message.
- Square and cube transformation has a moderate effect on distribution shape. It can be used to reduce left skewness.
- Square and cube root transformation has a fairly strong transformation effect on the distribution shape but it is weaker than logarithms. It can be applied to right-skewed data.
- Discretization can also be used to transform a numeric column or attribute. For example, the age of a group of candidates can be grouped into intervals such as 0-10, 11-20, and so on. We can also use discretization to assign conceptual labels instead of intervals such as youth, adult, and senior.

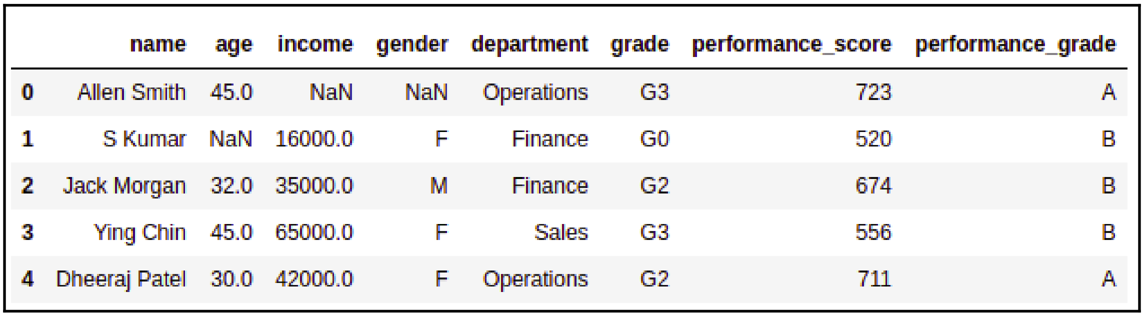
If the feature is right-skewed or positively skewed or grouped at lower values, then we can apply the square root, cube root, and logarithmic transformations, while if the feature is left-skewed or negative skewed or grouped at higher values, then we can apply the cube, square, and so on.

Let's take a look at an example of discretization transformation:

```
# Read the data
data=pd.read_csv('employee.csv')

# Create performance grade function
def performance_grade(score):
    if score>=700:
        return 'A'
    elif score<700 and score >= 500:
        return 'B'
    else:
        return 'C'
```

```
# Apply performance grade function on whole DataFrame using apply()
function.
data['performance_grade']=data.performance_score.apply(performance_grade)

# See initial 5 records
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	performance_grade
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	A
1	S Kumar	NaN	16000.0	F	Finance	G0	520	B
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	B
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	B
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	A

In the preceding example, we loaded the dataset and created the `performance_grade()` function. The `performance_grade()` function takes the performance score and converts it into grades; that is, `A`, `B`, and `C`.

Feature splitting

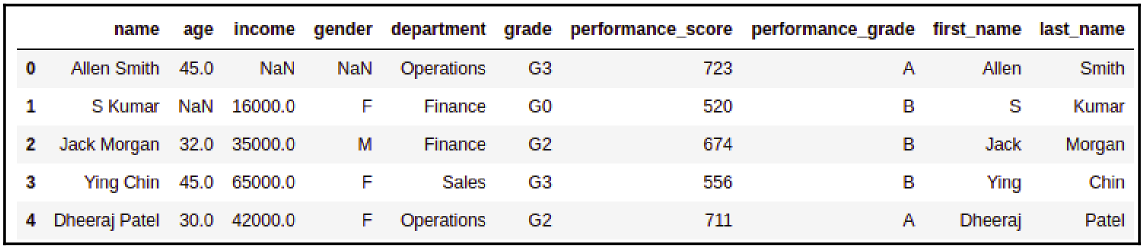
Feature splitting helps data analysts and data scientists create more new features for modeling. It allows machine learning algorithms to comprehend features and uncover potential information for decision-making; for example, splitting name features into first, middle, and last name and splitting an address into house number, locality, landmark, area, city, country, and zip code.

Composite features such as string and date columns violate the tidy data principles. Feature splitting is a good option if you wish to generate more features from a composite feature. We can utilize the components of a column to do this. For example, from a date object, we can easily get the year, month, and weekday. These features may directly affect the prediction model. There is no rule of thumb when it comes to breaking the features into components; this depends on the characteristics of the feature:

```
# Split the name column in first and last name
data['first_name']=data.name.str.split(" ").map(lambda var: var[0])
data['last_name']=data.name.str.split(" ").map(lambda var: var[1])
```

```
# Check top-5 records
data.head()
```

This results in the following output:



	name	age	income	gender	department	grade	performance_score	performance_grade	first_name	last_name
0	Allen Smith	45.0	NaN	NaN	Operations	G3	723	A	Allen	Smith
1	S Kumar	NaN	16000.0	F	Finance	G0	520	B	S	Kumar
2	Jack Morgan	32.0	35000.0	M	Finance	G2	674	B	Jack	Morgan
3	Ying Chin	45.0	65000.0	F	Sales	G3	556	B	Ying	Chin
4	Dheeraj Patel	30.0	42000.0	F	Operations	G2	711	A	Dheeraj	Patel

In the preceding example, we split the name column using the `split()` and `map()` functions. The `split()` function splits the name column using a space, while the `map()` function assigns the first divided string to the first name and the second divided string to the last name.

Summary

In this chapter, we explored data preprocessing and feature engineering with Python. This had helped you gain important skills for data analysis. The main focus of this chapter was on cleaning and filtering out dirty data. We started with EDA and discussed data filtering, handling missing values, and outliers. After this, we focused on feature engineering tasks such as transformation, feature encoding, feature scaling, and feature splitting. We then explored various methods and techniques we can use when it comes to feature engineering.

In the next chapter, Chapter 8, *Signal Processing and Time Series*, we will focus on the importance of signal processing and time series data in Python. We'll start this chapter by analyzing time series data and discussing moving averages, autocorrelations, autoregressive models, and ARMA models. Then, we will look at signal processing and discuss Fourier transform, spectral transform, and filtering on signals.

8
Signal Processing and Time Series

Signal processing is a subdomain of electrical engineering and applied mathematics. It covers the analysis and processing of time-related variables or variables that change over time, such as analog and digital signals. Analog signals are non-digitized signals, such as radio or telephone signals. Digital signals are digitized, discrete, time-sampled signals, such as computer and digital device signals. Time-series analysis is the category of signal processing that deals with ordered or sequential lists of observations. This data can be ordered hourly, daily, weekly, monthly, or annually. The time component in the time series plays a very important role. We need to extract all the relations in the data with respect to time. There are lots of examples that are related to time-series analysis, such as the production and sales of a product, predicting stock prices on an hourly or daily basis, economic forecasts, and census analysis.

In this chapter, our main focus is on signal processing and time-series operations using the NumPy, SciPy, `pandas`, and `statsmodels` libraries. This chapter will be helpful for data analysts to understand trends and patterns and forecast sales, stock prices, production, population, rainfall, and weather temperature.

We will cover the following topics in this chapter:

- The `statsmodels` modules
- Moving averages
- Window functions
- Defining cointegration
- STL decomposition
- Autocorrelation
- Autoregressive models

- ARMA models
- Generating periodic signals
- Fourier analysis
- Spectral analysis filtering

Technical requirements

This chapter has the following technical requirements:

- You can find the code and the dataset at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter08`.
- All the code blocks are in the `Ch8.ipynb` file.
- This chapter uses two CSV files (`beer_production.csv` and `sales.csv`) for practice purposes.
- In this chapter, we will use the `pandas` and Scikit-learn Python libraries.

The statsmodels modules

`statsmodels` is an open source Python module that offers functionality for various statistical operations, such as central values (mean, mode, and median), dispersion measures (standard deviation and variance), correlations, and hypothesis tests.

Let's install `statsmodels` using `pip` and run the following command:

```
pip3 install statsmodels
```

`statsmodels` provides the `statsmodels.tsa` submodule for time-series operations. `statsmodels.tsa` provides useful time-series methods and techniques, such as autoregression, autocorrelation, partial autocorrelation, moving averages, SimpleExpSmoothing, Holt's linear, Holt-Winters, ARMA, ARIMA, **vector autoregressive** (**VAR**) models, and lots of helper functions, which we will explore in the upcoming sections.

Moving averages

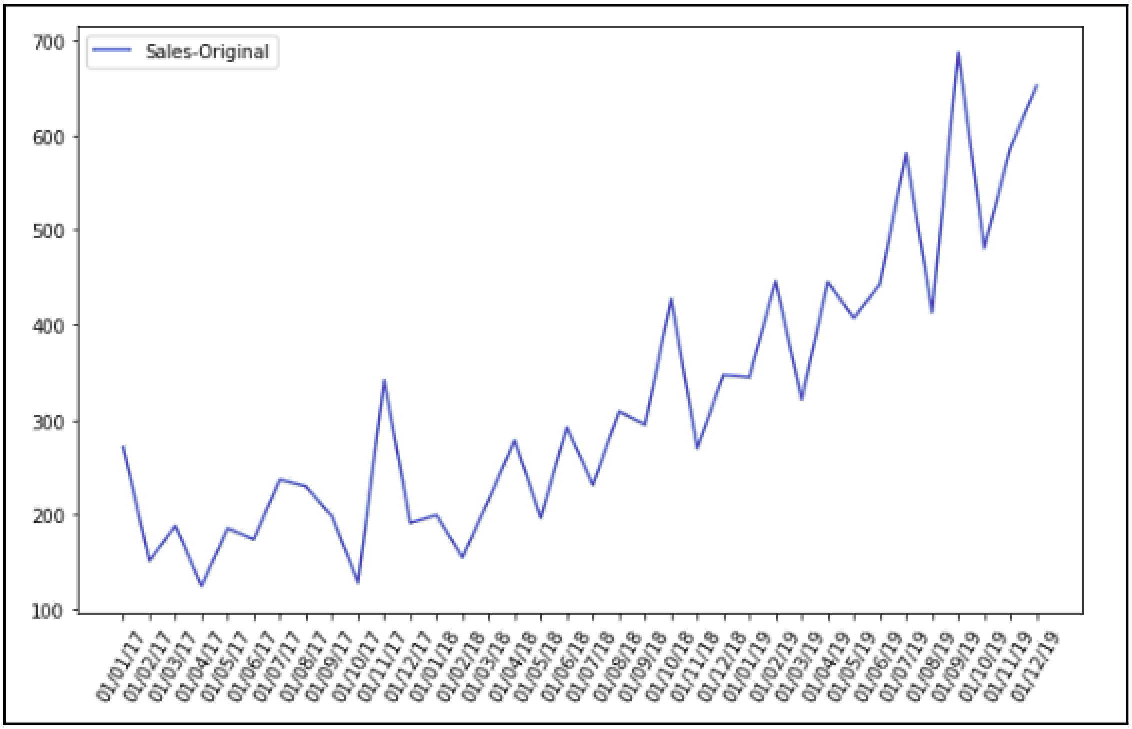
Moving averages, or rolling means, are time-series filters that filter impulsive responses by averaging the set or window of observations. It uses window size concepts and finds the average of the continuous window slides for each period. The simple moving average can be represented as follows:

$$SMA = \frac{a_m + a_{m-1} + \cdots + a_{m-(n-1)}}{n},$$

There are various types of moving averages available, such as centered, double, and weighted moving averages. Let's find the moving average using the `rolling()` function, but before that, we'll first load the data and visualize it:

```
# import needful libraries
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Read dataset
sales_data = pd.read_csv('sales.csv')

# Setting figure size
plt.figure(figsize=(10,6))

# Plot original sales data
plt.plot(sales_data['Time'], sales_data['Sales'], label="Sales-Original")

# Rotate xlabels
plt.xticks(rotation=60)

# Add legends
plt.legend()

#display the plot
plt.show()
```

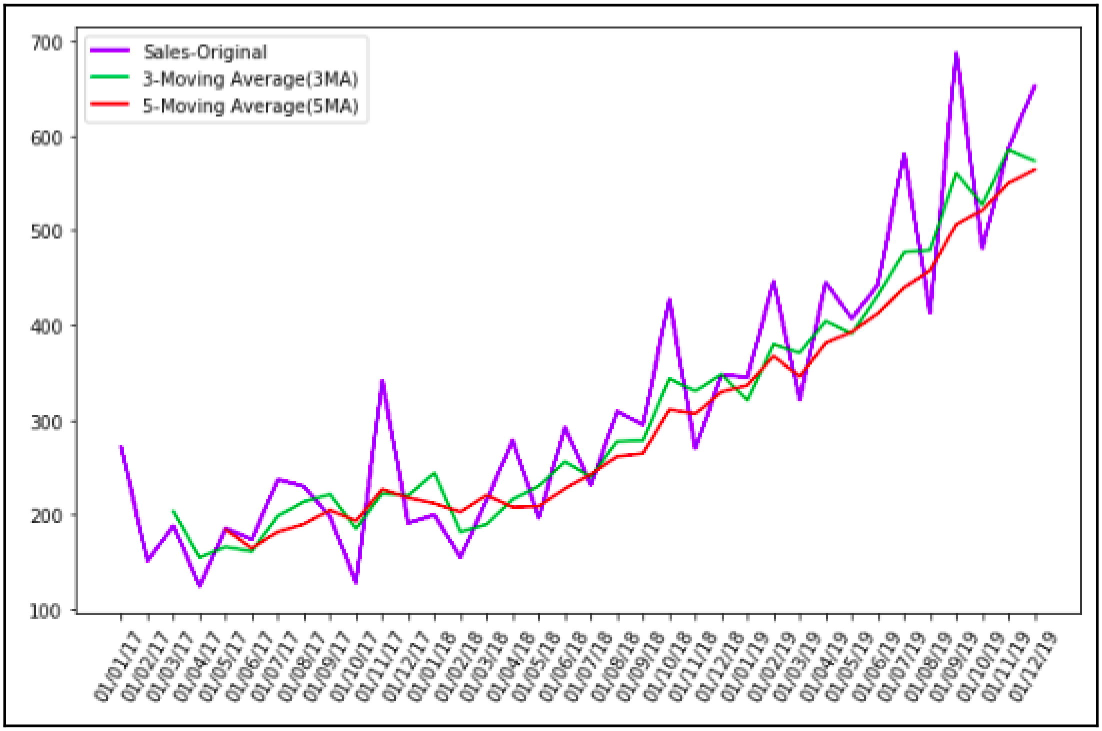
This results in the following output:



In the preceding code, we have read the sales dataset of 36 months from January 2017 to December 2019 and plotted it using Matplotlib. Now, we will compute the moving average using the rolling function:

```
# Moving average with window 3
sales_data['3MA']=sales_data['Sales'].rolling(window=3).mean()

# Moving average with window 5
sales_data['5MA']=sales_data['Sales'].rolling(window=5).mean()

# Setting figure size
plt.figure(figsize=(10,6))

# Plot original sales data
plt.plot(sales_data['Time'], sales_data['Sales'], label="Sales-Original",
color="blue")

# Plot 3-Moving Average of sales data
plt.plot(sales_data['Time'], sales_data['3MA'], label="3-Moving
Average(3MA)", color="green")

# Plot 5-Moving Average of sales data
plt.plot(sales_data['Time'], sales_data['5MA'], label="5-Moving
Average(5MA)", color="red")

# Rotate xlabels
plt.xticks(rotation=60)

# Add legends
plt.legend()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding code, we computed the 3 and 5 moving averages using the rolling mean and displayed the line plot using Matplotlib. Now, let's see different types of window functions for moving averages in the next section.

Window functions

NumPy offers several window options that can compute weights in a rolling window as we did in the previous section.

The window function uses an interval for spectral analysis and filter design (for more background information, refer to `http://en.wikipedia.org/wiki/Window_function`). The boxcar window is a rectangular window with the following formula:

$$w(n) = 1$$

The triangular window is shaped like a triangle and has the following formula:

$$w(n) = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{L}{2}} \right|$$

Here, *L* can be equal to *N*, *N*+1, or *N*–1.

If the value of *L* is *N*–1, it is known as the Bartlett window and has the following formula:

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right)$$

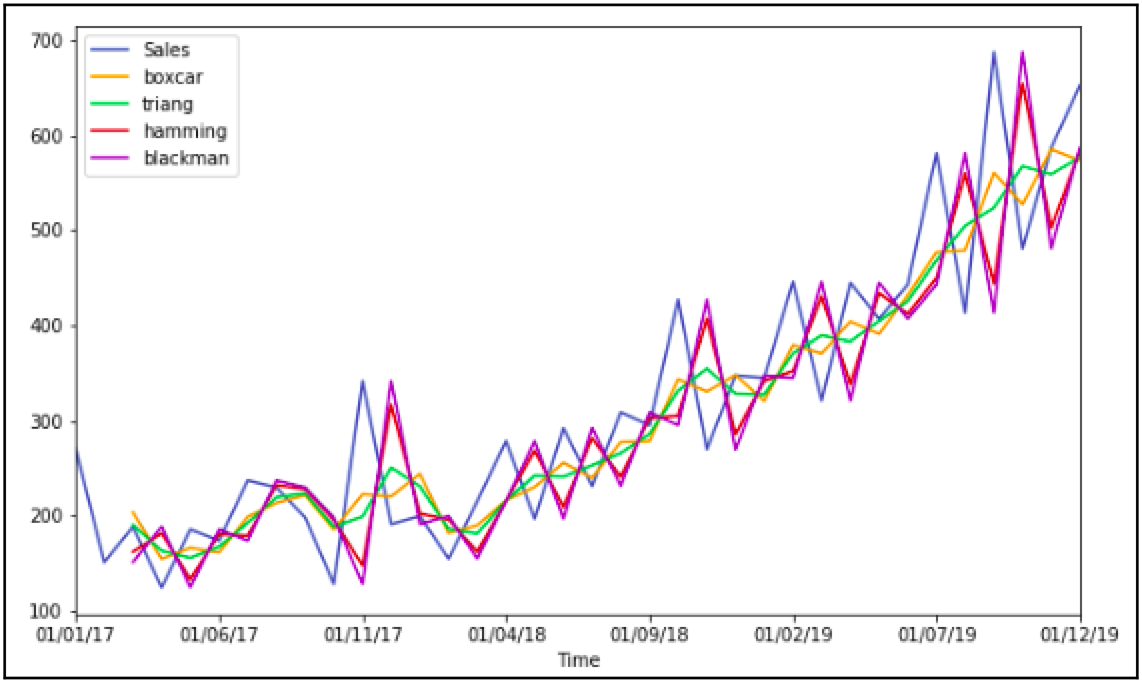
$$a_0 = \frac{1-\alpha}{2}; a_1 = \frac{1}{2}; a_2 = \frac{\alpha}{2}$$

In the `pandas` module, the `DataFrame.rolling()` function provides the same functionality using the `win_type` parameter for different window functions. Another parameter is the window for defining the size of the window, which is easy to set as shown in the previous section. Let's use the `win_type` parameter and try different window functions:

```
# import needful libraries
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

```
# Read dataset
sales_data = pd.read_csv('sales.csv', index_col ="Time")

# Apply all the windows on given DataFrame
sales_data['boxcar']=sales_data.Sales.rolling(3, win_type ='boxcar').mean()
sales_data['triang']=sales_data.Sales.rolling(3, win_type ='triang').mean()
sales_data['hamming']=sales_data.Sales.rolling(3, win_type
='hamming').mean()
sales_data['blackman']=sales_data.Sales.rolling(3, win_type
='blackman').mean()

#Plot the rolling mean of all the windows
sales_data.plot(kind='line',figsize=(10,6))
```

This results in the following output:



In the preceding code block, we have plotted the rolling mean for different window functions, such as boxcar, triangular, hamming, and Blackman window, using the `win_type` parameter in the `rolling()` function. Now, let's learn how to find a correlation between two time series using cointegration.

Defining cointegration

Cointegration is just like a correlation that can be viewed as a superior metric to define the relatedness of two time series. Cointegration is the stationary behavior of the linear combination of two time series. In this way, the trend of the following equation must be stationary:

$$y(t) - a\,x(t)$$

Consider a drunk man and his dog out on a walk. Correlation tells us whether they are going in the same direction. Cointegration tells us something about the distance over time between the man and his dog. We will show cointegration using randomly generated time-series and real data. The **Augmented Dickey-Fuller** (**ADF**) test tests for a unit root in a time series and can be used to determine the stationarity of time series.

Let's see an example to understand the cointegration of two time series.

You can check out the full code for this example at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/blob/master/Chapter08/Ch8.ipynb`.

Let's get started with the cointegration demo:

1. Import the required libraries and define the following function to calculate the ADF statistic:

```
# Import required library
import statsmodels.api as sm
import pandas as pd
import statsmodels.tsa.stattools as ts
import numpy as np

# Calculate ADF function
def calc_adf(x, y):
    result = sm.OLS(x, y).fit()
    return ts.adfuller(result.resid)
```

2. Load the Sunspot data into a NumPy array:

```
# Read the Dataset
data = sm.datasets.sunspots.load_pandas().data.values
N = len(data)
```

3. Generate a sine wave and calculate the cointegration of the sine with itself:

```
# Create Sine wave and apply ADF test
t = np.linspace(-2 * np.pi, 2 * np.pi, N)
sine = np.sin(np.sin(t))
print("Self ADF", calc_adf(sine, sine))
```

The code should print the following:

```
Self ADF (-5.0383000037165746e-16, 0.95853208606005591, 0, 308,
{'5%': -2.8709700936076912, '1%': -3.4517611601803702, '10%':
-2.5717944160060719}, -21533.113655477719)
```

In the printed results, the first value represents the ADF metric and the second value represents the p-value. As you can see, the p-value is very high. The following values are the lag and sample size. The dictionary at the end gives the t-distribution values for this exact sample size.

4. Now, add noise to the sine to demonstrate how noise will influence the signal:

```
# Apply ADF test on Sine and Sine with noise
noise = np.random.normal(0, .01, N)
print("ADF sine with noise", calc_adf(sine, sine + noise))
```

With the noise, we get the following results:

```
ADF sine with noise (-7.4535502402193075, 5.5885761455106898e- 11,
3, 305, {'5%': -2.8710633193086648, '1%': -3.4519735736206991,
'10%': -2.5718441306100512}, -1855.0243977703672)
```

The p-value has gone down considerably. The ADF metric here, `-7.45`, is lower than all the critical values in the dictionary. All these are strong arguments to reject cointegration.

5. Let's generate a cosine of a larger magnitude and offset. Again, let's add noise to it:

```
# Apply ADF test on Sine and Cosine with noise
cosine = 100 * np.cos(t) + 10

print("ADF sine vs cosine with noise", calc_adf(sine, cosine +
noise))
```

The following values get printed:

```
ADF sine vs cosine with noise (-17.927224617871534,
2.8918612252729532e-30, 16, 292, {'5%': -2.8714895534256861, '1%':
-3.4529449243622383, '10%': -2.5720714378870331},
-11017.837238220782)
```

Similarly, we have strong arguments to reject cointegration. Checking for cointegration between the sine and sunspots gives the following output:

```
print("Sine vs sunspots", calc_adf(sine, data))
```

The following values get printed:

```
Sine vs sunspots (-6.7242691810701016, 3.4210811915549028e-09, 16,
292,
{'5%': -2.8714895534256861, '1%': -3.4529449243622383,
'10%': -2.5720714378870331}, -1102.5867415291168)
```

The confidence levels are roughly the same for the pairs used here because they are dependent on the number of data points, which doesn't vary much. The outcome is summarized in the following table:

Pair	Statistic	p-value	5%	1%	10%	Reject
Sine with self	-5.03E-16	0.95	-2.87	-3.45	-2.57	No
Sine versus sine with noise	-7.45	5.58E-11	-2.87	-3.45	-2.57	Yes
Sine versus cosine with noise	-17.92	2.89E-30	-2.87	-3.45	-2.57	Yes
Sine versus sunspots	-6.72	3.42E-09	-2.87	-3.45	-2.57	Yes

In the preceding table, the results are summarized for all four sine waves and their significance level with rejection/acceptance is discussed. Let's now move on to another important topic of the chapter, which is STL decomposition of any time series.

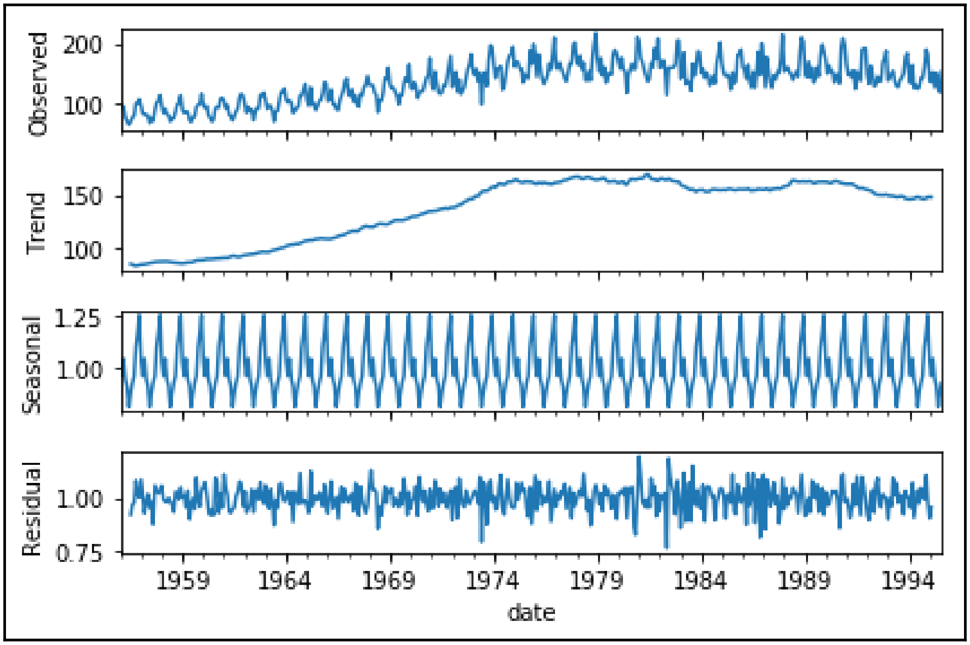
STL decomposition

STL stands for **seasonal and trend decomposition using LOESS**. STL is a time-series decomposition method that can decompose an observed signal into a trend, seasonality, and residual. It can estimate non-linear relationships and handle any type of seasonality. The `statsmodels.tsa.seasonal` subpackage offers the `seasonal_decompose` method for splitting a given input signal into trend, seasonality, and residual.

Let's see the following example to understand STL decomposition:

```
# import needful libraries
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose

# Read the dataset
data = pd.read_csv('beer_production.csv')
data.columns= ['date','data']

# Change datatype to pandas datetime
data['date'] = pd.to_datetime(data['date'])
data=data.set_index('date')

# Decompose the data
decomposed_data = seasonal_decompose(data, model='multiplicative')

# Plot decomposed data
decomposed_data.plot()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding code block, the given time-series signal is decomposed into trend, seasonal, and residual components using the `seasonal_decompose()` function of the `statsmodels` module. Let's now jump to autocorrelation to understand the relationship between a time series and its lagged series.

Autocorrelation

Autocorrelation, or lagged correlation, is the correlation between a time series and its lagged series. It indicates the trend in the dataset. The autocorrelation formula can be defined as follows:

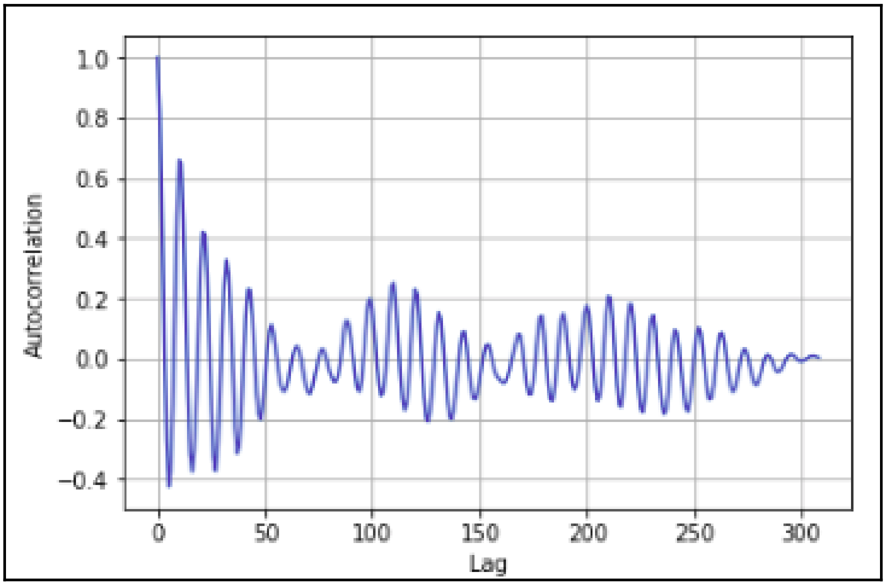
$$\frac{\sum (Y_t - \bar{Y})(Y_{t-k} - \bar{Y})}{\sum (Y_t - \bar{Y})^2}$$

We can calculate the autocorrelation using the NumPy `correlate()` function to calculate the actual autocorrelation of sunspot cycles. We can also directly visualize the autocorrelation plot using the `autocorrelation_plot()` function. Let's compute the autocorrelation and visualize it:

```
# import needful libraries
import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt

# Read the dataset
data = sm.datasets.sunspots.load_pandas().data

# Calculate autocorrelation using numpy
dy = data.SUNACTIVITY - np.mean(data.SUNACTIVITY)
dy_square = np.sum(dy ** 2)

# Cross-correlation
sun_correlated = np.correlate(dy, dy, mode='full')/dy_square
result = sun_correlated[int(len(sun_correlated)/2):]

# Diplay the Chart
plt.plot(result)

# Display grid
plt.grid(True)
```

```
# Add labels
plt.xlabel("Lag")

plt.ylabel("Autocorrelation")
# Display the chart
plt.show()
```

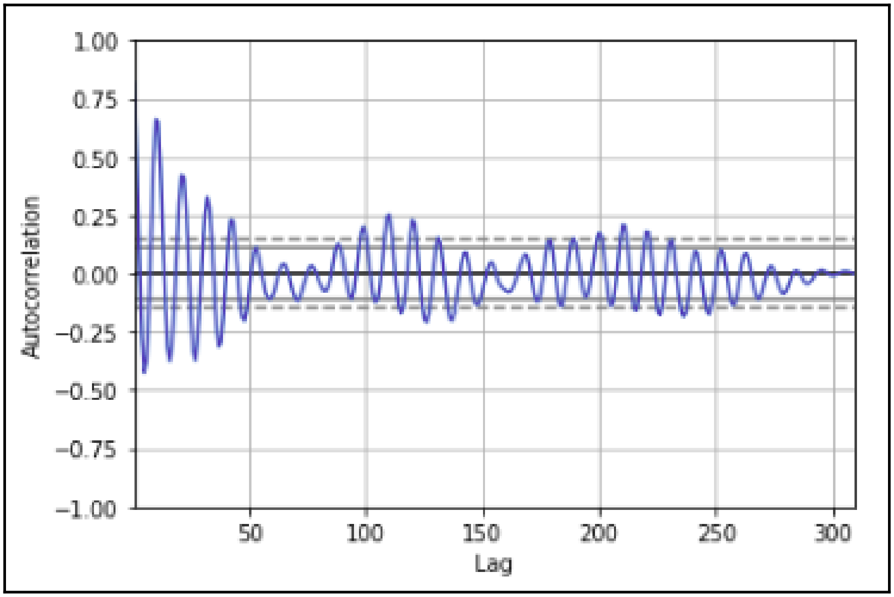
This results in the following output:



In the preceding code block, we have seen an autocorrelation example using the NumPy module. Let's compute the autocorrelation plot produced by `pandas`:

```
from pandas.plotting import autocorrelation_plot

# Plot using pandas function
autocorrelation_plot(data.SUNACTIVITY)
```

This results in the following output:



In the preceding code block, we have produced an autocorrelation plot using the `autocorrelation_plot()` function of the `pandas` library. It is easier to draw the autocorrelation plot using the `pandas` library compared to the NumPy library. Let's now jump to autoregressive models for time-series prediction.

Autoregressive models

Autoregressive models are time-series models used to predict future incidents. The following formula shows this:

$$x_t = c + \sum_{i=1}^{p} a_i x_{t-i} + \in_t$$

In the preceding formula, *c* is a constant and the last term is a random component, also known as white noise.

Let's build the autoregression model using the statsmodels.tsa subpackage:

1. Import the libraries and read the dataset:

```
# import needful libraries
from statsmodels.tsa.ar_model import AR
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import statsmodels.api as sm
from math import sqrt

# Read the dataset
data = sm.datasets.sunspots.load_pandas().data
```

2. Split the Sunspot data into train and test sets:

```
# Split data into train and test set
train_ratio=0.8

train=data[:int(train_ratio*len(data))]
test=data[int(train_ratio*len(data)):]
```

3. Train and fit the autoregressive model:

```
# AutoRegression Model training
ar_model = AR(train.SUNACTIVITY)
ar_model = ar_model.fit()

# print lags and
print("Number of Lags:", ar_model.k_ar)
print("Model Coefficients:\n", ar_model.params)
```

This results in the following output:

```
Number of Lags: 15
Model Coefficients:
const             9.382322
L1.SUNACTIVITY    1.225684
L2.SUNACTIVITY   -0.512193
L3.SUNACTIVITY   -0.130695
L4.SUNACTIVITY    0.193492
L5.SUNACTIVITY   -0.168907
L6.SUNACTIVITY    0.054594
L7.SUNACTIVITY   -0.056725
L8.SUNACTIVITY    0.109404
L9.SUNACTIVITY    0.108993
L10.SUNACTIVITY -0.117063
```

```
L11.SUNACTIVITY  0.200454
L12.SUNACTIVITY -0.075111
L13.SUNACTIVITY -0.114437
L14.SUNACTIVITY  0.177516
L15.SUNACTIVITY -0.091978
dtype: float64
```

In the preceding code, we have read the Sunspot dataset and split it into two parts: train and test sets. Then, we built the autoregressive model by creating an instance and fitting a model. Let's make predictions and assess the model's performance.

4. Perform predictions and assess the model:

```
# make predictions
start_point = len(train)
end_point = start_point + len(test)-1
pred = ar_model.predict(start=start_point, end=end_point,
dynamic=False)

# Calculate errors
mae = mean_absolute_error(test.SUNACTIVITY, pred)
mse = mean_squared_error(test.SUNACTIVITY, pred)
rmse = sqrt(mse)
print("MAE:",mae)
print("MSE:",mse)
print("RMSE:",rmse)
```

This results in the following output:

```
MAE: 31.17846098350052
MSE: 1776.9463826165913
RMSE: 42.15384184883498
```

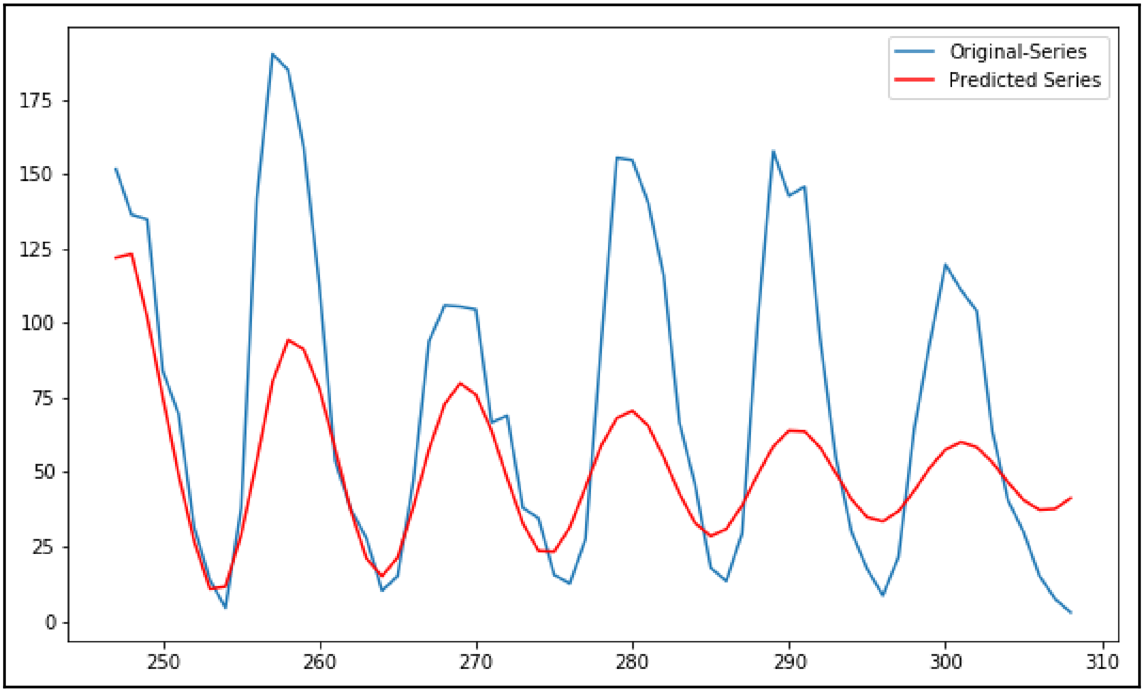
In the preceding code block, we have made the predictions on the test dataset and assessed the model's performance using **Mean Absolute Error** (**MAE**), **Mean Squared Error** (**MSE**), and **Root Mean Squared Error** (**RMSE**). Let's plot the line plot for the original series and prediction series.

5. Let's plot the predicted and original series to understand the forecasting results in a better way:

```
# Setting figure size
plt.figure(figsize=(10,6))

# Plot test data
plt.plot(test.SUNACTIVITY, label='Original-Series')
```

```
# Plot predictions
plt.plot(pred, color='red', label='Predicted Series')

# Add legends
plt.legend()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding plot, we can see the original series and predicted series using the autoregressive model. After generating the autoregressive model, we need to jump to one more advanced approach for time-series prediction, which is **Autoregressive Moving Average** (**ARMA**).

ARMA models

The ARMA model blends autoregression and moving averages. The ARMA model is commonly referred to as ARMA(*p*,*q*), where *p* is the order of the autoregressive part, and *q* is the order of the moving average:

$$x_t = c + \sum_{i=1}^{p} a_i x_{t-i} + \sum_{i=1}^{q} b_i \varepsilon_{t-i} + \in_t$$

In the preceding formula, just like in the autoregressive model formula, we have a constant and a white noise component; however, we try to fit the lagged noise components as well:

1. Import the libraries and read the dataset:

```
# import needful libraries
import statsmodels.api as sm
from statsmodels.tsa.arima_model import ARMA
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from math import sqrt

# Read the dataset
data = sm.datasets.sunspots.load_pandas().data
data.drop('YEAR',axis=1,inplace=True)
```

2. Split the Sunspot data into train and test sets:

```
# Split data into train and test set
train_ratio=0.8
train=data[:int(train_ratio*len(data))]
test=data[int(train_ratio*len(data)):]
```

3. Train and fit the autoregressive model:

```
# AutoRegression Model training
arma_model = ARMA(train, order=(10,1))
arma_model = arma_model.fit()
```

4. Perform predictions and assess the model:

```
# make predictions
start_point = len(train)
end_point = start_point + len(test)-1
pred = arma_model.predict(start_point,end_point)

# Calculate errors
mae = mean_absolute_error(test.SUNACTIVITY, pred)
mse = mean_squared_error(test.SUNACTIVITY, pred)
rmse = sqrt(mse)
print("MAE:",mae)
print("MSE:",mse)
print("EMSE:",rmse)
```

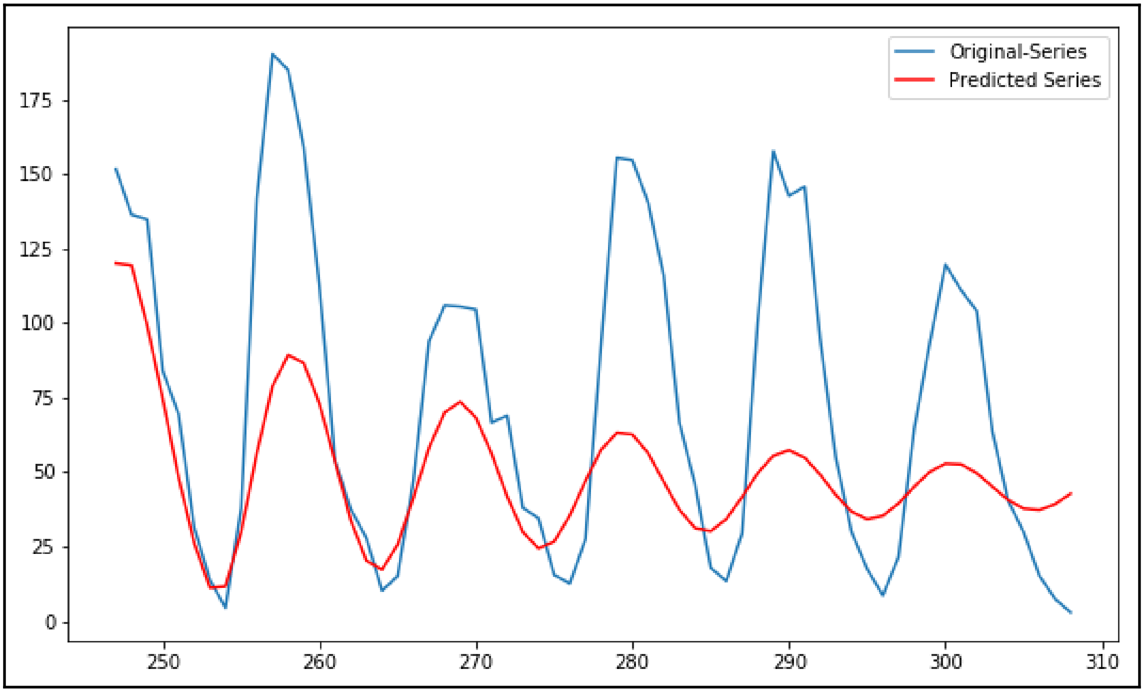
This results in the following output:

```
MAE: 33.95457845540467
MSE: 2041.3857010355755
EMSE: 45.18169652675268
```

5. Let's plot the predicted and original series to understand the forecasting results in a better way:

```
# Setting figure size
plt.figure(figsize=(10,6))

# Plot test data
plt.plot(test, label='Original-Series')

# Plot predictions
plt.plot(pred, color='red', label='Predicted Series')

# Add legends
plt.legend()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding code, we have read the Sunspot dataset and split it into two parts: train and test sets. Then, we built the ARMA model by creating an instance and fitting a model. We made the predictions on the test dataset and assessed the model performance using MAE, MSE, and RMSE. Finally, we saw the line plot for the original series and prediction series. Let's jump to one more important topic, which is generating periodic signals.

Generating periodic signals

Many natural phenomena are regular and trustworthy, such as an accurate clock. Some phenomena exhibit patterns that seem regular. A group of scientists found three cycles in the sunspot activity with the Hilbert-Huang transform (see `https://en.wikipedia.org/wiki/Hilbert%E2%80%93Huang_transform`). The cycles have a duration of 11, 22, and 100 years, approximately. Normally, we would simulate a periodic signal using trigonometric functions such as a sine function. You probably remember a bit of trigonometry from high school. That's all we need for this example. Since we have three cycles, it seems reasonable to create a model that is a linear combination of three sine functions. This just requires a tiny adjustment of the code for the autoregressive model:

1. Create model, error, and fit functions:

```
# Import required libraries
import numpy as np
import statsmodels.api as sm
from scipy.optimize import leastsq
import matplotlib.pyplot as plt

# Create model function
def model(p, t):
    C, p1, f1, phi1 , p2, f2, phi2, p3, f3, phi3 = p
    return C + p1 * np.sin(f1 * t + phi1) + p2 * np.sin(f2 * t +
phi2) +p3 * np.sin(f3 * t + phi3)

# Create error function
def error(p, y, t):
    return y - model(p, t)

# Create fit function
def fit(y, t):
    p0 = [y.mean(), 0, 2 * np.pi/11, 0, 0, 2 * np.pi/22, 0, 0, 2 *
np.pi/100, 0]
    params = leastsq(error, p0, args=(y, t))[0]
    return params
```

2. Let's load the dataset:

```
# Load the dataset
data_loader = sm.datasets.sunspots.load_pandas()
sunspots = data_loader.data["SUNACTIVITY"].values
years = data_loader.data["YEAR"].values
```

3. Apply and fit the model:

```
# Apply and fit the model
cutoff = int(.9 * len(sunspots))
params = fit(sunspots[:cutoff], years[:cutoff])
print("Params", params)

pred = model(params, years[cutoff:])
actual = sunspots[cutoff:]
```

4. Print the results:

```
print("Root mean square error", np.sqrt(np.mean((actual - pred) **
2)))
print("Mean absolute error", np.mean(np.abs(actual - pred)))
print("Mean absolute percentage error", 100 *
np.mean(np.abs(actual - pred)/actual))
mid = (actual + pred)/2
print("Symmetric Mean absolute percentage error", 100 *
       np.mean(np.abs(actual - pred)/mid))
print("Coefficient of determination", 1 - ((actual - pred)
        **2).sum()/ ((actual - actual.mean()) ** 2).sum())
```

This results in the following output:

```
Params [47.1880006  28.89947462   0.56827279 6.51178464  4.55214564
        0.29372076 -14.30924768 -18.16524123 0.06574835
-4.37789476]
Root mean square error 59.56205597915569
Mean absolute error 44.58158470150657
Mean absolute percentage error 65.16458348768887
Symmetric Mean absolute percentage error 78.4480696873044
Coefficient of determination -0.3635315489903188
```

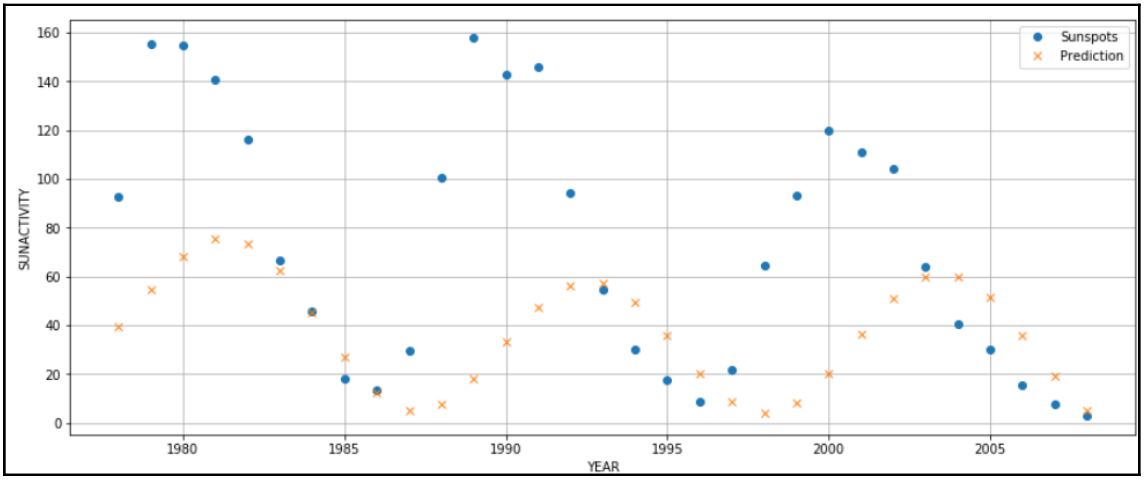
The first line displays the coefficients of the model we attempted. We have an MAE of 44, which means that we are off by that amount in either direction on average. We also want the coefficient of determination to be as close to 1 as possible to have a good fit. Instead, we get a negative value, which is undesirable. Let's create a graph to understand the results in detail.

5. Plot the original and predicted series:

```
year_range = data_loader.data["YEAR"].values[cutoff:]

# Plot the actual and predicted data points
plt.plot(year_range, actual, 'o', label="Sunspots")
plt.plot(year_range, pred, 'x', label="Prediction")
plt.grid(True)
```

```
        # Add labels
        plt.xlabel("YEAR")
        plt.ylabel("SUNACTIVITY")

        # Add legend
        plt.legend()

        # Display the chart
        plt.show()
```

This results in the following output:



From the preceding graph, we can conclude that the model is not able to capture the actual pattern of the series. This is why we get a negative coefficient of determination or R-squared. Now, we will look at another important technique for time-series analysis, Fourier analysis.

Fourier analysis

Fourier analysis uses the Fourier series concept thought up by the mathematician Joseph Fourier. The Fourier series is a mathematical method used to represent functions as an infinite series of sine and cosine terms. The functions in question can be real- or complex-valued:

$$\sum_{t=-\infty}^{\infty} \chi\left[t\right]e^{-i\omega t}$$

For Fourier analysis, the most competent algorithm is **Fast Fourier Transform** (**FFT**). FFT decomposes a signal into different frequency signals. This means it produces a frequency spectrum of a given signal. The SciPy and NumPy libraries provide functions for FFT.

The `rfft()` function performs FFT on real-valued data. We could also have used the `fft()` function, but it gives a warning on this Sunspot dataset. The `fftshift()` function moves the zero-frequency component to the middle of the spectrum.

Let's see the following example to understand FFT:

1. Import the libraries and read the dataset:

```
# Import required library
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
from scipy.fftpack import rfft
from scipy.fftpack import fftshift

# Read the dataset
data = sm.datasets.sunspots.load_pandas().data

# Create Sine wave
t = np.linspace(-2 * np.pi, 2 * np.pi,
len(data.SUNACTIVITY.values))
mid = np.ptp(data.SUNACTIVITY.values)/2
sine = mid + mid * np.sin(np.sin(t))
```

2. Compute the FFT for sine waves and sunspots:

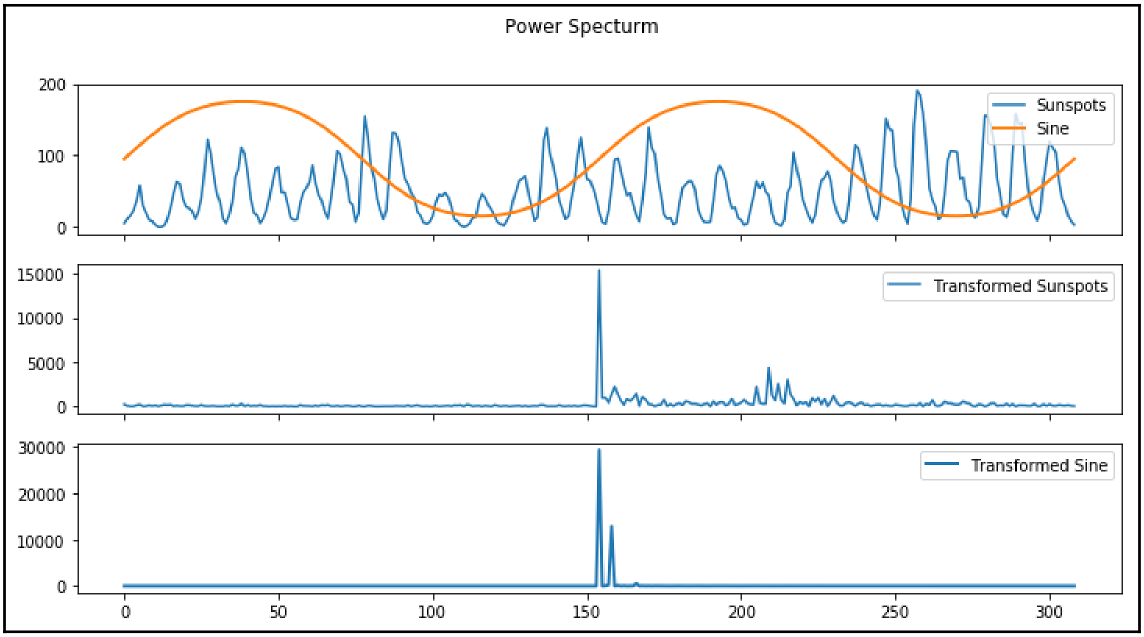
```
# Compute FFT for Sine wave
sine_fft = np.abs(fftshift(rfft(sine)))
print("Index of max sine FFT", np.argsort(sine_fft)[-5:])

# Compute FFT for sunspots dataset
transformed = np.abs(fftshift(rfft(data.SUNACTIVITY.values)))
print("Indices of max sunspots FFT", np.argsort(transformed)[-5:])
```

3. Create the subplots:

```
# Create subplots
fig, axs = plt.subplots(3,figsize=(12,6),sharex=True)
fig.suptitle('Power Specturm')
axs[0].plot(data.SUNACTIVITY.values, label="Sunspots")
axs[0].plot(sine, lw=2, label="Sine")
axs[0].legend() # Set legends
axs[1].plot(transformed, label="Transformed Sunspots")
```

```
axs[1].legend() # Set legends
axs[2].plot(sine_fft, lw=2, label="Transformed Sine")
axs[2].legend() # Set legends

# Display the chart
plt.show()
```

This results in the following output:



In the preceding code, first, we read the Sunspot dataset and created the sine wave. After that, we computed the FFT for the sine wave and the `SUNACTIVITY` column. Finally, we plotted the three graphs for the original series and sine wave and transformed sunspots and sine wave.

Spectral analysis filtering

In the previous section, we discussed the amplitude spectrum of the dataset. Now is the time to explore the power spectrum. The power spectrum of any physical signal can display the energy distribution of the signal. We can easily change the code and display the power spectrum by squaring the transformed signal using the following syntax:

```
plt.plot(transformed ** 2, label="Power Spectrum")
```

We can also plot the phase spectrum using the following Python syntax:

```
plt.plot(np.angle(transformed), label="Phase Spectrum")
```

Let's see the complete code for the power and phase spectrum for the Sunspot dataset:

1. Import the libraries and read the dataset:

```
# Import required library
import numpy as np
import statsmodels.api as sm
from scipy.fftpack import rfft
from scipy.fftpack import fftshift
import matplotlib.pyplot as plt

# Read the dataset
data = sm.datasets.sunspots.load_pandas().data
```

2. Compute `FFT`, `Spectrum`, and `Phase`:

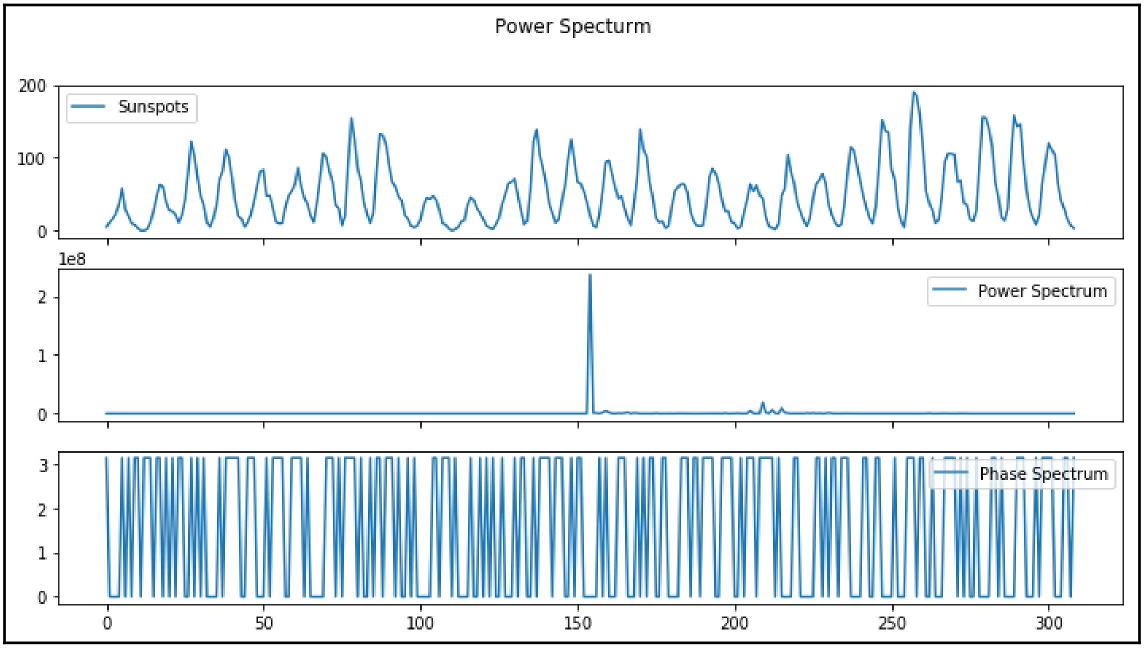
```
# Compute FFT
transformed = fftshift(rfft(data.SUNACTIVITY.values))

# Compute Power Spectrum
power=transformed ** 2

# Compute Phase
phase=np.angle(transformed)
```

3. Create the subplot:

```
# Create subplots
fig, axs = plt.subplots(3,figsize=(12,6),sharex=True)
fig.suptitle('Power Specturm')
axs[0].plot(data.SUNACTIVITY.values, label="Sunspots")
axs[0].legend() # Set legends
axs[1].plot(power, label="Power Spectrum")
axs[1].legend() # Set legends
```

```
axs[2].plot(phase, label="Phase Spectrum")
axs[2].legend() # Set legends

# Display the chart
plt.show()
```

This results in the following output:



In the preceding code, first, we read the Sunspot dataset and computed the FFT for the `SUNACTIVITY` column. After this, we computed the power and phase spectrum for the transformed FFT. Finally, we plotted the three graphs for the original series and the power and phase spectrums using subplots.

Summary

In this chapter, the time-series examples we used were annual sunspot cycles data, sales data, and beer production. We learned that it's common to try to derive a relationship between a value and another data point or a combination of data points with a fixed number of periods in the past in the same time series. We learned how moving averages convert the random variation trend into a smooth trend using a window size. We learned how the `DataFrame.rolling()` function provides `win_type` string parameters for different window functions. Cointegration is similar to correlation and is a metric to define the relatedness of two time series. We also focused on STL decomposition, autocorrelation, autoregression, the ARMA model, Fourier analysis, and spectral analysis filtering.

The next chapter, `Chapter 9`, *Supervised Learning – Regression Analysis*, will focus on the important topics of regression analysis and logistic regression in Python. The chapter starts with multiple linear regression, multicollinearity, dummy variables, and model evaluation measures. In the later sections of the chapter, the focus will be on logistic regression.

3

Section 3: Deep Dive into Machine Learning

The main objective of this section is to deep dive into machine learning algorithms and develop predictive models. This section focuses on regression, classification, PCA, and clustering methods. This section will mostly use pandas and scikit-learn.

This section includes the following chapters:

- `Chapter 9`, *Supervised Learning – Regression Analysis*
- `Chapter 10`, *Supervised Learning – Classification Techniques*
- `Chapter 11`, *Unsupervised Learning – PCA and Clustering*

9
Supervised Learning - Regression Analysis

Regression is the most popular algorithm in statistics and machine learning. In the machine learning and data science field, regression analysis is a member of the supervised machine learning domain that helps us to predict continuous variables such as stock prices, house prices, sales, rainfall, and temperature. As a sales manager at an electronic store, for example, say you need to predict the sales of upcoming weeks for all types of products, such as televisions, air conditioners, laptops, refrigerators, and many more. Lots of factors can affect your sales, such as weather conditions, festivals, promotion strategy, competitor offers, and so on. Regression analysis is one of the tools that can help you to identify the importance of such factors that are important to make decisions at the store.

Regression analysis identifies how the dependent variable depends upon independent variables. For example, say as an education officer you want to identify the impact of sports activities, smart classes, teacher-student ratio, extra classes, and teachers' training on students' results. **Ordinary Least Square** (**OLS**) minimizes the sum of squares error (or error variance) to find out the best fit function. It predicts the most probable outcome under the given conditions. The main objective of this chapter is to learn the fundamentals of **Multiple Linear Regression** (**MLR**), multicollinearity, dummy variables, regression, and model evaluation measures such as R-squared, **Mean Squared Error** (**MSE**), **Mean Absolute Error** (**MAE**), and **Root Mean Square Error** (**RMSE**). Another objective is creating a logistic regression classification model.

The topics covered in this chapter are listed as follows:

- Linear regression
- Understanding multicollinearity
- Dummy variables
- Developing a linear regression model
- Evaluating regression model performance

- Fitting polynomial regression
- Regression models for classification
- Logistic regression
- Implementing logistic regression using scikit-learn

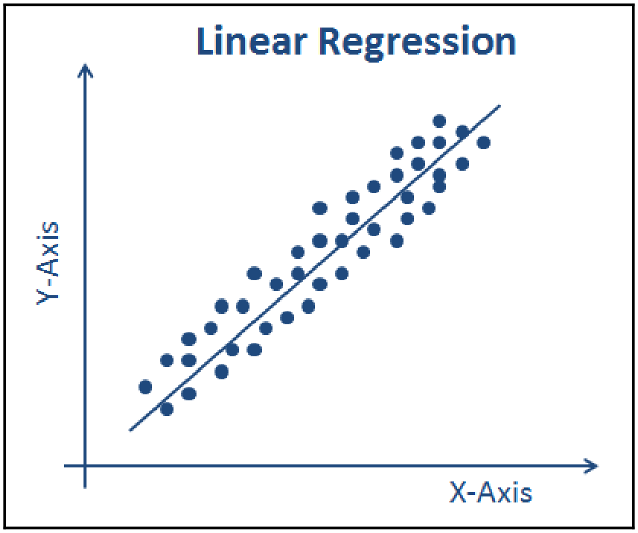
Technical requirements

This chapter has the following technical requirements:

- You can find the code and the datasets at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter09`.
- All the code blocks are available in the `ch9.ipynb` file.
- This chapter uses three CSV files (`Advertising.csv`, `bloodpress.txt`, and `diabetes.csv`) for practice purposes.
- In this chapter, we will use the Matplotlib, `pandas`, Seaborn, and scikit-learn Python libraries.

Linear regression

Linear regression is a kind of curve-fitting and prediction algorithm. It is used to discover the linear association between a dependent (or target) column and one or more independent columns (or predictor variables). This relationship is deterministic, which means it predicts the dependent variable with some amount of error. In regression analysis, the dependent variable is continuous and independent variables of any type are continuous or discrete. Linear regression has been applied to various kinds of business and scientific problems, for example, stock price, crude oil price, sales, property price, and GDP growth rate predictions. In the following graph, we can see how linear regression can fit data in two-dimensional space:



The main objective is to find the best-fit line to understand the relationship between variables with minimum error. Error in regression is the difference between the forecasted and actual values. Coefficients of regression are estimated using the OLS method. OLS tries to minimize the sum of squares residuals. Let's see the equation for the regression model:

$$y = \beta_0 + \beta_1 x + \varepsilon$$

Here, x is the independent variable and y is a dependent variable. β_0 intercepts are the coefficient of x, and ε (the Greek letter pronounced as epsilon) is an error term that will act as a random variable.

The parameters of linear regression are estimated using OLS. OLS is a method that is widely used to estimate the regression intercept and coefficients. It reduces the sum of squares of residuals (or error), which is the difference between the predicted and actual.

After getting an idea about linear regression, it's now time to learn about MLR.

Multiple linear regression

MLR is a generalized form of simple linear regression. It is a statistical method used to predict the continuous target variable based on multiple features or explanatory variables. The main objective of MLR is to estimate the linear relationship between the multiple features and the target variable. MLR has a wide variety of applications in real-life scenarios. The MLR model can be represented as a mathematical equation:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \varepsilon$$

Here, $x_1, x_2, \cdots x_p$ are the independent variables and $y$is a dependent variable. β_0 intercepts are coefficients of *x* and ε (the Greek letter pronounced as epsilon) is an error term that will act as a random variable.

Now that we know what linear regression is, let's move on to multicollinearity.

Understanding multicollinearity

Multicollinearity represents the very high intercorrelations or inter-association among the independent (or predictor) variables.

Multicollinearity takes place when independent variables of multiple regression analysis are highly associated with each other. This association is caused by a high correlation among independent variables. This high correlation will trigger a problem in the linear regression model prediction results. It's the basic assumption of linear regression analysis to avoid multicollinearity for better results:

- It occurs due to the inappropriate use of dummy variables.
- It also occurs due to the repetition of similar variables.
- It is also caused due to synthesized variables from other variables in the data.
- It can occur due to high correlation among variables.

Multicollinearity causes the following problems:

- It causes difficulty in estimating the regression coefficients precisely and coefficients become more susceptible to minor variations in the model.
- It can also cause a change in the signs and magnitudes of the coefficient.
- It causes difficulty in assessing the relative importance of independent variables.

Removing multicollinearity

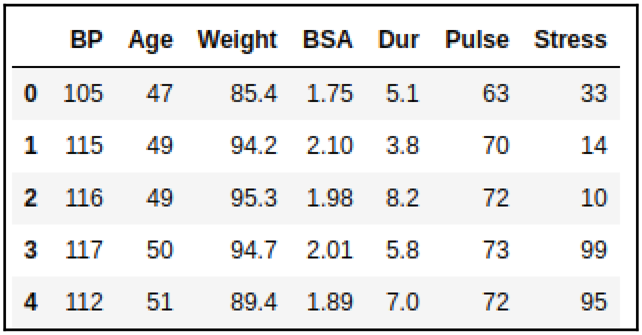
Multicollinearity can be detected using the following:

- The correlation coefficient (or correlation matrix) between independent variables
- **Variance Inflation Factor** (**VIF**)
- Eigenvalues

Correlation coefficients or correlation matrices will help us to identify a high correlation between independent variables. Using the correlation coefficient, we can easily detect the multicollinearity by checking the correlation coefficient magnitude:

```
# Import pandas
import pandas as pd

# Read the blood pressure dataset
data = pd.read_csv("bloodpress.txt",sep='\t')

# See the top records in the data
data.head()
```

This results in the following output:



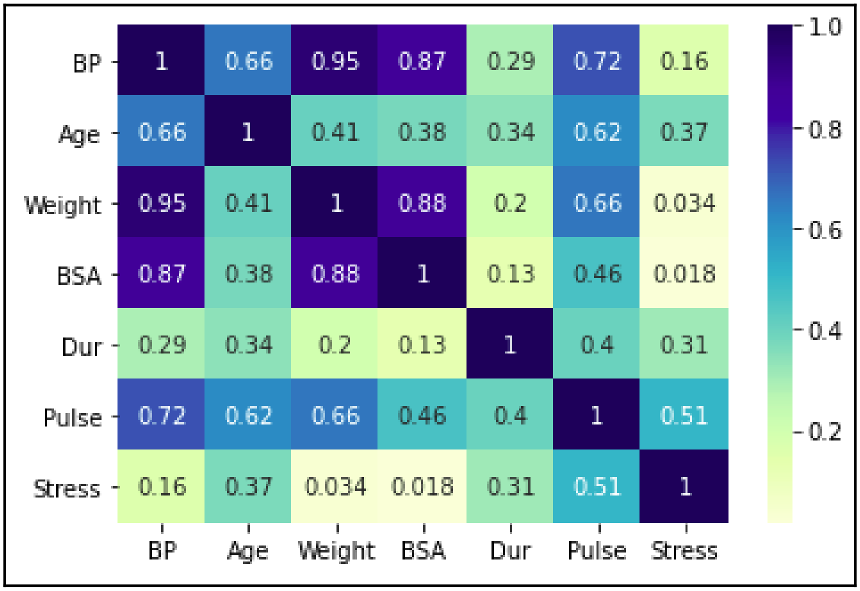
	BP	Age	Weight	BSA	Dur	Pulse	Stress
0	105	47	85.4	1.75	5.1	63	33
1	115	49	94.2	2.10	3.8	70	14
2	116	49	95.3	1.98	8.2	72	10
3	117	50	94.7	2.01	5.8	73	99
4	112	51	89.4	1.89	7.0	72	95

In the preceding code block, we read the `bloodpress.txt` data using the `read_csv()` function. We also checked the initial records of the dataset. This dataset has `BP`, `Age`, `Weight`, `BSA`, `Dur`, `Pulse`, and `Stress` fields. Let's check the multicollinearity in the dataset using the correlation matrix:

```
# Import seaborn and matplotlib
import seaborn as sns
import matplotlib.pyplot as plt

# Correlation matrix
```

```
corr=data.corr()

# Plot Heatmap on correlation matrix
sns.heatmap(corr, annot=True, cmap='YlGnBu')

# display the plot
plt.show()
```

This results in the following output:



In the preceding example, we are finding the correlation between multiple variables using the correlation matrix. We loaded the `bloodpress.txt` file and found the correlation using the `corr()` function. Finally, we visualized the correlation matrix using the `heatmap()` function.

Here, **BP** (**Blood Pressure**) is the dependent or target variable, and the rest of the columns are independent variables or features. We can see that **Weight** and **BSA** (**Body Surface Area**) have a high correlation. We need to remove one variable (either **Weight** or **BSA**) to remove the multicollinearity. In our case, weight is easier to measure compared to BSA, so experts will choose the weight and remove the BSA.

Dummy variables

Dummy variables are categorical independent variables used in regression analysis. It is also known as a Boolean, indicator, qualitative, categorical, and binary variable. Dummy variables convert a categorical variable with *N* distinct values into *N*–1 dummy variables. It only takes the 1 and 0 binary values, which are equivalent to existence and nonexistence.

`pandas` offers the `get_dummies()` function to generate the dummy values. Let's understand the `get_dummies()` function through an example:

```
# Import pandas module
import pandas as pd

# Create pandas DataFrame
data=pd.DataFrame({'Gender':['F','M','M','F','M']})

# Check the top-5 records
data.head()
```

This results in the following output:

	Gender
0	F
1	M
2	M
3	F
4	M

In the preceding code block, we have created the DataFrame with the `Gender` column and generated the dummy variable using the `get_dummies()` function. Let's see an example in the following code:

```
# Dummy encoding
encoded_data = pd.get_dummies(data['Gender'])

# Check the top-5 records of the dataframe
encoded_data.head()
```

This results in the following output:

```
  F M
0 1 0
1 0 1
2 0 1
3 1 0
4 0 1
```

Here, in the preceding example, the `get_dummies()` function is generating two columns, which means a separate column for each value.

We can remove one column to avoid collinearity using the `drop_first=True` argument and drop first the *N*–1 dummies out of *N* categorical levels by removing the first level:

```
# Dummy encoding
encoded_data = pd.get_dummies(data['Gender'], drop_first=True)

# Check the top-5 records of the dataframe
encoded_data.head()
```

This results in the following output:

```
  M
0 0
1 1
2 1
3 0
4 1
```

In the preceding code block, we have created the dummy variables for the `Gender` column using the `get_dummies()` function with the `drop_first=True` parameter. This has removed the first column and leaves *N*–1 columns. Let's now learn how to implement the linear regression model using the `scikit-learn` library.

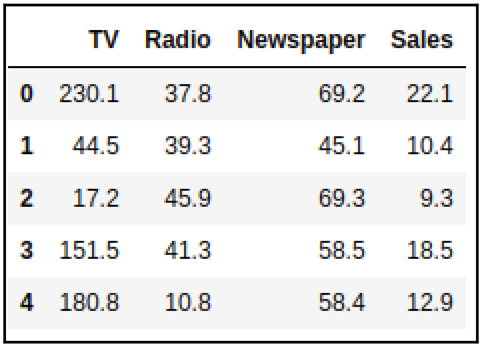
Developing a linear regression model

After understanding the concepts of regression analysis, multicollinearity, and dummy variables, it's time to get some hands-on experience with regression analysis. Let's learn how to build the regression model using the scientific toolkit for machine learning (scikit-learn):

1. We will first load the dataset using the `read_csv()` function:

```
# Import pandas
import pandas as pd

# Read the dataset using read_csv method
df = pd.read_csv("Advertising.csv")

# See the top-5 records in the data
df.head()
```

This results in the following output:



	TV	Radio	Newspaper	Sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9

Now that we have loaded the `Advertising.csv` dataset using `read_csv()` and checked the initial records using the `head()` function, we will split the data into two parts: dependent or target variable and independent variables or features.

2. In this step, we will split the data two times:

   - Split into two parts: dependent or target variable and independent variables or features.
   - Split data into training and test sets. This can be done using the following code:

```
# Independent variables or Features
```

```
X = df[['TV', 'Radio', 'Newspaper']]

# Dependent or Target variable
y = df.Sales
```

After splitting the columns into dependent and independent variable parts, we will split the data into train and test sets in a 75:25 ratio using `train_test_split()`. The ratio can be specified using the `test_size` parameter and `random_state` is used as a seed value for reproducing the same data split each time. If `random_state` is `None`, then it will randomly split the records each time, which will give different performance measures:

```
# Lets import the train_test_split method
from sklearn.model_selection import train_test_split

# Distribute the features(X) and labels(y) into two parts training
and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)
```

In the preceding code block, we have divided the data into two parts – train and test sets – in a 75:25 or 3:1 ratio.

3. Let's import the `LinearRegression` model, create its object, and fit it to the training dataset (`X_train`, `y_train`). After fitting the model, we can predict the values for testing data (`X_test`). We can see the intercept and coefficient of the regression equation using the `intercept_` and `coef_` attributes:

```
# Import linear regression model
from sklearn.linear_model import LinearRegression
# Create linear regression model
lin_reg = LinearRegression()

# Fit the linear regression model
lin_reg.fit(X_train, y_train)

# Predict the values given test set
predictions = lin_reg.predict(X_test)

# Print the intercept and coefficients
print("Intercept:",lin_reg.intercept_)
print("Coefficients:",lin_reg.coef_)
```

This results in the following output:

```
Intercept: 2.8925700511511483
Coefficients: [0.04416235 0.19900368 0.00116268]
```

In the preceding code, we have prepared the linear regression model, performed the predictions on test sets, and displayed the intercepts and coefficients. In the upcoming section, we will assess the regression model's performance using model evaluation measures such as R-squared and error functions.

Evaluating regression model performance

In this section, we will review the regression evaluation measures for understanding the performance level of a regression model. Model evaluation is one of the key aspects of any machine learning model building process. It helps us to assess how our model will perform when we put it into production. We will use the following metrics for model evaluation:

- **R-squared**
- **MSE**
- **MAE**
- **RMSE**

R-squared

R-squared (or coefficient of determination) is a statistical model evaluation measure that assesses the goodness of a regression model. It helps data analysts to explain model performance compared to the base model. Its value lies between 0 and 1. A value near 0 represents a poor model while a value near 1 represents a perfect fit. Sometimes, R-squared results in a negative value. This means your model is worse than the average base model. We can explain R-squared using the following formula:

$$R - Square = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$$

Let's understand all the components one by one:

- **Sum of Squares Regression** (**SSR**): This estimates the difference between the forecasted value and the mean of the data.
- **Sum of Squared Errors** (**SSE**): This estimates the change between the original or genuine value and the forecasted value.
- **Total Sum of Squares** (**SST**): This is the change between the original or genuine value and the mean of the data.

MSE

MSE is an abbreviation of mean squared error. It is explained as the square of change between the original and forecasted values and the average between them for all the values:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \quad (y - \hat{y})^2$$

Here, y is the original value and $\overline{y}$ is the forecasted value.

MAE

MAE is an abbreviation of mean absolute error. It is explained as the absolute change between the original and forecasted values and the average between them for all the values:

$$MAE = \frac{1}{n}\sum_{i=1}^{n} \quad |y - \hat{y}|$$

Here, y is the original value, and $\overline{y}$ is the forecasted value.

RMSE

RMSE is an abbreviation of root mean squared error. It is explained as the square root of MSE:

$$RMSE = \sqrt{MSE}$$

Let's evaluate the model performance on a testing dataset. In the previous section, we predicted the values for the test set. Now, we will compare the predicted values with the actual values of the test set (`y_test`). scikit-learn offers the `metrics` class for evaluating the models. For regression model evaluation, we have methods for R-squared, MSE, MAE, and RMSE. Each of the methods takes two inputs: the actual values of the test set and the predicted values (`y_test` and `y_pred`). Let's assess the performance of the linear regression model:

```
# Import the required libraries
import numpy as np
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score

# Evaluate mean absolute error
print('Mean Absolute Error(MAE):', mean_absolute_error(y_test,predictions))

# Evaluate mean squared error
print("Mean Squared Error(MSE):", mean_squared_error(y_test, predictions))

# Evaluate root mean squared error
print("Root Mean Squared Error(RMSE):", np.sqrt(mean_squared_error(y_test,
predictions)))

# Evaluate R-square
print("R-Square:",r2_score(y_test, predictions))
```

This results in the following output:

```
Mean Absolute Error(MAE): 1.300032091923545
Mean Squared Error(MSE): 4.0124975229171
Root Mean Squared Error(RMSE): 2.003121944095541
R-Square: 0.8576396745320893
```

In the example, we have evaluated the linear regression model using MAE, MSE, RMSE, and R-squared. Here, R-squared is 0.85, which indicates that the model explains the 85% variability of the data.

Fitting polynomial regression

Polynomial regression is a type of regression analysis that is used to adapt the nonlinear relationships between dependent and independent variables. In this type of regression, variables are modeled as the *n*th polynomial degree. It is used to understand the growth rate of various phenomena, such as epidemic outbreaks and growth in sales. Let's understand the equation of polynomial regression:

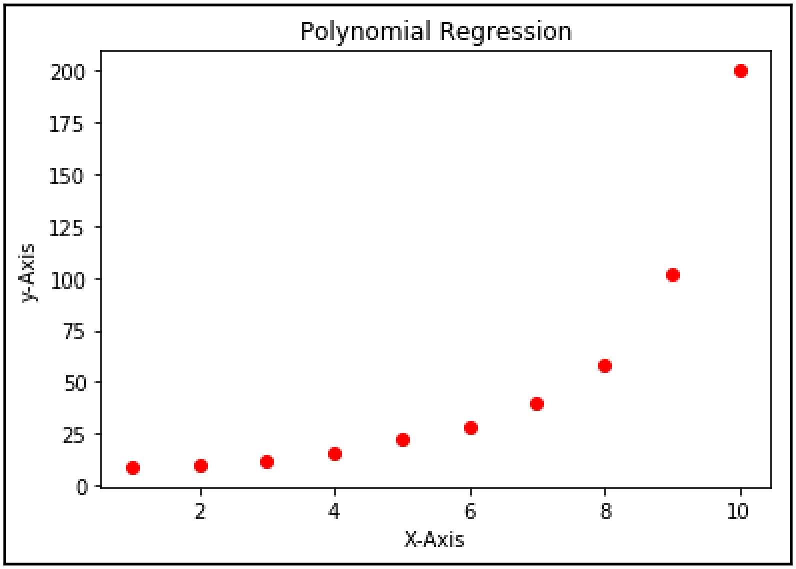
$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon.$$

Here, x is the independent variable and y is a dependent variable. The β_0 intercepts, β_1,β_2... β_n, are a coefficient of *x* and ε (the Greek letter pronounced as epsilon) is an error term that will act as a random variable.

Let's see an example to understand the polynomial concept in detail:

```
# import libraries
import matplotlib.pyplot as plt
import numpy as np

# Create X and Y lists
X=[1,2,3,4,5,6,7,8,9,10]
y=[9,10,12,16,22,28,40,58,102,200]

# Plot scatter diagram
plt.scatter(X,y, color = 'red')
plt.title('Polynomial Regression')
plt.xlabel('X-Axis')
plt.ylabel('y-Axis')
```

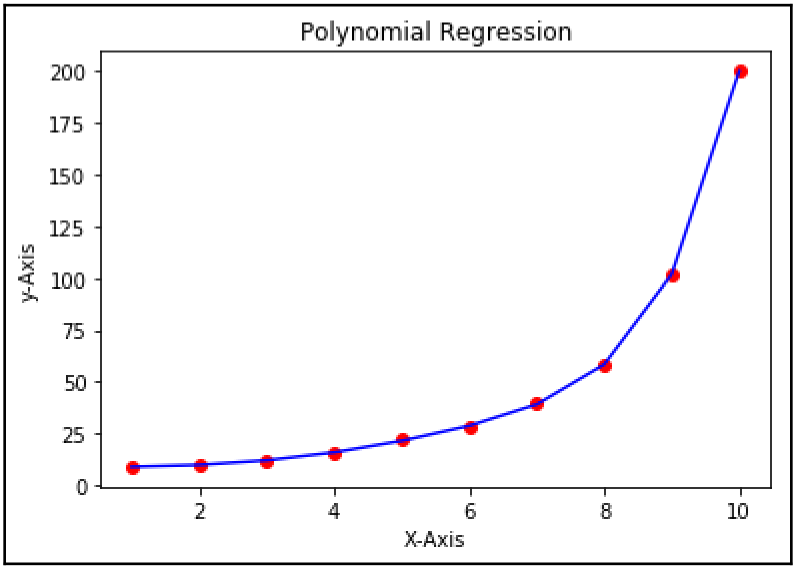
This results in the following output:



In the preceding code, we have displayed a dataset that has a polynomial relationship. Let's see how we can map this relationship in regression analysis:

```
# import libraries
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Prepare dataset
data = pd.DataFrame({"X":[1,2,3,4,5,6,7,8,9,10],
"y":[9,10,12,16,22,28,40,58,102,200]})

X = data[['X']] y = data[['y']]

# Apply Polynomial Features
polynomial_reg = PolynomialFeatures(degree = 6)
X_polynomial = polynomial_reg.fit_transform(X)

# Apply Linear Regression Model
linear_reg = LinearRegression()
linear_reg.fit(X_polynomial, y)
predictions=linear_reg.predict(X_polynomial)

# Plot the results
plt.scatter(X,y, color = 'red')
plt.plot(X, predictions, color = 'red')
plt.title('Polynomial Regression')
plt.xlabel('X-Axis')
plt.ylabel('y-Axis')
```

This results in the following output:



In the preceding code, we have read the polynomial relationship dataset, converted the *X* column into a polynomial *n*th degree column using `PolynomialFeatures()`, and then applied linear regression on `X_polynomial` and `label`. The preceding output plot shows that the resultant model captures the performance. Now, it's time to jump to another type of regression model, which can be used for classification purposes.

Regression models for classification

Classification is the most utilized technique in the area of machine and statistical learning. Most machine learning problems are classification problems, such as detecting spam emails, analyzing financial risk, churn analysis, and discovering potential customers.

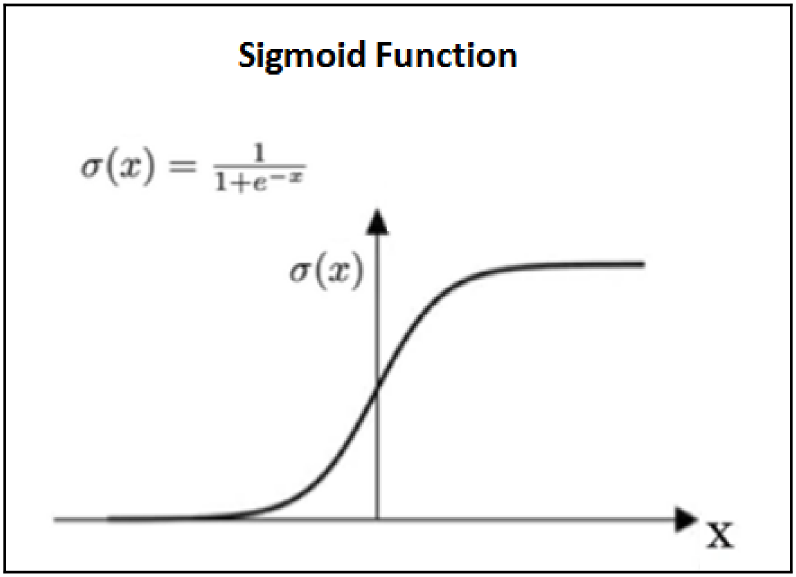
Classification can be of two types: binary and multi-class classification. Binary classification target variables have only two values: either 0 and 1 or yes or no. Examples of binary classification are whether a customer will buy an item or not, whether the customer will switch or churn to another brand or not, spam detection, disease prediction, and whether a loan applicant will default or not. Multi-class classification has more than two classes, for example, for categories of news articles, the classes could be sports, politics, business, and many more.

Logistic regression is one of the classification methods, although its name ends with regression. It is a commonly used binary class classification method. It is a basic machine learning algorithm for all kinds of classification problems. It finds the association between dependent (or target) variables and sets of independent variables (or features). In the next section, we will look at logistic regression in detail.

Logistic regression

Logistic regression is a kind of supervised machine learning algorithm that is utilized to forecast a binary outcome and classify observations. Its dependent variable is a binary variable with two classes: 0 or 1. For example, it can be used to detect whether a loan applicant will default or not. It is a unique type of regression where the dependent or target variable is binary. It computes a log of the odds ratio of the target variable, which represents the probability of occurrence of an event, for example, the probability of a person suffering from diabetes.

Logistic regression is a kind of simple linear regression where the dependent or target variable is categorical. It uses the sigmoid function on the prediction result of linear regression. We can also use the logistic regression algorithm for multiple target classes. For multiple-class problems, it is called multinomial logistic regression. Multinomial logistic regression is a modification of logistic regression; it uses the softmax function instead of the sigmoid activation function:



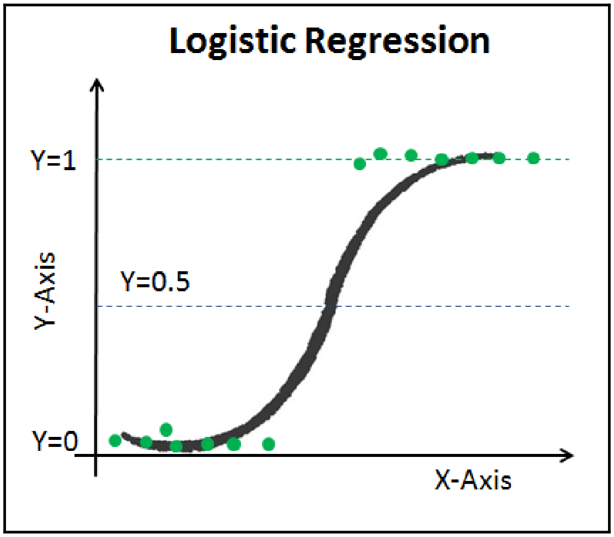
The sigmoid function is also known as a logistic function or an S-shaped curve. It maps input values between the ranges 0 and 1, which represents the probability of occurrence of an event. If the curve moves toward positive infinity, then the outcome becomes 1 and if the curve moves toward negative infinity, then the outcome becomes 1. Let's see the formula for the sigmoid function and logistic regression equation:

$$f(x) = \frac{1}{1 + exp(-x)}$$

The following formula shows the logistic regression equation:

$$log(\frac{P}{1 - P}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

The term in the `log()` function is known as an odds ratio or "odds." The odds ratio is the ratio of the probability of the occurrence of an event to the probability of not occurrence of an event. In the following graph, you can see how logistic regression output behaves:



We can see the ratio lands roughly around 0.5 here. Let's explore logistic regression a bit more in the upcoming subsections.

Characteristics of the logistic regression model

In this subsection, we will focus on the basic characteristics and assumptions of logistic regression. Let's understand the following characteristics:

- The dependent or target variable should be binary in nature.
- There should be no multicollinearity among independent variables.
- Coefficients are estimated using maximum likelihood.
- Logistic regression follows Bernoulli distribution.
- There is no R-squared for model evaluation. The model was evaluated using concordance, KS statistics.

Types of logistic regression algorithms

There are various types of logistic regression algorithms available for different use cases and scenarios. In this section, we will focus on binary, multinomial, and ordinal logistic regression. Let's see each of them and understand where we can utilize them:

- **Binary logistic regression model**:

  In the binary logistic regression model, the dependent or target column has only two values, such as whether a loan will default or not default, an email is spam or not spam, or a patient is diabetic or non-diabetic.

- **Multinomial logistic regression model**:

  In a multinomial logistic regression model, a dependent or target column has three or more than three values, such as predicting the species of the iris flower and predicting the category of news articles, such as politics, business, and sports.

- **Ordinal logistic regression**:

  In the ordinal logistic regression model, a dependent variable will have ordinal or sequence classes, such as movie and hotel ratings.

Advantages and disadvantages of logistic regression

The logistic regression model not only provides prediction (0 or 1) but also gives the probabilities of outcomes, which helps us to understand the confidence of a prediction. It is easy to implement and understand and is interpretable.

A large number of independent variables will increase the amount of variance explained, which results in model overfitting. Logistic regression cannot work with non-linear relationships. It will also not perform well with highly correlated feature variables (or independent variables).

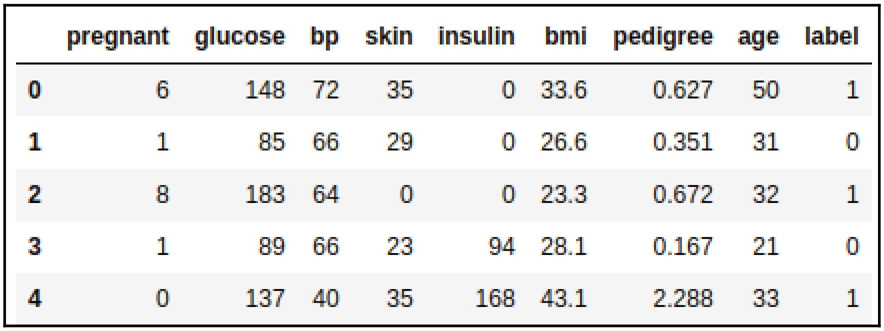
Implementing logistic regression using scikit-learn

Now that you know all about logistic regression, let's implement it in Python using the `scikit-learn` library. Let's create a model using naive Bayes classification. We will do so using the following steps:

1. We will first import the dataset and the required libraries using the following code:

```
# Import libraries
import pandas as pd
# read the dataset
diabetes = pd.read_csv("diabetes.csv")

# Show top 5-records
diabetes.head()
```

This results in the following output:



	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In our preceding example, we are reading the Pima Indians Diabetes dataset. This dataset does not give the column names, so we have to do so.

2. In the `read_csv()` function, we will pass the header to `None` and names to the column list that was created before reading the CSV file:

```
# Split dataset in two parts: feature set and target label
feature_set = ['pregnant', 'insulin', 'bmi',
'age','glucose','bp','pedigree']

features = diabetes[feature_set]

target = diabetes.label
```

```
# Partition data into training and testing set
from sklearn.model_selection import train_test_split
feature_train, feature_test, target_train, target_test =
train_test_split(features, target, test_size=0.3, random_state=1)
```

After loading the dataset, we need to divide the dataset into independent (feature set) column features and dependent (or label) column targets. After this, the dataset will be partitioned into training and testing sets. Now, both the dependent and independent columns are divided into train and test sets (`feature_train`, `feature_test`, `target_train`, and `target_test`) using `train_test_split()`. `train_test_split()` takes dependent and independent DataFrames, `test_size` and `random_state`. Here, `test_size` will decide the ratio of the train-test split (that is, a `test_size` value of `0.3` means 30% testing set and the remaining 70% will be the training set), and `random_state` is used as a seed value for reproducing the same data split each time. If `random_state` is `None`, then it will randomly split the records each time, which will give different performance measures:

```
# import logistic regression scikit-learn model
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# instantiate the model
logreg = LogisticRegression(solver='lbfgs')

# fit the model with data
logreg.fit(feature_train,target_train)

# Forecast the target variable for given test dataset
predictions = logreg.predict(feature_test)

# Assess model performance using accuracy measure
print("Logistic Regression Model
Accuracy:",accuracy_score(target_test, predictions))
```

This results in the following output:

```
Logistic Regression Model Accuracy: 0.7835497835497836
```

Now, we are ready to create a logistic regression model. First, we will import the `LogisticRegression` class and create its object or model. This model will fit on the training dataset (`X_train` and `y_train`). After training, the model is ready to make predictions using the `predict()` method. scikit-learn's `metrics` class offers various methods for performance evaluation, such as accuracy. The `accuracy_score()` methods will take actual labels (`y_test`) and predicted labels (`y_pred`).

Summary

In this chapter, we discovered regression analysis algorithms. This will benefit you in gaining an important skill for predictive data analysis. You have gained an understanding of concepts such as regression analysis, multicollinearity, dummy variables, regression evaluation measures, and logistic regression. The chapter started with simple linear and multiple regressions. After simple linear and multiple regressions, our main focus was on multicollinearity, model development, and model evaluation measures. In later sections, we focused on logistic regression, characteristics, types of regression, and its implementation.

The next chapter, `Chapter 10`, *Supervised Learning – Classification Techniques*, will focus on classification, its techniques, the train-test split strategy, and performance evaluation measures. In later sections, the focus will be on data splitting, the confusion matrix, and performance evaluation measures such as accuracy, precision, recall, F1-score, ROC, and AUC.

10

Supervised Learning - Classification Techniques

Most real-world machine learning problems use supervised learning. In supervised learning, the model will learn from a labeled training dataset. A label is a target variable that we want to predict. It is an extra piece of information that helps in making decisions or predictions, for example, which loan application is safe or risky, whether a patient suffers from a disease or not, house prices, and credit eligibility scores. These labels act as a supervisor or teacher for the learning process. Supervised learning algorithms can be of two types: classification or regression. A classification problem has a categorical target variable, such as a loan application status as safe or risky, whether a patient suffers from a "disease" or "not disease," or whether a customer is "potential" or "not potential."

This chapter focuses on supervised machine learning, and specifically covers classification techniques. This chapter will mostly be using scikit-learn. It will delve into basic techniques of classification, such as naive Bayes, **Support Vector Machines** (**SVMs**), **K-Nearest Neighbor** (**KNN**), and decision trees. Also, it focuses on train-test split strategies and model evaluation methods and parameters.

The topics of this chapter are listed as follows:

- Classification
- Naive Bayes classification
- Decision tree classification
- KNN classification
- SVM classification
- Splitting training and testing sets
- Evaluating the classification model performance
- ROC curve and AUC

Technical requirements

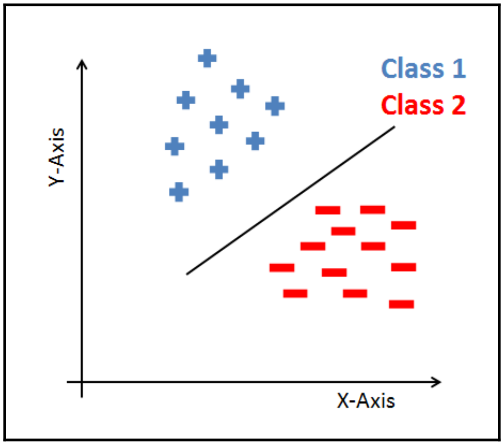
This chapter has the following technical requirements:

- You can find the code and the datasets at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter10`.
- All the code blocks are available in the `ch10.ipynb` file.
- This chapter uses only one CSV file (`diabetes.csv`) for practice purposes.
- In this chapter, we will use the `pandas` and `scikit-learn` Python libraries.

Classification

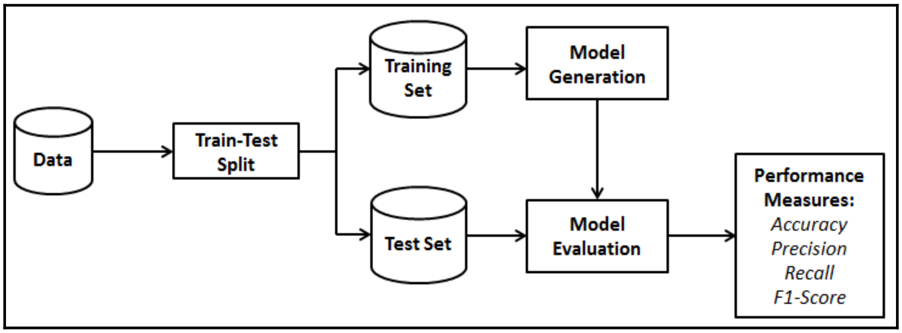
As a healthcare data analyst, your job is to identify patients or sufferers that have a higher chance of a particular disease, for example, diabetes or cancer. These predictions will help you to treat patients before the disease occurs. Similarly, a sales and marketing manager wants to predict potential customers who have more of a chance of buying a product. This is the process of categorizing customers into two or more categories known as classification. The classification model predicts the categorical class label, such as whether the customer is potential or not. In the classification process, the model is trained on available data, makes predictions, and evaluates the model performance. Developed models are called classifiers. This means it has three stages: training, prediction, and evaluation. The trained model is evaluated using parameters such as accuracy, precision, recall, F1-score, and **Area Under Curve** (**AUC**). Classification has a variety of applications in various domains, such as banking, finance, citizen services, healthcare, text analysis, image identification, and object detection.

As an analyst, you have to first define the problem that you want to solve using classification and then identify the potential features that predict the labels accurately. Features are the columns or attributes that are responsible for prediction. In diabetes prediction problems, health analysts will collect patient information, such as age, exercise routine, junk food-eating habits, alcohol consumption, and smoking habit characteristics or features. These features will be used to predict whether the patient will suffer from diabetes. You can see in the following diagram how data can be classified into two classes using a line:



Machine learning and data mining processes have various steps: data collection, data preprocessing, train-test split, model generation, and evaluation. We have seen data analysis models such as KDD, SEMMA, and CRISP-DM. In classification, we only focus on the train-test split, model generation, and evaluation.

The classification model has three stages: train-test split, model generation, and model evaluation. In the train-test split stage, data is divided into two parts: training and testing sets. In training, the training set is used to generate the model, and testing is used in the model evaluation stage to assess the model's performance using evaluation metrics such as accuracy, error, precision, and recall. You can see the classification process in the following diagram:



In the preceding diagram, steps for the classification process are presented. Now that we understand the classification process, it's time to learn the classification techniques. In the next section, we will focus on the naive Bayes classification algorithm.

Naive Bayes classification

Naive Bayes is a classification method based on the Bayes theorem. Bayes' theorem is named after its inventor, the statistician Thomas Bayes. It is a fast, accurate, robust, easy-to-understand, and interpretable technique. It can also work faster on large datasets. Naive Bayes is effectively deployed in text mining applications such as document classification, predicting sentiments of customer reviews, and spam filtering.

The naive Bayes classifier is called naive because it assumes class conditional independence. Class conditional independence means each feature column is independent of the remaining other features. For example, in the case of determining whether a person has diabetes or not, it depends upon their eating habits, their exercise routine, the nature of their profession, and their lifestyle. Even if features are correlated or depend on each other, naive Bayes will still assume they are independent. Let's understand the Bayes theorem formula:

$$p(y|X) = \frac{p(X|y).p(y)}{p(X)}$$

Here, y is the target and X is the set of features. $p(y)$ and $p(X)$ are the prior probabilities regardless of evidence. This means the probability of events before evidence is seen. $p(y|X)$ is the posterior probability of event X after evidence is seen. It is the probability of y given evidence X. $p(X|y)$ is the posterior probability of event y after evidence is seen. It is the probability of X given evidence y. Let's take an example of the preceding equation:

$$p(diabetes = \text{"}yes\text{"}|smoking\_freq = \text{"}high\text{"}) = \frac{p(diabetes = \text{"}yes\text{"}).p(diabetes = \text{"}yes\text{"})}{p(smoking\_freq = \text{"}high\text{"})}$$

Here, we are finding the probability of a patient who will suffer from diabetes based on their smoking frequency using Bayes' theorem.

Let's see the working of the naive Bayes classification algorithm. Assume that dataset D has X features and label y. Features can be n-dimensional, X=$X1$, $X2$, $X3$... Xn. Label y may have m classes, $C1$, $C2$, $C3$...Cm. It will work as follows:

1. Calculate the prior probabilities, $p(y)$ and $p(X)$, for the given class labels.
2. Calculate the posterior probabilities, $p(y|X)$and $p(X|y)$, with each attribute for each class:

$$p(X|y) = \prod_{k=1}^{n} \quad p(X_k|\, y_i)$$

3. Multiply the same class posterior probability, $p(X|y)$:

$$p(X|y_i) = p(X_1|\, y_i) * \; p(X_2|\, y_i) * \; p(X_3|\, y_i) \ldots\, p(X_n|\, y_i)$$

4. If the attribute is categorical then there should be several records of class y_i in with x_k value, divided by y_i records in the dataset.
5. If the attribute is continuous, then it is calculated using Gaussian distribution:

$$p(X|y_i) = g\left(x_k, \mu_{y_i}, \sigma_{y_i}\right) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

6. Multiply the prior probability, *p*(*y*), by the posterior probability from *step* 3:

$$p(X) = p(y_i) * p(y_i)$$

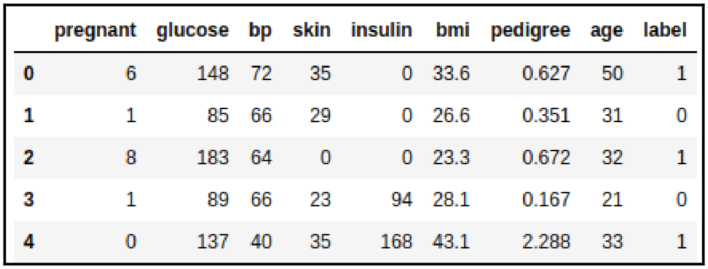
7. Find the class with the maximum probability for the given input feature set. This class will be our final prediction.

Now, let's create a model using naive Bayes classification in Python:

1. Load the Pima Indians Diabetes dataset (`https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/blob/master/Chapter09/diabetes.csv`) using the following lines of code:

```
# Import libraries
import pandas as pd
# read the dataset
diabetes = pd.read_csv("diabetes.csv")

# Show top 5-records
diabetes.head()
```

This results in the following output:



	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

We have thus imported `pandas` and read the dataset. In the preceding example, we are reading the Pima Indians Diabetes dataset.

2. We will now split the dataset into two parts, as follows:

```
# split dataset in two parts: feature set and target label
feature_set = ['pregnant', 'insulin', 'bmi',
'age','glucose','bp','pedigree'] features = diabetes[feature_set]
target = diabetes.label

# partition data into training and testing set
from sklearn.model_selection import train_test_split

feature_train,feature_test, target_train, target_test = \
train_test_split(features, target, test_size=0.3, random_state=1)
```

After loading the dataset, we divide the dataset into a dependent or label column (`target`) and independent or feature columns (`feature_set`). After this, the dataset will be broken up into train and test sets. Now, both the dependent and independent columns are broken up into train and test sets (`feature_train`, `feature_test`, `target_train`, and `target_test`) using `train_test_split()`. `train_test_split()` takes dependent and independent DataFrames, `test_size` and `random_state`. Here, `test_size` will decide the ratio of the train-test split (that is, `test_size 0.3` means 30% is the testing set and the remaining 70% of data will be the training set), and `random_state` is used as a seed value for reproducing the same data split each time. If `random_state` is `None`, then it will randomly split the records each time, which will give different performance measures.

3. We will now build the naive Bayes classification model:

```
# Import Gaussian Naive Bayes model
from sklearn.naive_bayes import GaussianNB
```

```
# Create a Gaussian Classifier
model = GaussianNB()

# Train the model using the training sets
model.fit(feature_train,target_train)

# Forecast the target variable for given test dataset
predictions = model.predict(feature_test)
```

Here, we have created a naive Bayes model. First, we will import the `GaussianNB` class and create its object or model. This model will fit on the training dataset (`feature_train`, `target_train`). After training, the model is ready to make predictions using the `predict()` method.

4. Finally, we will evaluate the model's performance:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
# Calculate model accuracy
print("Accuracy:",accuracy_score(target_test, predictions))

# Calculate model precision
print("Precision:",precision_score(target_test, predictions))

# Calculate model recall
print("Recall:",recall_score(target_test, predictions))

# Calculate model f1 score
print("F1-Score:",f1_score(target_test, predictions))
```

This results in the following output:

```
Accuracy: 0.7748917748917749
Precision: 0.7391304347826086
Recall: 0.6
F1-Score: 0.6623376623376623
```

scikit-learn's `metrics` class offers various methods for performance evaluation, for example, accuracy, precision, recall, and F1-score metrics. These methods will take actual target labels (`target_test`) and predicted labels (`predictions`). We will understand these metrics in detail in the *Evaluating the classification model performance* section.

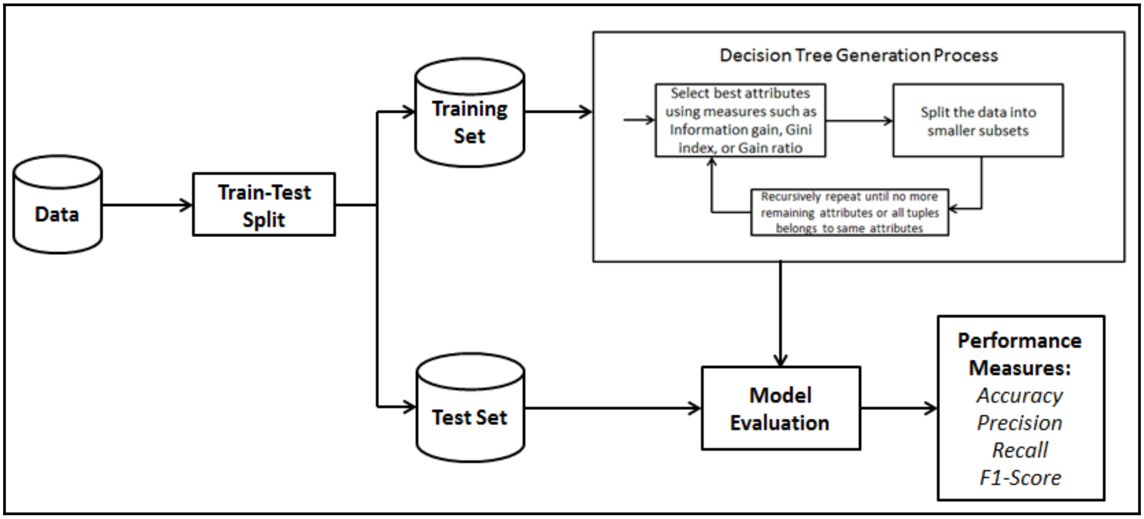
Naive Bayes is a simple, fast, accurate, and easy-to-understand method for prediction. It has a lower computation cost and can work with large datasets. Naive Bayes can also be employed in multi-class classification problems. The naive Bayes classifier performs better compared to logistic regression when data has a class independence assumption.

Naive Bayes suffers from the **zero frequency problem**. Zero frequency means that if any category in the feature is missing, then it will have a zero frequency count. This problem is solved by Laplacian correction. Laplacian correction (or Laplace transformation) is a kind of smoothing technique that will add one record for each class so that the frequency count for the missing class will become 1, thus probabilities of Bayes' theorem will not be affected. Another issue with naive Bayes is its assumption of class conditional independence, as it is practically impossible for all the predictors to be fully independent. In this section, we have learned about naive Bayes classification. Now it's time to learn about the decision tree classification algorithm.

Decision tree classification

A decision tree is one of the most well-known classification techniques. It can be employed for both types of supervised learning problems (classification and regression problems). It is a flowchart-like tree structure and mimics human-level thinking, which makes it easier to understand and interpret. It also makes you see the logic behind the prediction unlike black-box algorithms such as SVMs and neural networks.

The decision tree has three basic components: the internal node, the branch, and leaf nodes. Here, each terminal node represents a feature, the link represents the decision rule or split rule, and the leaf provides the result of the prediction. The first starting or master node in the tree is the root node. It partitions the data based on features or attributes values. Here, we divide the data and again divide the remaining data recursively until all the items refer to the same class or there are no more columns left. Decision trees can be employed in both types of problems: classification and regression. There are lots of decision tree algorithms available, for example, CART, ID3, C4.5, and CHAID. But here, we are mainly focusing on CART and ID3 because in scikit-learn, these are the two that are available. Let's see the decision tree classifier generation process in the following figure:


Data
Train-Test Split
Training Set
Test Set
Decision Tree Generation Process
Select best attributes using measures such as Information gain, Gini index, or Gain ratio
Split the data into smaller subsets
Recursively repeat until no more remaining attributes or all tuples belongs to same attributes
Model Evaluation
Performance Measures:
Accuracy
Precision
Recall
F1-Score

CART stands for **Classification and Regression Tree**. CART utilizes the Gini index for selecting the best column. The Gini index is the difference between the sum of the squared probabilities of each class from 1. The feature or column with the minimum Gini index value is selected as the splitting or partition feature. The value of the Gini index lies in the range of 0 and 1. If the Gini index value is 0, it indicates that all items belong to one class, and if the Gini index value is exactly 1, it indicates that all the elements are randomly distributed. A 0.5 value of the Gini index indicates the equal distribution of items into some classes:

$$Gini\ Index = \sum_{i=1}^{C} p_i^2$$

ID3 stands for **Iterative Dichotomiser 3**. It uses information gain or entropy as an attribute selection measure. Entropy was invented by Shannon, and it measures the amount of impurity or randomness in a dataset. Information gain measures the variations between entropy before partition and mean entropy after the partition of the dataset for a specific column. The feature or attribute with the largest value of information gain will be selected as a splitting feature or attribute. If entropy is 0, it indicates that there exists only a single class, and if entropy is 1, it indicates that items are equally distributed:

$$Information\ Gain = \sum_{i=1}^{C} p_i \log_2(p_i)$$

The decision tree is very intuitive and easy to understand, interpret, and explain to stakeholders. There is no need to normalize features and distribution-free algorithms. Decision trees are also used to predict missing values. They have the capability to capture non-linear patterns. Decision trees can overfit and are sensitive to noisy data. Decision trees are biased with imbalanced data, which is why before applying decision trees, we should balance out the dataset. Decision trees are more expensive in terms of time and complexity.

Let's work on a decision tree using scikit-learn and perform a prediction dataset. After this, we will be ready for the model building:

1. First, you need to import `pandas` and load the Pimas dataset using the `read_csv()` method that we already saw in the last section.

2. After this, we need to divide the dataset into training and testing datasets similar to what we performed in the preceding section.

3. Now, we will build the decision tree classification model:

```
# Import Decision Tree model
from sklearn.tree import DecisionTreeClassifier

# Create a Decision Tree classifier object
clf = DecisionTreeClassifier()

# Train the model using training dataset
clf = clf.fit(feature_train,target_train)

# Predict the response for test dataset
predictions = clf.predict(feature_test)
```

   Here, we have created a decision tree model. First, we will import the `DecisionTreeClassifier` class and create its object or model. This model will fit on the training dataset (`feature_train`, `target_train`). After training, the model is ready to make predictions using the `predict()` method.

4. We will now evaluate the model's performance:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

# Calculate model accuracy
print("Accuracy:",accuracy_score(target_test, predictions))
```

```
# Calculate model precision
print("Precision:",precision_score(target_test, predictions))

# Calculate model recall
print("Recall:",recall_score(target_test, predictions))

# Calculate model f1 score
print("F1-Score:",f1_score(target_test, predictions))
```

This results in the following output:

```
Accuracy: 0.7229437229437229
Precision: 0.6438356164383562
Recall: 0.5529411764705883
F1-Score: 0.5949367088607594
```

In the preceding example, model performance is assessed using accuracy, precision, recall, and F1-score.

After getting a full understanding of decision trees, let's move on to the KNN classification.

KNN classification

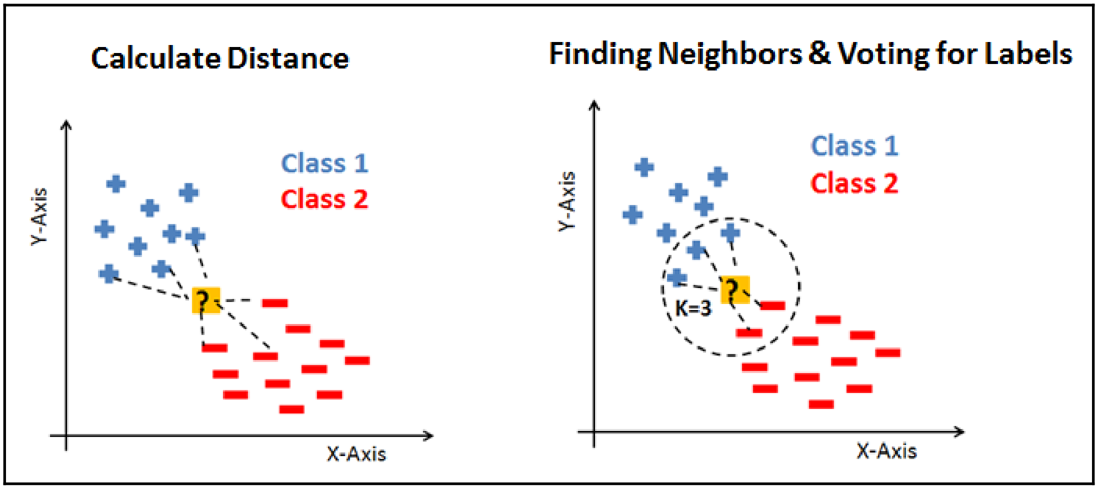
KNN is a simple, easy-to-comprehend, and easy-to-implement classification algorithm. It can also be used for regression problems. KNN can be employed in lots of use cases, such as item recommendations and classification problems. Specifically, it can suggest movies on Netflix, articles on Medium, candidates on naukari.com, products on eBay, and videos on YouTube. In classification, it can be used to classify instances such as, for example, banking institutes that can classify the loan of risky candidates, or political scientists can classify potential voters.

KNN has three basic properties, which are non-parametric, lazy learner, and instance-based learning. Non-parametric means the algorithm is distribution-free and there is no need for parameters such as mean and standard deviation. Lazy learner means KNN does not train the model; that is, the model is trained in the testing phase. This makes for faster training but slower testing. It is also more time- and memory-consuming. Instance-based learning means the predicted outcome is based on the similarity with its nearest neighbors. It does not create any abstract equations or rules for prediction; instead, it stores all the data and queries each record.

The KNN classification algorithm finds the *k* most similar instances from the training dataset and the majority decides the predicted label of the given input features. The following steps will be performed by the KNN classifier to make predictions:

1. Compute the distance for an input observation with all the observations in the training dataset.
2. Find the *K* top closest neighbors by sorting the distance with all the instances in ascending order.
3. Perform voting on the *K* top closest neighbors and predict the label with the majority of votes.

   This is better represented using the following diagram:



Let's work on a KNN classifier using scikit-learn and perform a prediction on a dataset:

1. Load the Pima Indians Diabetes dataset.

   First, you need to import `pandas` and load the dataset using the `read_csv()` method that we have already seen in the *Naive Bayes classification* session.

2. Split the dataset.

   After this, we need to break down the dataset into two sets – a training and a testing set – as we did in the *Naive Bayes classification* section.

3. Build the KNN classification model.

Now, we are ready for the model building:

```
# Import KNN model
from sklearn.neighbors import KNeighborsClassifier

# Create a KNN classifier object
model = KNeighborsClassifier(n_neighbors=3)

# Train the model using the training dataset
model.fit(feature_train,target_train)

# Predict the target variable for test dataset
predictions = model.predict(feature_test)
```

In the preceding code block, we imported the `KNeighborsClassifier` class and created its object or model. Here, we have taken 3 neighbors as an input parameter to the model. If we do not specify the number of neighbors as an input parameter, then the model will choose 5 neighbors by default. This model will fit on the training dataset (`feature_train`, `target_train`). After training, the model is ready to make predictions using the `predict()` method.

4. Evaluate the model's performance:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

# Calculate model accuracy
print("Accuracy:",accuracy_score(target_test, predictions))

# Calculate model precision
print("Precision:",precision_score(target_test, predictions))

# Calculate model recall
print("Recall:",recall_score(target_test, predictions))

# Calculate model f1 score
print("F1-Score:",f1_score(target_test, predictions))
```

This results in the following output:

```
Accuracy: 0.7532467532467533
Precision: 0.7058823529411765
Recall: 0.5647058823529412
F1-Score: 0.6274509803921569
```

In the preceding example, model performance is assessed using accuracy, precision, recall, and F1-score.

After understanding the KNN classification algorithm, it's time to learn about the SVM classification algorithm.

SVM classification

SVMs are the most preferred and favorite machine learning algorithms by many data scientists due to their accuracy with less computation power. They are employed for both regression and classification problems. They also offer a kernel trick to model non-linear relationships. SVM has a variety of use cases, such as intrusion detection, text classification, face detection, and handwriting recognition.

SVM is a discriminative model that generates optimal hyperplanes with a large margin in n-dimensional space to separate data points. The basic idea is to discover the **Maximum Marginal Hyperplane** (**MMH**) that perfectly separates data into given classes. The maximum margin means the maximum distance between data points of both classes.

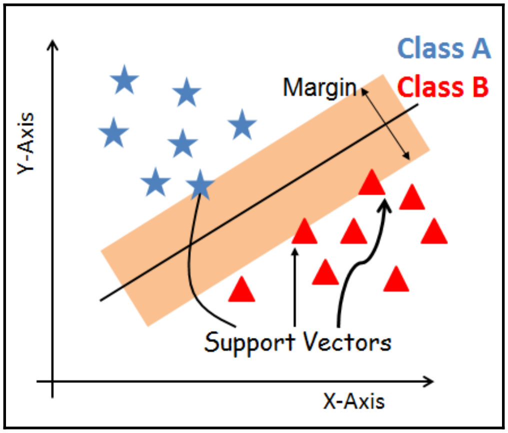
Terminology

We will now explore some of the terminology that goes into SVM classification:

- **Hyperplane**: Hyperplane is a decision boundary used to distinguish between two classes. Hyperplane dimensionality is decided by the number of features. It is also known as a decision plane.
- **Support vectors**: Support vectors are the closest points to the hyperplane and help in the orientation of the hyperplane by maximizing the margin.
- **Margin**: Margin is the maximum gap between the closest points. The larger the margin, the better the classification is considered. The margin can be calculated by the perpendicular distance from the support vector line.

The core objective of an SVM is to choose the hyperplane with the largest possible boundary between support vectors. The SVM finds the MMH in the following two stages:

1. Create hyperplanes that separate the data points in the best possible manner.
2. Select the hyperplane with maximum margin hyperplane:



The SVM algorithm is a faster and more accurate classifier compared to naive Bayes. It performs better with a larger margin of separation. SVM is not favorable for large datasets. Its performance also depends upon the type of kernel used. It performs poorly with overlapping classes.

Let's work on support vector classifiers using `scikit-learn` and perform a prediction dataset. After this, we will divide the dataset into two sets of training and testing sets as we did in the *Naive Bayes classification* section. After this, we are ready with the model building:

1. Load the Pima Indians Diabetes dataset.

   First, you need to import `pandas` and load the dataset using the `read_csv()` method that we already saw in the *Naive Bayes classification* session.

2. Split the dataset.

   After this, we need to break the dataset up into two sets – a training and testing set – as we did in the *naive Bayes classification* section.

3. Build the SVM classification model.

   Now, we are ready with the model building:

```
# Import SVM model
from sklearn import svm

# Create a SVM classifier object
clf = svm.SVC(kernel='linear')

# Train the model using the training sets
```

```
clf.fit(feature_train,target_train)

# Predict the target variable for test dataset
predictions = clf.predict(feature_test)
```

In the preceding code block, we will import the `svm` module and create its `svm.SVC()` object or model. Here, we have passed the `linear` kernel. You can also pass another kernel, such as `poly`, `rbf`, or `sigmoid`. If we don't specify the kernel, then it will select `rbf` by default as the kernel. The linear kernel will create a linear hyperplane to separate diabetic and non-diabetic patients. This model will fit on the training dataset (`feature_train`, `target_train`). After training, the model is ready to make predictions using the `predict()` method.

4. Evaluate the model's performance:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

# Calculate model accuracy
print("Accuracy:",accuracy_score(target_test, predictions))

# Calculate model precision
print("Precision:",precision_score(target_test, predictions))

# Calculate model recall
print("Recall:",recall_score(target_test, predictions))

# Calculate model f1 score
print("F1-Score:",f1_score(target_test, predictions))
```

This results in the following output:

```
Accuracy: 0.7835497835497836
Precision: 0.7868852459016393
Recall: 0.5647058823529412
F1-Score: 0.6575342465753424
```

In the preceding example, model performance will be assessed using metrics such as accuracy, precision, recall, and F1-score. After understanding all these classifiers, it's time to see the training and testing set splitting strategies.

Splitting training and testing sets

Data scientists need to assess the performance of a model, overcome overfitting, and tune the hyperparameters. All these tasks require some hidden data records that were not used in the model development phase. Before model development, the data needs to be divided into some parts, such as train, test, and validation sets. The training dataset is used to build the model. The test dataset is used to assess the performance of a model that was trained on the train set. The validation set is used to find the hyperparameters. Let's look at the following strategies for the train-test split in the upcoming subsections:

- Holdout method
- K-fold cross-validation
- Bootstrap method

Holdout

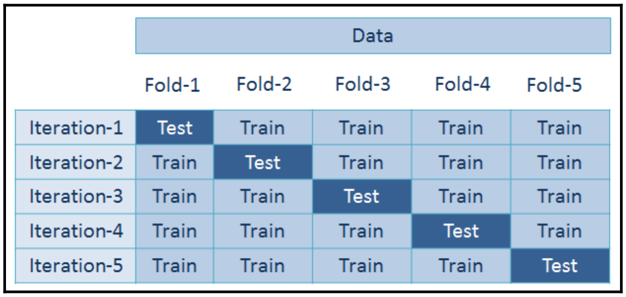
In this method, the dataset is divided randomly into two parts: a training and testing set. Generally, this ratio is 2:1, which means 2/3 for training and 1/3 for testing. We can also split it into different ratios, such as 6:4, 7:3, and 8:2:

```
# partition data into training and testing set
from sklearn.model_selection import train_test_split

# split train and test set
feature_train, feature_test, target_train, target_test =
train_test_split(features, target, test_size=0.3, random_state=1)
```

In the preceding example, `test_size=0.3` represents 30% for the testing set and 70% for the training set. `train_test_split()` splits the dataset into 7:3.

K-fold cross-validation

In this approach, the data is split into *k* partitions of approximately equal size. It will train *k* models and evaluate them using each partition. In each iteration, one partition will hold for testing, and the remaining *k* partitions are collectively used for training purposes. Classification accuracy will be the average of all accuracies. It also ensures that the model is not overfitting:

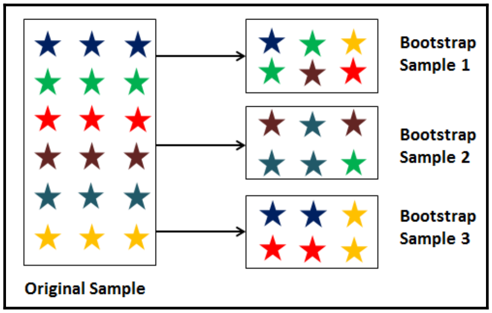


	Data				
	Fold-1	Fold-2	Fold-3	Fold-4	Fold-5
Iteration-1	Test	Train	Train	Train	Train
Iteration-2	Train	Test	Train	Train	Train
Iteration-3	Train	Train	Test	Train	Train
Iteration-4	Train	Train	Train	Test	Train
Iteration-5	Train	Train	Train	Train	Test

In stratified cross-validation, *k* partitions are divided with approximately the same class distribution. This means it preserves the percentages of each class in each partition.

Bootstrap method

Bootstrap is a resampling technique. It performs a sampling iteratively from the dataset with replacement. Sampling with replacement will make random selections. It requires the size of the sample and the number of iterations. In each iteration, it uniformly selects the records. Each record has equal chances of being selected again. The samples that are not selected are known as "out-of-bag" samples. Let's understand bootstrap using the following diagram:



In the preceding diagram, we can see that each element has an equal chance of selection in each bootstrap sample. Let's jump to another important topic of classification, which is classification model evaluation. The next topic helps us to assess the performance of the classification model.

Evaluating the classification model performance

Up to now, we have learned how to create classification models. Creating a machine learning classification model is not enough; as a business or data analyst, you also want to assess its performance so that you can deploy it in live projects.

scikit-learn offers various metrics, such as a confusion matrix, accuracy, precision, recall, and F1-score, to evaluate the performance of a model.

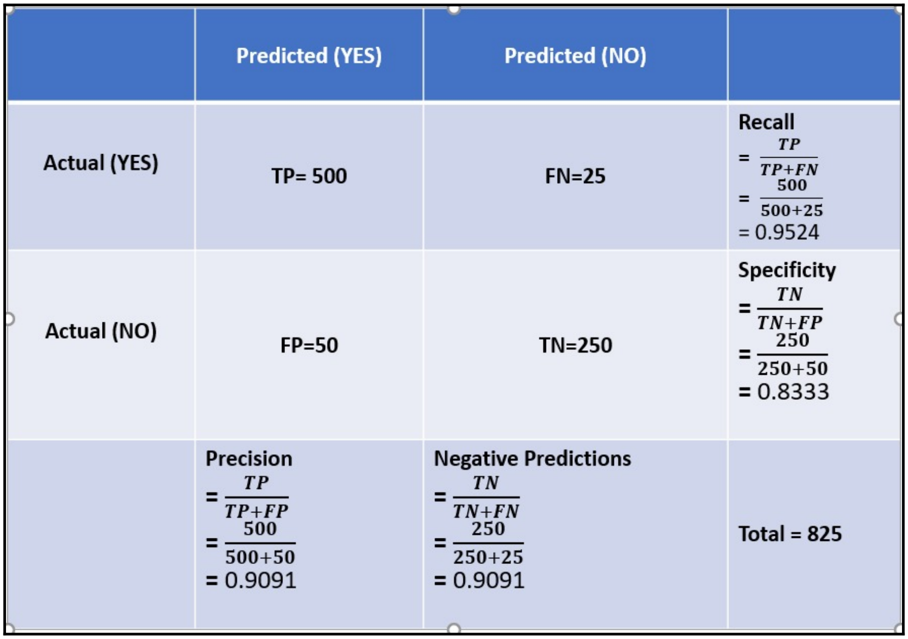
Confusion matrix

A confusion matrix is an approach that gives a brief statement of prediction results on a binary and multi-class classification problem. Let's assume we have to find out whether a person has diabetes or not. The concept behind the confusion matrix is to find the number of right and mistaken forecasts, which are further summarized and separated into each class. It clarifies all the confusion related to the performance of our classification model. This 2x2 matrix not only shows the error being made by our classifier but also represents what sort of mistakes are being made. A confusion matrix is used to make a complete analysis of statistical data faster and also make the results more readable and understandable through clear data visualization. It contains two rows and columns, as shown in the following list. Let's understand the basic terminologies of the confusion matrix:

- **True Positive** (**TP**): This represents cases that are forecasted as `Yes` and in reality, the cases are `Yes`; for example, we have forecasted them as fraudulent cases and in reality, they are fraudulent.
- **True Negative** (**TN**): This represents cases that are forecasted as `No` and in reality, the cases are `No`; for example, we have forecasted them as non-fraudulent cases and in reality, they are non-fraudulent.

- **False Positive** (**FP**): This represents cases that are forecasted as `Yes` and in reality, the cases are `No`; for example, we have forecasted them as fraudulent cases and in reality, they are not fraudulent. This type of incident class represents a Type I error.
- **False Negative** (**FN**): This represents cases that are forecasted as `No` and in reality, the cases are `No`; for example, we have forecasted them as non-fraudulent cases and in reality, they are fraudulent. This type of incident class represents a Type II error.

Let's take an example of a fraud detection problem:



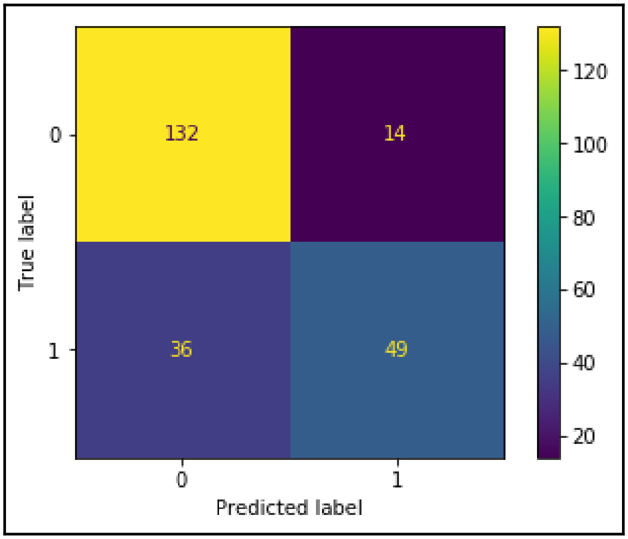
	Predicted (YES)	Predicted (NO)	
Actual (YES)	TP= 500	FN=25	Recall $= \frac{TP}{TP+FN}$ $= \frac{500}{500+25}$ = 0.9524
Actual (NO)	FP=50	TN=250	Specificity $= \frac{TN}{TN+FP}$ $= \frac{250}{250+50}$ = 0.8333
	Precision $= \frac{TP}{TP+FP}$ $= \frac{500}{500+50}$ = 0.9091	Negative Predictions $= \frac{TN}{TN+FN}$ $= \frac{250}{250+25}$ = 0.9091	Total = 825

In the preceding example, we have taken two classes of fraud: **Yes** and **No**. **Yes** indicates fraudulent activity and **No** indicates non-fraudulent activity. The total number of predicted records is **825**, which means 825 transactions were tested. In all these 825 cases, the model or classifier forecasted 550 times **Yes** and 275 times **No**. In reality, actual fraudulent cases are 525 and non-fraudulent cases are 300.

Let's create a confusion matrix in Python using scikit-learn:

```
# Import libraries
import pandas as pd

# read the dataset
diabetes = pd.read_csv("diabetes.csv")

# split dataset in two parts: feature set and target label
feature_set = ['pregnant', 'insulin', 'bmi',
'age','glucose','bp','pedigree']
features = diabetes[feature_set]

target = diabetes.label

# partition data into training and testing set
from sklearn.model_selection import train_test_split
feature_train, feature_test, target_train, target_test =
train_test_split(features, target, test_size=0.3, random_state=1)

# import logistic regression scikit-learn model
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score # for performance evaluation

# instantiate the model
logreg = LogisticRegression(solver='lbfgs')

# fit the model with data
logreg.fit(feature_train,target_train)

# Forecast the target variable for given test dataset
predictions = logreg.predict(feature_test)

# Get prediction probability
predictions_prob = logreg.predict_proba(feature_test)[::,1]

# Import the confusion matrix
from sklearn.metrics import plot_confusion_matrix

# Plot Confusion matrix
plot_confusion_matrix(logreg , feature_test, target_test,
values_format='d')
```

This results in the following output:



In the preceding example, we have loaded the data and divided the data into two parts: training and testing sets. After this, we performed model training using logistic regression as we did in the previous chapter. Here, to plot the confusion matrix, we have used the `plot_confusion_matrix()` method with the model object, testing feature set, testing label set, and `values_format` parameters.

Accuracy

Now, we will find the accuracy of the model calculated from the confusion matrix. It tells us how accurate our predictive model is:

$$Accuracy = \frac{TP + TN}{Total}$$

$$= \frac{500 + 250}{825} = 0.91$$

Precision

When the model predicted Yes, how often was it correct? This is the percentage of positive cases out of the total predicted cases in the dataset. In simple terms, we can understand precision as "Up to what level our model is right when it says it's right":

$$Precision = \frac{TP}{TP+FP} = \frac{TP}{Predicted\ Positives}$$

$$= \frac{500}{500+50} = 0.91$$

Recall

When it is actually Yes, how often did the model predict Yes? This is also known as sensitivity. This is the percentage of positive cases out of all the total actual cases present in the dataset:

$$Recall = \frac{TP}{TP+FN} = \frac{TP}{Actual\ Positives}$$

$$= \frac{500}{500+25} = 0.95$$

F-measure

F-measure is considered as one of the better ways to assess the model. In lots of areas of data science, competition model performance is assessed using F-measure. It is a harmonic mean of precision and recall. The higher the value of the F1-score, the better the model is considered. F1-score provides equal weightage to precision and recall, which means it indicates a balance between both:

$$F_1 Measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

$$= \frac{2 * 0.91 * 0.95}{0.91 + 0.95} = 0.93$$

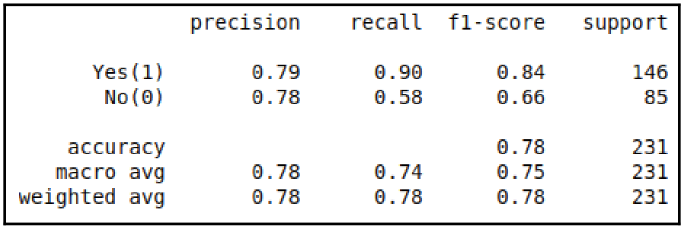
One drawback of F-measure is that it assigns equal weightage to precision and recall but in some examples, one needs to be higher than the other, which is the reason why the F1-score may not be an exact metric.

In the preceding sections, we have seen classification algorithms such as naive Bayes, decision trees, KNN, and SVMs. We have assessed the model performance using scikit-learn's `accuracy_score()` for model accuracy, `precision_score()` for model precision, `recall_score()` for model recall, and `f1_score()` for model F1-score.

We can also print the classification report to dig down into the details to understand the classification model. Let's create the confusion report:

```
# import classification report
from sklearn.metrics import classification_report

# Create classification report
print(classification_report(target_test, predictions,
target_names=['Yes(1)','No(0)']))
```

This results in the following output:

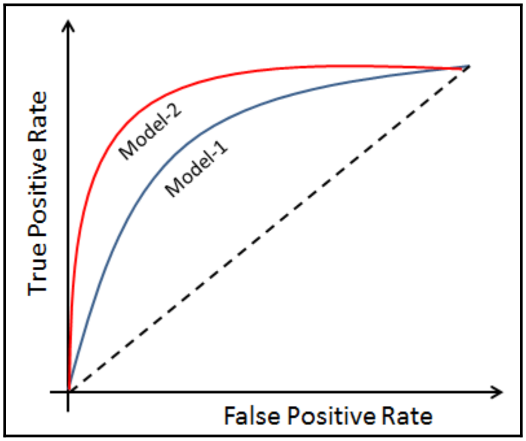


```
              precision    recall  f1-score   support

      Yes(1)       0.79      0.90      0.84       146
       No(0)       0.78      0.58      0.66        85

    accuracy                           0.78       231
   macro avg       0.78      0.74      0.75       231
weighted avg       0.78      0.78      0.78       231
```

In the preceding code, we have printed the confusion matrix report using the `confusion_report()` method with test set labels, prediction set or predicted labels, and target value list parameters.

ROC curve and AUC

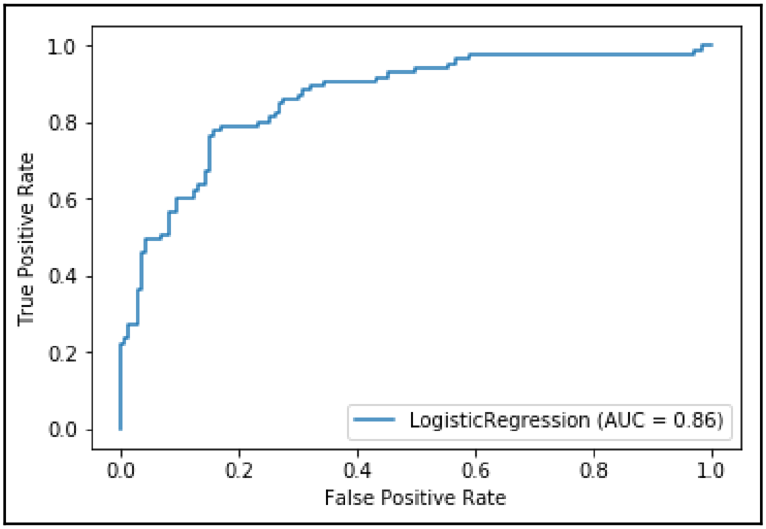
AUC-ROC curve is a tool to measure and assess the performance of classification models. **ROC** (**Receiver Operating Characteristics**) is a pictorial visualization of model performance. It plots a two-dimensional probability plot between the FP rate (or 1-specificity) and the TP rate (or sensitivity). We can also represent the area covered by a model with a single number using AUC:



Let's create the ROC curve using the scikit-learn module:

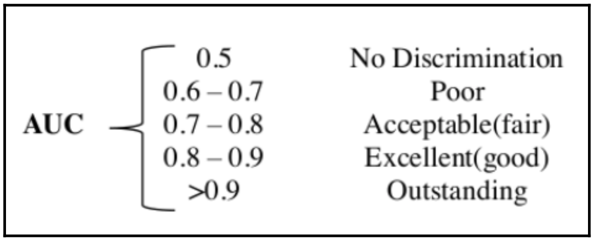
```
# import plot_roc_curve
from sklearn.metrics import plot_roc_curve

plot_roc_curve(logreg , feature_test, target_test)
```

This results in the following output:



In the preceding example, We have drawn the ROC plot `plot_roc_curve()` method with model object, testing feature set, and testing label set parameters.

In the ROC curve, the AUC is a measure of divisibility. It tells us about the model's class distinction capability. The higher the AUC value, the better the model is at distinguishing between "fraud" and "not fraud." For an ideal classifier, the AUC is equal to 1:



AUC	0.5	No Discrimination
	0.6 – 0.7	Poor
	0.7 – 0.8	Acceptable(fair)
	0.8 – 0.9	Excellent(good)
	>0.9	Outstanding

Let's compute an AUC score as follows:

```
# import ROC AUC score
from sklearn.metrics import roc_auc_score

# Compute the area under ROC curve
auc = roc_auc_score(target_test, predictions_prob)

# Print auc value
print("Area Under Curve:",auc)
```

This results in the following output:

```
Area Under Curve: 0.8628525382755843
```

scikit-learn's `metrics` class offers an AUC performance evaluation measure. `roc_auc_score()` methods will take actual labels (`y_test`) and predicted probability (`y_pred_prob`).

Summary

In this chapter, we discovered classification, its techniques, the train-test split strategy, and performance evaluation measures. This will benefit you in gaining an important skill for predictive data analysis. You have seen how to develop linear and non-linear classifiers for predictive analytics using scikit-learn. In the earlier topics of the chapter, you got an understanding of the basics of classification and machine learning algorithms, such as naive Bayes classification, decision tree classification, KNN, and SVMs. In later sections, you saw data splitting approaches and model performance evaluation measures such as accuracy score, precision score, recall score, F1-score, ROC curve, and AUC score.

The next chapter, `Chapter 11`, *Unsupervised Learning – PCA and Clustering*, will concentrate on the important topics of unsupervised machine learning techniques and dimensionality reduction techniques in Python. The chapter starts with dimension reduction and principal component analysis. In the later sections of the chapter, the focus will be on clustering methods such as k-means, hierarchical, DBSCAN, and spectral clustering.

11

Unsupervised Learning - PCA and Clustering

Unsupervised learning is one of the most important branches of machine learning. It enables us to make predictions when we don't have target labels. In unsupervised learning, the model learns only via features because the dataset doesn't have a target label column. Most machine learning problems start with something that helps automate the process. For example, when you want to develop a prediction model for detecting diabetes patients, you need a set of target labels for each patient in your dataset. In the initial stages, arranging target labels for any machine learning problem is not an easy task, because it requires changing the business process to get the labels, whether by manual in-house labeling or collecting the data again with labels.

In this chapter, our focus is on learning about unsupervised learning techniques that can handle situations where target labels are not available. We will especially cover dimensionality reduction techniques and clustering techniques. Dimensionality reduction will be used where we have a large number of features and we want to reduce that amount. This will reduce the model complexity and training cost because it means we can achieve the results we want with just a few features.

Clustering techniques find groups in data based on similarity. These groups essentially represent *unsupervised classification*. In clustering, classes or labels for feature observations are found in an unsupervised manner. Clustering is useful for various business operations, such as cognitive search, recommendations, segmentation, and document clustering.

Here are the topics of this chapter:

- Unsupervised learning
- Reducing the dimensions of data
- Principal component analysis
- Clustering
- Partitioning data using k-means clustering

- Hierarchical clustering
- DBSCAN clustering
- Spectral clustering
- Evaluating clustering performance

Technical requirements

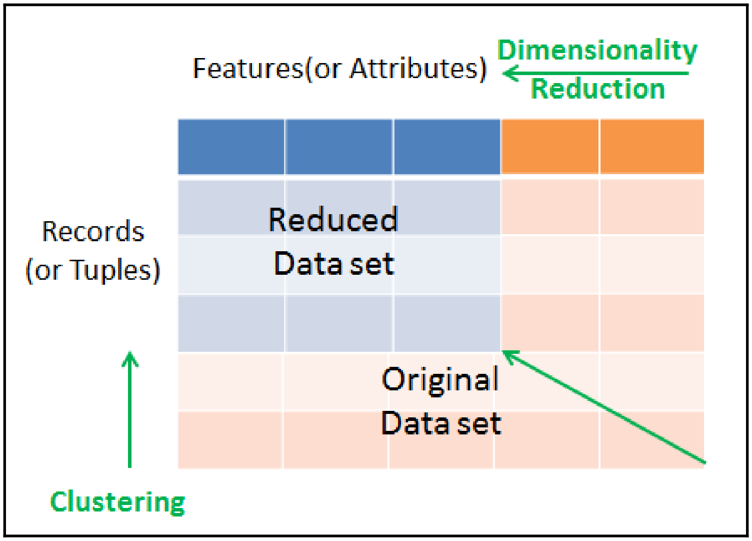
This chapter has the following technical requirements:

- You can find the code and the datasets at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter11`.
- All the code blocks are available in the `ch11.ipynb` file.
- This chapter uses only one CSV file (`diabetes.csv`) for practice purposes.
- In this chapter, we will use the pandas and scikit-learn Python libraries.

Unsupervised learning

Unsupervised learning means learning by observation, not by example. This type of learning works with unlabeled data. Dimensionality reduction and clustering are examples of such learning. Dimensionality reduction is used to reduce a large number of attributes to just a few that can produce the same results. There are several methods that are available for reducing the dimensionality of data, such as **principal component analysis** (**PCA**), t-SNE, wavelet transformation, and attribute subset selection.

The term cluster means a group of similar items that are closely related to each other. Clustering is an approach for generating units or groups of items that are similar to each other. This similarity is computed based on certain features or characteristics of items. We can say that a cluster is a set of data points that are similar to others in its cluster and dissimilar to data points of other clusters. Clustering has numerous applications, such as in searching documents, business intelligence, information security, and recommender systems:



In the preceding diagram, we can see how clustering puts data records or observations into a few groups, and dimensionality reduction reduces the number of features or attributes. Let's look at each of these topics in detail in the upcoming sections.

Reducing the dimensionality of data

Reducing dimensionality, or dimensionality reduction, entails scaling down a large number of attributes or columns (features) into a smaller number of attributes. The main objective of this technique is to get the best number of features for classification, regression, and other unsupervised approaches. In machine learning, we face a problem called the curse of dimensionality. This is where there is a large number of attributes or features. This means more data, causing complex models and overfitting problems.

Dimensionality reduction helps us to deal with the curse of dimensionality. It can transform data linearly and nonlinearly. Techniques for linear transformations include PCA, linear discriminant analysis, and factor analysis. Non-linear transformations include techniques such as t-SNE, Hessian eigenmaps, spectral embedding, and isometric feature mapping. Dimensionality reduction offers the following benefits:

- It filters redundant and less important features.
- It reduces model complexity with less dimensional data.
- It reduces memory and computation costs for model generation.
- It visualizes high-dimensional data.

In the next section, we will focus on one of the important and popular dimension reduction techniques, PCA.

PCA

In machine learning, it is considered that having a large amount of data means having a good-quality model for prediction, but a large dataset also poses the challenge of higher dimensionality (or the curse of dimensionality). It causes an increase in complexity for prediction models due to the large number of attributes. PCA is the most commonly used dimensionality reduction method and helps us to identify patterns and correlations in the original dataset to transform it into a lower-dimension dataset with no loss of information.

The main concept of PCA is the discovery of unseen relationships and correlations among attributes in the original dataset. Highly correlated attributes are so similar as to be redundant. Therefore, PCA removes such redundant attributes. For example, if we have 200 attributes or columns in our data, it becomes very difficult for us to proceed, what with such a huge number of attributes. In such cases, we need to reduce that number to 10 or 20 variables. Another objective of PCA is to reduce the dimensionality without affecting the significant information. For *p*-dimensional data, the PCA equation can be written as follows:

$$PC_j = w_{1j}X_1 + w_{2j}X_2 + \ldots\ldots + w_{pj}X_p$$

Principal components are a weighted sum of all the attributes. Here, $X_1, X_2, X_3 \ldots . X_p$ are the attributes in the original dataset and $w_{1j}, \ w_{2j}, \ w_{3j}, \ldots . w_{pj}$ are the weights of the attributes.

Let's take an example. Let's consider the streets in a given city as attributes, and let's say you want to visit this city. Now the question is, how many streets you will visit? Obviously, you will want to visit the popular or main streets of the city, which let's say is 10 out of the 50 available streets. These 10 streets will give you the best understanding of that city. These streets are then principal components, as they explain enough of the variance in the data (the city's streets).

Performing PCA

Let's perform PCA from scratch in Python:

1. Compute the correlation or covariance matrix of a given dataset.
2. Find the eigenvalues and eigenvectors of the correlation or covariance matrix.
3. Multiply the eigenvector matrix by the original dataset and you will get the principal component matrix.

Let's implement PCA from scratch:

1. We will begin by importing libraries and defining the dataset:

```
# Import numpy
import numpy as np
# Import linear algebra module
from scipy import linalg as la
# Create dataset
data=np.array([[7., 4., 3.],
[4., 1., 8.],
[6., 3., 5.],
[8., 6., 1.],
[8., 5., 7.],
[7., 2., 9.],
[5., 3., 3.],
[9., 5., 8.],
[7., 4., 5.],
[8., 2., 2.]])
```

2. Calculate the covariance matrix:

```
# Calculate the covariance matrix
# Center your data
data -= data.mean(axis=0)
cov = np.cov(data, rowvar=False)
```

3. Calculate the eigenvalues and eigenvector of the covariance matrix:

```
# Calculate eigenvalues and eigenvector of the covariance matrix
evals, evecs = la.eig(cov)
```

4. Multiply the original data matrix by the eigenvector matrix:

```
# Multiply the original data matrix with Eigenvector matrix.

# Sort the Eigen values and vector and select components
num_components=2
```

```
    sorted_key = np.argsort(evals)[::-1][:num_components]
    evals, evecs = evals[sorted_key], evecs[:, sorted_key]

    print("Eigenvalues:", evals)
    print("Eigenvector:", evecs)
    print("Sorted and Selected Eigen Values:", evals)
    print("Sorted and Selected Eigen Vector:", evecs)

    # Multiply original data and Eigen vector
    principal_components=np.dot(data,evecs)
    print("Principal Components:", principal_components)
```

This results in the following output:

```
Eigenvalues: [0.74992815+0.j 3.67612927+0.j 8.27394258+0.j]
Eigenvector: [[-0.70172743 0.69903712 -0.1375708 ]
[ 0.70745703 0.66088917 -0.25045969]
[ 0.08416157 0.27307986 0.95830278]]

Sorted and Selected Eigen Values: [8.27394258+0.j 3.67612927+0.j]

Sorted and Selected Eigen Vector: [[-0.1375708 0.69903712]
[-0.25045969 0.66088917]
[ 0.95830278 0.27307986]]

Principal Components: [[-2.15142276 -0.17311941]
[ 3.80418259 -2.88749898]
[ 0.15321328 -0.98688598]
[-4.7065185 1.30153634]
[ 1.29375788 2.27912632]
[ 4.0993133 0.1435814 ]
[-1.62582148 -2.23208282]
[ 2.11448986 3.2512433 ]
[-0.2348172 0.37304031]
[-2.74637697 -1.06894049]]
```

Here, we have computed a principal component matrix from scratch. First, we centered the data and computed the covariance matrix. After calculating the covariance matrix, we calculated the eigenvalues and eigenvectors. Finally, we chose two principal components (the number of components should be equal to the number of eigenvalues greater than 1) and multiplied the original data by the sorted and selected eigenvectors. We can also perform PCA using the scikit-learn library.

Let's perform PCA using scikit-learn in Python:

```
# Import pandas and PCA
import pandas as pd
```

```
# Import principal component analysis
from sklearn.decomposition import PCA

# Create dataset
data=np.array([[7., 4., 3.],
[4., 1., 8.],
[6., 3., 5.],
[8., 6., 1.],
[8., 5., 7.],
[7., 2., 9.],
[5., 3., 3.],
[9., 5., 8.],
[7., 4., 5.],
[8., 2., 2.]])

# Create and fit_transformed PCA Model
pca_model = PCA(n_components=2)
components = pca_model.fit_transform(data)
components_df = pd.DataFrame(data = components,

columns = ['principal_component_1', 'principal_component_2'])
print(components_df)
```

This results in the following output:

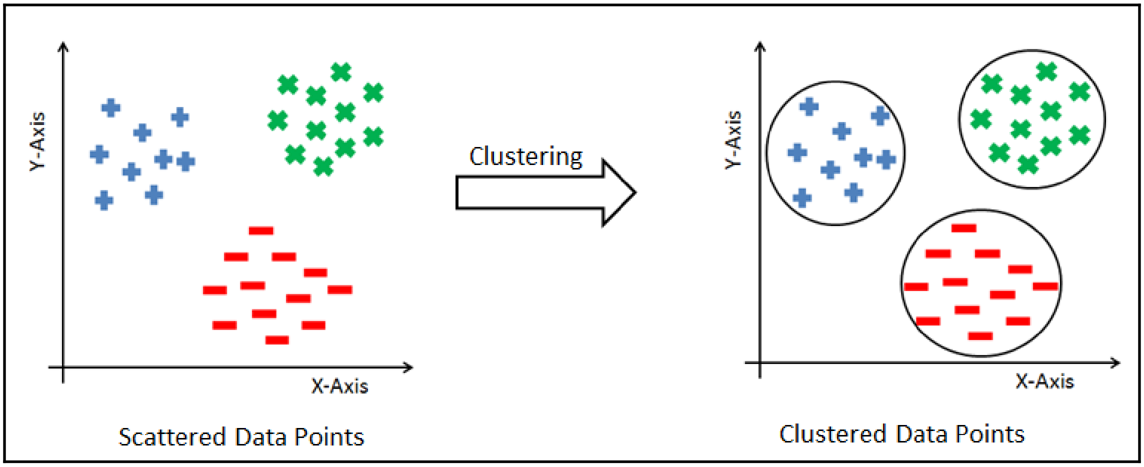
```
principal_component_1 principal_component_2

0 2.151423 -0.173119
1 -3.804183 -2.887499
2 -0.153213 -0.986886
3 4.706518 1.301536
4 -1.293758 2.279126
5 -4.099313 0.143581
6 1.625821 -2.232083
7 -2.114490 3.251243
8 0.234817 0.373040
9 2.746377 -1.068940
```

In the preceding code, we performed PCA using the scikit-learn library. First, we created the dataset and instantiated the PCA object. After this, we performed `fit_transform()` and generated the principal components.

That was all about PCA. Now it's time to learn about another unsupervised learning concept, clustering.

Clustering

Clustering means grouping items that are similar to each other. Grouping similar products, grouping similar articles or documents, and grouping similar customers for market segmentation are all examples of clustering. The core principle of clustering is minimizing the intra-cluster distance and maximizing the intercluster distance. The intra-cluster distance is the distance between data items within a group, and the inter-cluster distance is the distance between different groups. The data points are not labeled, so clustering is a kind of unsupervised problem. There are various methods for clustering and each method uses a different way to group the data points. The following diagram shows how data observations are grouped together using clustering:



As we are combining similar data points, the question that arises here is how to find the similarity between two data points so we can group similar data objects into the same cluster. In order to measure the similarity or dissimilarity between data points, we can use distance measures such as Euclidean, Manhattan, and Minkowski distance:

$$Euclidean\ dist. = \sqrt{\sum_{i=1}^{k} \quad (x_i - y_i)^2}$$

$$Manhattan\ dist. = \sum_{i=1}^{k} |x_i - y_i|$$

$$Minkowski\ dist. = \left(\sum_{i=1}^{k} (|x_i - y_i|)^q \right)^{\left(\frac{1}{q}\right)}$$

Here, the distance formula calculates the distance between two k-dimensional vectors, x_i and y_i.

Now we know what clustering is, but the most important question is, how many numbers of clusters should be considered when grouping the data? That's the biggest challenge for most clustering algorithms. There are lots of ways to decide the number of clusters. Let's discuss those methods in the next section.

Finding the number of clusters

In this section, we will focus on the most fundamental issue of clustering algorithms, which is discovering the number of clusters in a dataset – there is no definitive answer. However, not all clustering algorithms require a predefined number of clusters. In hierarchical and DBSCAN clustering, there is no need to define the number of clusters, but in k-means, k-medoids, and spectral clustering, we need to define the number of clusters. Selecting the right value for the number of clusters is tricky, so let's look at a couple of the methods for determining the best number of clusters:

- The elbow method
- The silhouette method

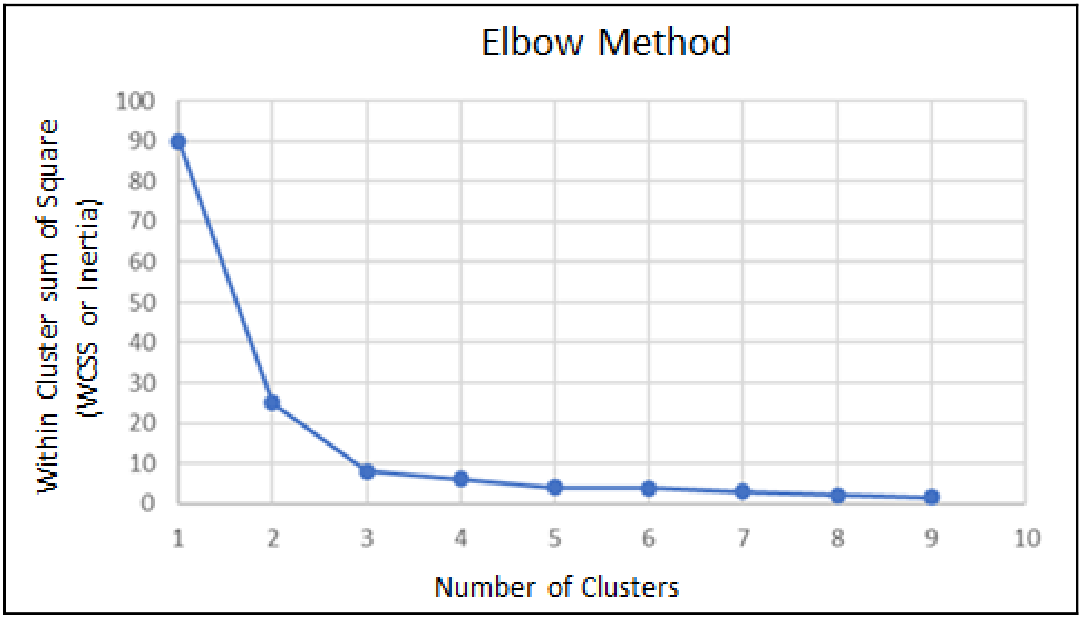
Let's look at these methods in detail.

The elbow method

The elbow method is a well-known method for finding out the best number of clusters. In this method, we focus on the percentage of variance for the different numbers of clusters. The core concept of this method is to select the number of clusters that appending another cluster should not cause a huge change in the variance. We can plot a graph for the sum of squares within a cluster using the number of clusters to find the optimal value. The sum of squares is also known as the **Within-Cluster Sum of Squares** (**WCSS**) or inertia:

$$WCSS = \sum_{j=1}^{k} \sum_{i}^{n} distance(x_i, C_j)^2$$

HereC_j is the cluster centroid and x_i is the data points in each cluster:

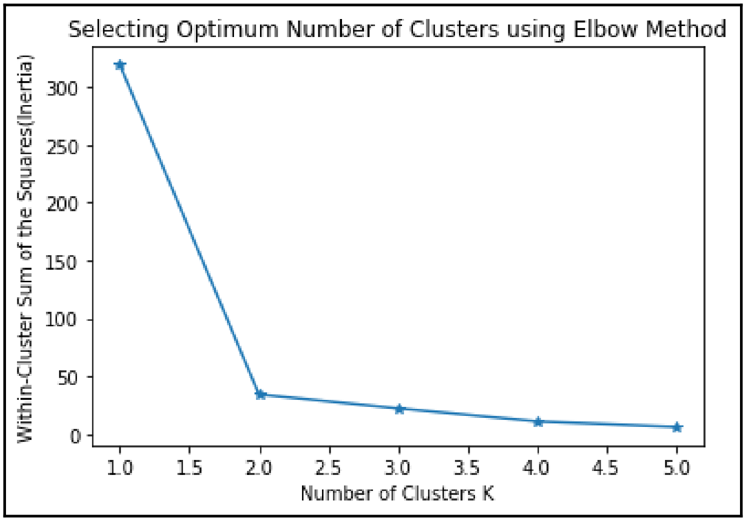


As you can see, at k = 3, the graph begins to flatten significantly, so we would choose 3 as the number of clusters.

Let's find the optimal number of clusters using the elbow method in Python:

```
# import pandas
import pandas as pd

# import matplotlib
import matplotlib.pyplot as plt

# import K-means
from sklearn.cluster import KMeans

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})
wcss_list = []

# Run a loop for different value of number of cluster
for i in range(1, 6):
    # Create and fit the KMeans model
    kmeans_model = KMeans(n_clusters = i, random_state = 123)
    kmeans_model.fit(data)
    # Add the WCSS or inertia of the clusters to the score_list
    wcss_list.append(kmeans_model.inertia_)

# Plot the inertia(WCSS) and number of clusters
plt.plot(range(1, 6), wcss_list, marker='*')

# set title of the plot
plt.title('Selecting Optimum Number of Clusters using Elbow Method')

# Set x-axis label
plt.xlabel('Number of Clusters K')

# Set y-axis label
plt.ylabel('Within-Cluster Sum of the Squares(Inertia)')

# Display plot
plt.show()
```

This results in the following output:



In the preceding example, we created a DataFrame with two columns, `X` and `Y`. We generated the clusters using `K-means` and computed the WCSS. After this, we plotted the number of clusters and inertia. As you can see at k = 2, the graph begins to flatten significantly, so we would choose 2 as the best number of clusters.

The silhouette method

The silhouette method assesses and validates cluster data. It finds how well each data point is classified. The plot of the silhouette score helps us to visualize and interpret how well data points are tightly grouped within their own clusters and separated from others. It helps us to evaluate the number of clusters. Its score ranges from -1 to +1. A positive value indicates a well-separated cluster and a negative value indicates incorrectly assigned data points. The more positive the value, the further data points are from the nearest clusters; a value of zero indicates data points that are at the separation line between two clusters. Let's see the formula for the silhouette score:

$$S(i) = \frac{b_i - a_i}{max(b_i, a_i)}$$

a_i is the average distance of the *i*th data point from other points within the cluster.

b_i is the average distance of the *i*th data point from other cluster points.

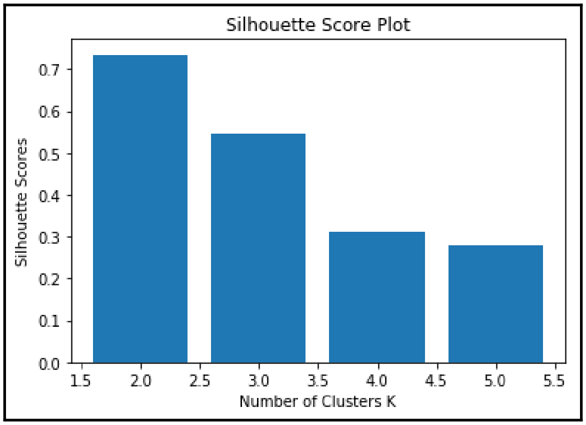
This means we can easily say that *S(i)* would be between [-1, 1]. So, for *S(i)* to be near to 1, a_i must be very small compared to $b_{i,\ \text{that is,}}e.\ a_i << b_i$.

Let's find the optimum number of clusters using the silhouette score in Python:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# import k-means for performing clustering
from sklearn.cluster import KMeans

# import silhouette score
from sklearn.metrics import silhouette_score

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})
score_list = []

# Run a loop for different value of number of cluster
for i in range(2, 6):
    # Create and fit the KMeans model
    kmeans_model = KMeans(n_clusters = i, random_state = 123)
    kmeans_model.fit(data)
    # Make predictions
    pred=kmeans_model.predict(data)
    # Calculate the Silhouette Score
    score = silhouette_score (data, pred, metric='euclidean')

    # Add the Silhouette score of the clusters to the score_list
    score_list.append(score)

# Plot the Silhouette score and number of cluster
plt.bar(range(2, 6), score_list)

# Set title of the plot
plt.title('Silhouette Score Plot')

# Set x-axis label
plt.xlabel('Number of Clusters K')
```

```
# Set y-axis label
plt.ylabel('Silhouette Scores')

# Display plot
plt.show()
```

This results in the following output:



In the preceding example, we created a DataFrame with two columns, `X` and `Y`. We generated clusters with different numbers of clusters on the created DataFrame using `K-means` and computed the silhouette score. After this, we plotted the number of clusters and the silhouette scores using a barplot. As you can see, at k = 2, the silhouette score has the highest value, so we would choose 2 clusters. Let's jump to the k-means clustering technique.

Partitioning data using k-means clustering

k-means is one of the simplest, most popular, and most well-known clustering algorithms. It is a kind of partitioning clustering method. It partitions input data by defining a random initial cluster center based on a given number of clusters. In the next iteration, it associates the data items to the nearest cluster center using Euclidean distance. In this algorithm, the initial cluster center can be chosen manually or randomly. k-means takes data and the number of clusters as input and performs the following steps:

1. Select *k* random data items as the initial centers of clusters.
2. Allocate the data items to the nearest cluster center.

3. Select the new cluster center by averaging the values of other cluster items.
4. Repeat steps 2 and 3 until there is no change in the clusters.

This algorithm aims to minimize the sum of squared errors:

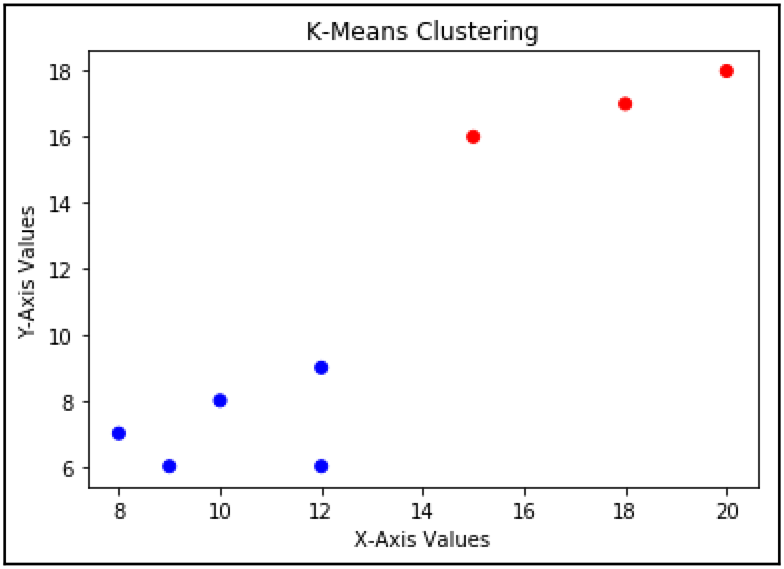
$$\sum_{i=1}^{k} \sum_{p \in C_i} dist(p, C_i)^2$$

k-means is one of the fastest and most robust algorithms of its kind. It works best with a dataset with distinct and separate data items. It generates spherical clusters. k-means requires the number of clusters as input at the beginning. If data items are very much overlapped, it doesn't work well. It captures the local optima of the squared error function. It doesn't perform well with noisy and non-linear data. It also doesn't work well with non-spherical clusters. Let's create a clustering model using k-means clustering:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# Import K-means
from sklearn.cluster import KMeans

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})

# Define number of clusters
num_clusters = 2

# Create and fit the KMeans model
km = KMeans(n_clusters=num_clusters)
km.fit(data)

# Predict the target variable
pred=km.predict(data)

# Plot the Clusters
plt.scatter(data.X,data.Y,c=pred, marker="o", cmap="bwr_r")

# Set title of the plot
plt.title('K-Means Clustering')
```

```
# Set x-axis label
plt.xlabel('X-Axis Values')

# Set y-axis label
plt.ylabel('Y-Axis Values')

# Display the plot
plt.show()
```

This results in the following output:

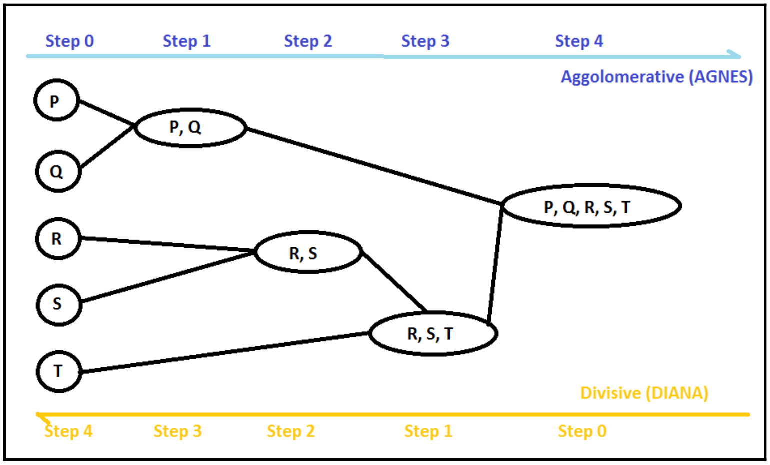


In the preceding code example, we imported the `KMeans` class and created its object or model. This model will fit it on the dataset (without labels). After training, the model is ready to make predictions using the `predict()` method. After predicting the results, we plotted the cluster results using a scatter plot. In this section, we have seen how k-means works and its implementation using the scikit-learn library. In the next section, we will look at hierarchical clustering.

Hierarchical clustering

Hierarchical clustering groups data items based on different levels of a hierarchy. It combines the items in groups based on different levels of a hierarchy using top-down or bottom-up strategies. Based on the strategy used, hierarchical clustering can be of two types – agglomerative or divisive:

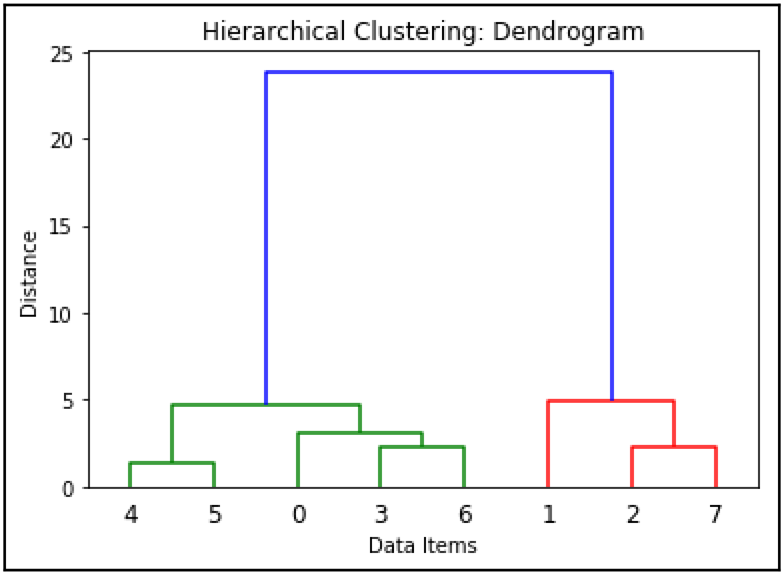
- The agglomerative type is the most widely used hierarchical clustering technique. It groups similar data items in the form of a hierarchy based on similarity. This method is also called **Agglomerative Nesting** (**AGNES**). This algorithm starts by considering every data item as an individual cluster and combines clusters based on similarity. It iteratively collects small clusters and combines them into a single large cluster. This algorithm gives its result in the form of a tree structure. It works in a bottom-up manner; that is, every item is initially considered as a single element cluster and in each iteration of the algorithm, the two most similar clusters are combined and form a bigger cluster.
- Divisive hierarchical clustering is a top-down strategy algorithm. It is also known as **Divisive Analysis** (**DIANA**). It starts with all the data items as a single big cluster and partitions recursively. In each iteration, clusters are divided into two non-similar or heterogeneous sub-clusters:



In order to decide which clusters should be grouped or split, we use various distances and linkage criteria such as single, complete, average, and centroid linkage. These criteria decide the shape of the cluster. Both types of hierarchical clustering (agglomerative and divisive hierarchical clustering) require a predefined number of clusters or a distance threshold as input to terminate the recursive process. It is difficult to decide the distance threshold, so the easiest option is to check the number of clusters using a dendrogram. Dendrograms help us to understand the process of hierarchical clustering. Let's see how to create a dendrogram using the `scipy` library:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# Import dendrogram
from scipy.cluster.hierarchy import dendrogram
from scipy.cluster.hierarchy import linkage

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})

# create dendrogram using ward linkage
dendrogram_plot = dendrogram(linkage(data, method = 'ward'))

# Set title of the plot
plt.title('Hierarchical Clustering: Dendrogram')

# Set x-axis label
plt.xlabel('Data Items')

# Set y-axis label
plt.ylabel('Distance')

# Display the plot
plt.show()
```

This results in the following output:

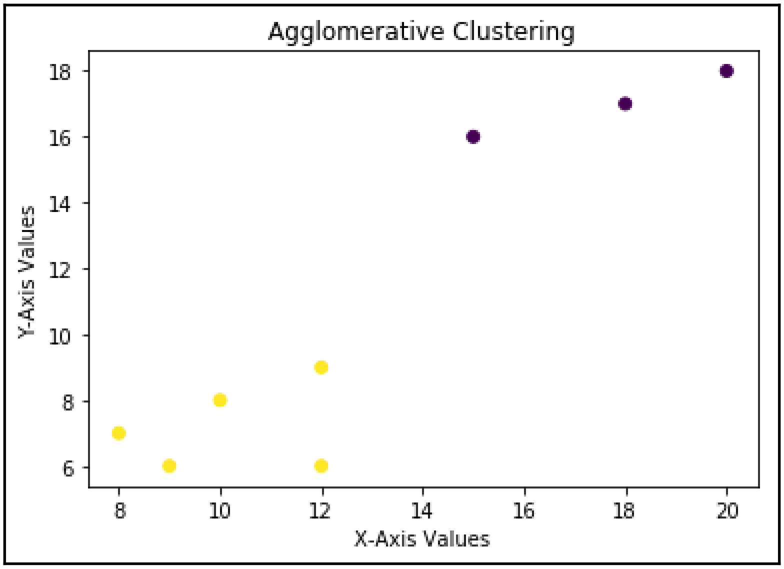


In the preceding code example, we created the dataset and generated the dendrogram using ward linkage. For the dendrograms, we used the `scipy.cluster.hierarchy` module. To set the plot title and axis labels, we used `matplotlib`. In order to select the number of clusters, we need to draw a horizontal line without intersecting the clusters and count the number of vertical lines to find the number of clusters. Let's create a clustering model using agglomerative clustering:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# Import Agglomerative Clustering
from sklearn.cluster import AgglomerativeClustering

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})

# Specify number of clusters
num_clusters = 2

# Create agglomerative clustering model
```

```
ac = AgglomerativeClustering(n_clusters = num_clusters, linkage='ward')

# Fit the Agglomerative Clustering model
ac.fit(data)

# Predict the target variable
pred=ac.labels_

# Plot the Clusters
plt.scatter(data.X,data.Y,c=pred, marker="o")

# Set title of the plot
plt.title('Agglomerative Clustering')

# Set x-axis label
plt.xlabel('X-Axis Values')

# Set y-axis label
plt.ylabel('Y-Axis Values')

# Display the plot
plt.show()
```

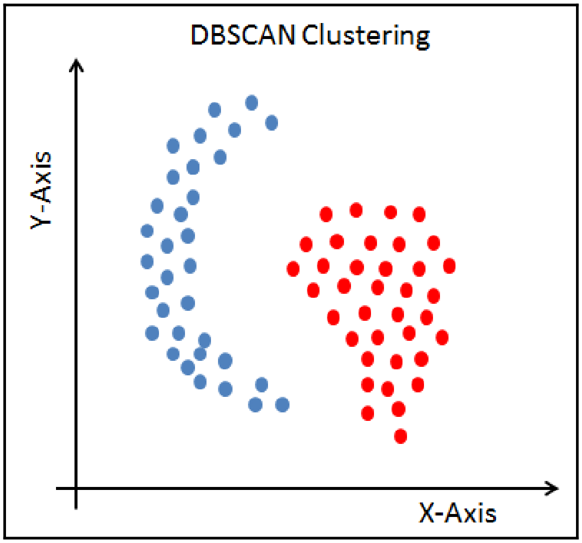
This results in the following output:



In the preceding code example, we imported the `AgglomerativeClustering` class and created its object or model. This model will fit on the dataset without labels. After training, the model is ready to make predictions using the `predict()` method. After predicting the results, we plotted the cluster results using a scatter plot. In this section, we have seen how hierarchical clustering works and its implementation using the `scipy` and scikit-learn libraries. In the next section, we will look at density-based clustering.

DBSCAN clustering

Partitioning clustering methods, such as k-means, and hierarchical clustering methods, such as agglomerative clustering, are good for discovering spherical or convex clusters. These algorithms are more sensitive to noise or outliers and work for well-separated clusters:



Intuitively, we can say that a density-based clustering approach is most similar t how we as humans might instinctively group items. In all the preceding figures, we can quickly see the number of different groups or clusters due to the density of the items.

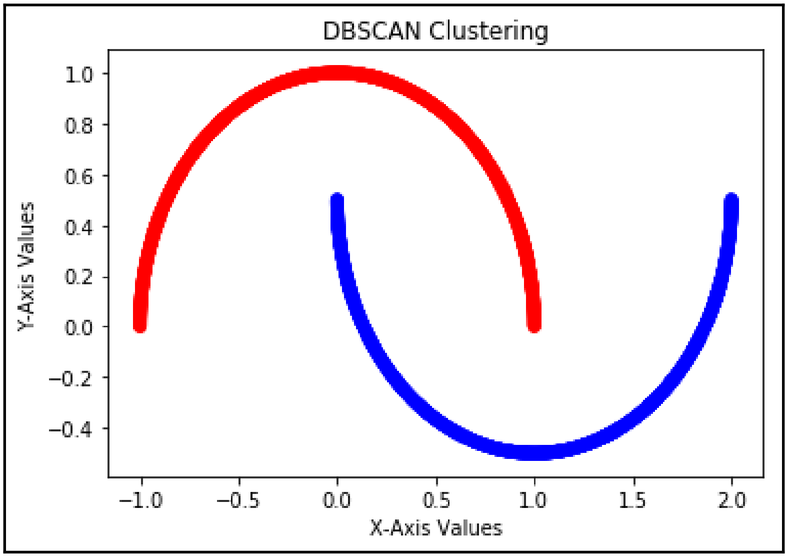
Density-Based Spatial Clustering of Applications with Noise (**DBSCAN**) is based on the idea of groups and noise. The main idea behind it is that each data item of a group or cluster has a minimum number of data items in a given radius.

The main goal of DBSCAN is to discover the dense region that can be computed using minimum number of objects (`minPoints`) and given radius (`eps`). DBSCAN has the capability to generate random shapes of clusters and deal with noise in a dataset. Also, there is no requirement to feed in the number of clusters. DBSCAN automatically identifies the number of clusters in the data.

Let's create a clustering model using DBSCAN clustering in Python:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# Import DBSCAN clustering model
from sklearn.cluster import DBSCAN

# import make_moons dataset
from sklearn.datasets import make_moons

# Generate some random moon data
features, label = make_moons(n_samples = 2000)

# Create DBSCAN clustering model
db = DBSCAN()

# Fit the Spectral Clustering model
db.fit(features)

# Predict the target variable
pred_label=db.labels_

# Plot the Clusters
plt.scatter(features[:, 0], features[:, 1], c=pred_label,
marker="o",cmap="bwr_r")

# Set title of the plot
plt.title('DBSCAN Clustering')

# Set x-axis label
plt.xlabel('X-Axis Values')

# Set y-axis label
plt.ylabel('Y-Axis Values')

# Display the plot
plt.show()
```

This results in the following output:



First, we import the `DBSCAN` class and create the moon dataset. After this, we create the DBSCAN model and fit it on the dataset. DBSCAN does not need the number of clusters. After training, the model is ready to make predictions using the `predict()` method. After predicting the results, we plotted the cluster results using a scatter plot. In this section, we have seen how DBSCAN clustering works and its implementation using the scikit-learn library. In the next section, we will see the spectral clustering technique.

Spectral clustering

Spectral clustering is a method that employs the spectrum of a similarity matrix. The spectrum of a matrix represents the set of its eigenvalues, and a similarity matrix consists of similarity scores between each data point. It reduces the dimensionality of data before clustering. In other words, we can say that spectral clustering creates a graph of data points, and these points are mapped to a lower dimension and separated into clusters.

A similarity matrix converts data to conquer the lack of convexity in the distribution. For any dataset, the data points could be *n*-dimensional, and here could be *m* data points. From these *m* points, we can create a graph where the points are nodes and the edges are weighted with the similarity between points. A common way to define similarity is with a Gaussian kernel, which is a nonlinear function of Euclidean distance:

$$K(x_i, x_j) = e^{\left(-\frac{\|x_i - x_j\|^2}{2\,\sigma^2}\right)}$$

The distance of this function ranges from 0 to 1. The fact that it's bounded between zero and one is a nice property. The absolute distance (it can be anything) in Euclidean distance can cause instability and difficulty in modeling. You can think of the Gaussian kernel as a normalization function for Euclidean distance.

After getting the graph, create an adjacency matrix and put in each cell of the matrix the weight of the edge ij^{th}. This is a symmetric matrix. Let's call the adjacency matrix A. We can also create a "degree" diagonal matrix D, which will have in each X_i element the sum of the weights of all edges linked to node *i*. Let's call this matrix D. For a given graph G with *n* vertices, its *n***n* Laplacian matrix can be defined as follows:

$$L = D - A$$

Here *D* is the degree matrix and *A* is the adjacency matrix of the graph.

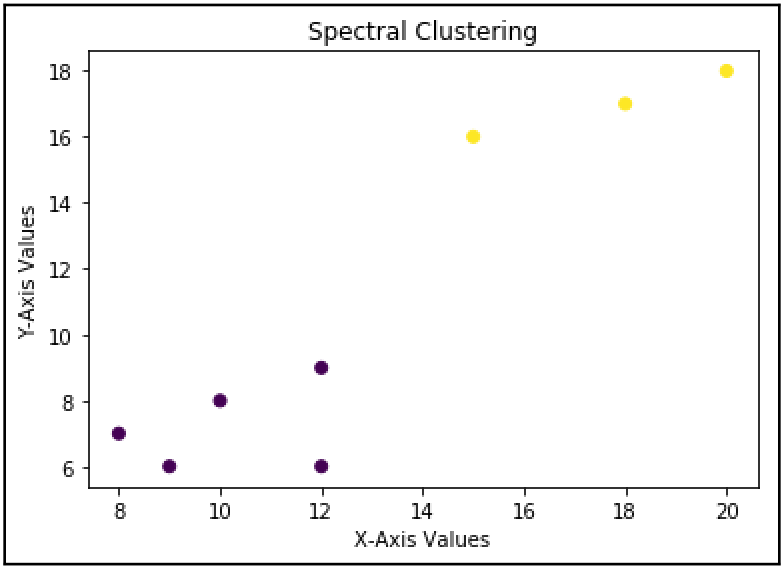
Now we have the Laplacian matrix of the graph (G). We can compute the spectrum of a matrix of eigenvectors. If we take *k* least-significant eigenvectors, we get a representation in *k* dimensions. The least-significant eigenvectors are the ones associated with the smallest eigenvalues. Each eigenvector provides information about the connectivity of the graph.

The idea of spectral clustering is to cluster the points using these *k* eigenvectors as features. So, you take the *k* least-significant eigenvectors and you have your *m* points in *k* dimensions. You run a clustering algorithm, such as k-means, and then you have your result. This *k* in spectral clustering is deeply related to the Gaussian kernel k-means. You can also think about it as a clustering method where your points are projected into a space of infinite dimensions, clustered there, and then you use those results as the results of clustering your points.

Spectral clustering is used when k-means works badly because the clusters are not linearly distinguishable in their original space. We can also try other clustering methods, such as hierarchical clustering or density-based clustering, to solve this problem. Let's create a clustering model using spectral clustering in Python:

```
# import pandas
import pandas as pd

# import matplotlib for data visualization
import matplotlib.pyplot as plt

# Import Spectral Clustering
from sklearn.cluster import SpectralClustering

# Create a DataFrame
data=pd.DataFrame({"X":[12,15,18,10,8,9,12,20],
"Y":[6,16,17,8,7,6,9,18]})

# Specify number of clusters
num_clusters = 2

# Create Spectral Clustering model
sc=SpectralClustering(num_clusters, affinity='rbf', n_init=100,
assign_labels='discretize')

# Fit the Spectral Clustering model
sc.fit(data)

# Predict the target variable
pred=sc.labels_

# Plot the Clusters
plt.scatter(data.X,data.Y,c=pred, marker="o")

# Set title of the plot
plt.title('Spectral Clustering')

# Set x-axis label
plt.xlabel('X-Axis Values')

# Set y-axis label
plt.ylabel('Y-Axis Values')

# Display the plot
plt.show()
```

This results in the following output:



In the preceding code example, we imported the `SpectralClustering` class and created a dummy dataset using pandas. After this, we created the model and fit it on the dataset. After training, the model is ready to make predictions using the `predict()` method. In this section, we have seen how spectral clustering works and its implementation using the scikit-learn library. In the next section, we will see how to evaluate a clustering algorithm's performance.

Evaluating clustering performance

Evaluating clustering performance is an essential step to assess the strength of a clustering algorithm for a given dataset. Assessing performance in an unsupervised environment is not an easy task, but in the literature, many methods are available. We can categorize these methods into two broad categories: internal and external performance evaluation. Let's learn about both of these categories in detail.

Internal performance evaluation

In internal performance evaluation, clustering is evaluated based on feature data only. This method does not use any target label information. These evaluation measures assign better scores to clustering methods that generate well-separated clusters. Here, a high score does not guarantee effective clustering results.

Internal performance evaluation helps us to compare multiple clustering algorithms but it does not mean that a better-scoring algorithm will generate better results than other algorithms. The following internal performance evaluation measures can be utilized to estimate the quality of generated clusters:

The Davies-Bouldin index

The **Davies-Bouldin index** (**BDI**) is the ratio of intra-cluster distance to inter-cluster distance. A lower DBI value means better clustering results. This can be calculated as follows:

$$DBI = \frac{1}{n}\sum_{i=1}^{n}\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}$$

Here, the following applies:

- n: The number of clusters
- c_i: The centroid of cluster *i*
- σ_i: The intra-cluster distance or average distance of all cluster items from the centroid c_i
- $d(c_i, c_j)$: The inter-cluster distance between two cluster centroids c_i and c_j

The silhouette coefficient

The silhouette coefficient finds the similarity of an item in a cluster to its own cluster items and other nearest clusters. It is also used to find the number of clusters, as we have seen elsewhere in this chapter. A high silhouette coefficient means better clustering results. This can be calculated as follows:

$$S(i) = \frac{b_i - a_i}{max(b_i, a_i)}$$

a_i is the average distance of the i^{th} data point to other points within the cluster.

b_i is the average distance of the i^{th} data point to other cluster points.

So, we can say that *S(i)* would be between [-1, 1]. So, for *S(i)* to be near to 1, a_i must be very small compared to b_i, that is, $a_i << b_i$.

External performance evaluation

In external performance evaluation, generated clustering is evaluated using the actual labels of clusters that are not used in the clustering process. It is similar to a supervised learning evaluation process; that is, we can use the same confusion matrix here to assess performance. The following external evaluation measures are used to evaluate the quality of generated clusters.

The Rand score

The Rand score shows how similar a cluster is to the benchmark classification and computes the percentage of correctly made decisions. A lower value is preferable because this represents distinct clusters. This can be calculated as follows:

$$RandScore = \frac{TP + TN}{TP + FP + TN + FN}$$

Here, the following applies:

- TP: Total number of true positives
- TN: Total number of true negatives
- FP: Total number of false positives
- FN: Total number of false negatives

The Jaccard score

The Jaccard score computes the similarity between two datasets. It ranges from 0 to 1. 1 means the datasets are identical and 0 means the datasets have no common elements. A low value is preferable because it indicates distinct clusters. This can be calculated as follows:

$$J(A, B) = \frac{A \cap B}{A \cup B} = \frac{TP}{TP + FP + FN}$$

Here A and B are two datasets.

F-Measure or F1-score

The F-measure is a harmonic mean of precision and recall values. It measures both the precision and robustness of clustering algorithms. It also tries to equalize the participation of false negatives using the value of β. This can be calculated as follows:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F_\beta = \frac{(\beta^2 + 1). Precision. Recall}{\beta^2. Precision + Recall}$$

Here β is the non-negative value. β=1 gives equal weight to precision and recall, β = 0.5 gives twice the weight to precision than to recall, and β = 0 gives no importance to recall.

The Fowlkes-Mallows score

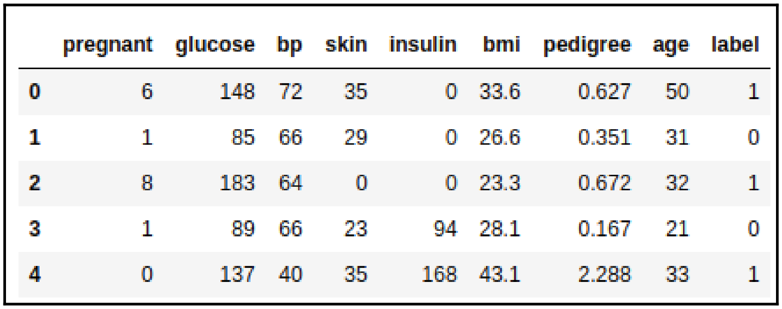
The Fowlkes-Mallows score is a geometric mean of precision and recall. A high value represents similar clusters. This can be calculated as follows:

$$Fowlkes - Mallows\ Score = \sqrt{Precision. Recall}$$

Let's create a cluster model using k-means clustering and evaluate the performance using the internal and external evaluation measures in Python using the Pima Indian Diabetes dataset (`https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/blob/master/Chapter11/diabetes.csv`):

```
# Import libraries
import pandas as pd

# read the dataset
diabetes = pd.read_csv("diabetes.csv")

# Show top 5-records
diabetes.head()
```

This results in the following output:



	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

First, we need to import pandas and read the dataset. In the preceding example, we are reading the Pima Indian Diabetes dataset:

```
# split dataset in two parts: feature set and target label
feature_set = ['pregnant', 'insulin', 'bmi',
'age','glucose','bp','pedigree']

features = diabetes[feature_set]
target = diabetes.label
```

After loading the dataset, we need to divide the dataset into dependent/label columns (target) and independent/feature columns (`feature_set`). After this, the dataset will be broken into train and test sets. Now both dependent and independent columns are broken into train and test sets (`feature_train`, `feature_test`, `target_train`, and `target_test`) using `train_test_split()`. Let's split the dataset into train and test parts:

```
# partition data into training and testing set
from sklearn.model_selection import train_test_split

feature_train, feature_test, target_train, target_test =
train_test_split(features, target, test_size=0.3, random_state=1)
```

Here, `train_test_split()` takes the dependent and independent DataFrames, `test_size` and `random_state`. Here, `test_size` will decide the ratio for the train-test split (having a `test_size` value of `0.3` means 30% of the data will go to the testing set and the remaining 70% will be for the training set), and `random_state` is used as a seed value for reproducing the same data split each time. If `random_state` is `None`, then it will split the records in a random fashion each time, which will give different performance measures:

```
# Import K-means Clustering
from sklearn.cluster import KMeans
```

```
# Import metrics module for performance evaluation
from sklearn.metrics import davies_bouldin_score
from sklearn.metrics import silhouette_score
from sklearn.metrics import adjusted_rand_score
from sklearn.metrics import jaccard_score
from sklearn.metrics import f1_score
from sklearn.metrics import fowlkes_mallows_score

# Specify the number of clusters
num_clusters = 2

# Create and fit the KMeans model
km = KMeans(n_clusters=num_clusters)
km.fit(feature_train)

# Predict the target variable
predictions = km.predict(feature_test)

# Calculate internal performance evaluation measures
print("Davies-Bouldin Index:", davies_bouldin_score(feature_test,
predictions))
print("Silhouette Coefficient:", silhouette_score(feature_test,
predictions))

# Calculate External performance evaluation measures
print("Adjusted Rand Score:", adjusted_rand_score(target_test,
predictions))
print("Jaccard Score:", jaccard_score(target_test, predictions))
print("F-Measure(F1-Score):", f1_score(target_test, predictions))
print("Fowlkes Mallows Score:", fowlkes_mallows_score(target_test,
predictions))
```

This results in the following output:

```
Davies-Bouldin Index: 0.7916877512521091
Silhouette Coefficient: 0.5365443098840619
Adjusted Rand Score: 0.03789319261940484
Jaccard Score: 0.22321428571428573
F-Measure(F1-Score): 0.36496350364963503
Fowlkes Mallows Score: 0.6041244457314743
```

First, we imported the `KMeans` and `metrics` modules. We created a k-means object or model and fit it on the training dataset (without labels). After training, the model makes predictions and these predictions are assessed using internal measures, such as the DBI and the silhouette coefficient, and external evaluation measures, such as the Rand score, the Jaccard score, the F-Measure, and the Fowlkes-Mallows score.

Summary

In this chapter, we have discovered unsupervised learning and its techniques, such as dimensionality reduction and clustering. The main focus was on PCA for dimensionality reduction and several clustering methods, such as k-means clustering, hierarchical clustering, DBSCAN, and spectral clustering. The chapter started with dimensionality reduction and PCA. After PCA, our main focus was on clustering techniques and how to identify the number of clusters. In later sections, we moved on to cluster performance evaluation measures such as the DBI and the silhouette coefficient, which are internal measures. After looking at internal clustering measures, we looked at external measures such as the Rand score, the Jaccard score, the F-measure, and the Fowlkes-Mallows index.

The next chapter, `Chapter 12`, *Analyzing Textual Data*, will focus on text analytics, covering the text preprocessing and text classification using NLTK, SpaCy, and scikit-learn. The chapter starts by exploring basic operations on textual data such as text normalization using tokenization, stopwords removal, stemming and lemmatization, parts-of-speech tagging, entity recognition, dependency parsing, and word clouds. In later sections, the focus will be on feature engineering approaches such as Bag of Words, term presence, TF-IDF, sentiment analysis, text classification, and text similarity.

4

Section 4: NLP, Image Analytics, and Parallel Computing

The main objective of this section is to get an overview of NLP, image analytics, and parallel computing. NLP skills comprise text preprocessing, sentiment analysis, and text similarity using NLTK and SpaCy. Image analytics comprises image processing and face detection using OpenCV. This section also focuses on the parallel computation of DataFrames, arrays, and machine learning algorithms.

This section includes the following chapters:

- `Chapter 12`, *Analyzing Textual Data*
- `Chapter 13`, *Analyzing Image Data*
- `Chapter 14`, *Parallel Computing Using Dask*

12
Analyzing Textual Data

In the age of information, data is produced at incredible speeds and volumes. The data produced is not only structured or tabular types, it can also be in a variety of unstructured types such as textual data, image or graphic data, speech data, and video. Text is a very common and rich type of data. Articles, blogs, tutorials, social media posts, and website content all produce unstructured textual data. Thousands of emails, messages, comments, and tweets are sent by people every minute. Such a large amount of text data needs to be mined. Text analytics offers lots of opportunities for business people; for instance, Amazon can interpret customer feedback on a particular product, news analysts can analyze news trends and the latest issues on Twitter, and Netflix can also interpret reviews of each movie and web series. Business analysts can interpret customer activities, reviews, feedback, and sentiments to drive their business effectively using NLP and text analysis.

In this chapter, we will start with basic text analytics operations such as tokenization, removing stopwords, stemming, lemmatization, PoS tagging, and entity recognition. After this, we will see how to visualize your text analysis using WordCloud. We will see how to find out the opinions of customers about a product based on reviews, using sentiment analysis. Here, we will perform sentiment analysis using text classification and assess model performance using accuracy, precision, recall, and f1-score. Finally, we will focus on text similarity between two sentences using Jaccard and cosine similarity.

The topics of this chapter are listed as follows:

- Installing NLTK and SpaCy
- Text normalization
- Tokenization
- Removing stopwords
- Stemming and lemmatization
- POS tagging

- Recognizing entities
- Dependency parsing
- Creating a word cloud
- Bag of words
- TF-IDF
- Sentiment analysis using text classification
- Text similarity

Technical requirements

This chapter has the following technical requirements:

- You can find the code and the datasets at the following GitHub link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter12`.
- All the code blocks are available in the `ch12.ipynb` file.
- This chapter uses only one TSV file (`amazon_alexa.tsv`) for practice purposes.
- In this chapter, we will use the NLTK, SpaCy, WordCloud, matplotlib, seaborn, and scikit-learn Python libraries.

Installing NLTK and SpaCy

NLTK is one of the popular and essential Python packages for natural language processing. It offers all the basic, as well as advanced, NLP operations. It comprises common algorithms such as tokenization, stemming, lemmatization, part-of-speech, and named entity recognition. The main features of the NLTK library are that it's open-source, easy to learn, easy to use, has a prominent community, and has well-organized documentation. The NLTK library can be installed using the `pip install` command running on the command line as follows:

```
pip install nltk
```

NLTK is not a pre-installed library in Anaconda. We can directly install `nltk` in the Jupyter Notebook. We can use an exclamation point (!) before the command in the cell:

```
!pip install nltk
```

SpaCy is another essential and powerful Python package for NLP. It offers a common NLP algorithm as well as advanced functionalities. It is designed for production purposes and develops applications for a large volume of data. The SpaCy library can be installed using the `pip install` command running on the command line as follows:

```
pip install spacy
```

After installing spaCy, we need to install a `spacy` English-language model. We can install it using the following command:

```
python -m spacy download en
```

Spacy and its English model are not pre-installed in Anaconda. We can directly install `spacy` using the following code. We can use the exclamation point (!) before the command in the cell:

```
!pip install spacy
!python -m spacy download en
```

Using the preceding syntax, we can install `spacy` and its English model in Jupyter Notebooks.

Text normalization

Text normalization converts text into standard or canonical form. It ensures consistency and helps in processing and analysis. There is no single approach to the normalization process. The first step in normalization is the lower case all the text. It is the simplest, most applicable, and effective method for text pre-processing. Another approach could be handling wrongly spelled words, acronyms, short forms, and the use of out-of-vocabulary words; for example, "super," "superb," and "superrrr" can be converted into "super". Text normalization handles the noise and disturbance in test data and prepares noise-free data. We also apply stemming and lemmatization to normalize the words present in the text.

Let's perform a basic normalization operation by converting the text into lowercase:

```
# Input text
paragraph="""Taj Mahal is one of the beautiful monuments. It is one of the
wonders of the world. It was built by Shah Jahan in 1631 in memory of his
third beloved wife Mumtaj Mahal."""

# Converting paragraph in lowercase
print(paragraph.lower())
```

This results in the following output:

```
taj mahal is one of the beautiful monuments. it is one of the wonders of
the world. it was built by shah jahan in 1631 in memory of his third
beloved wife mumtaj mahal.
```

In the preceding code block, we have converted the given input paragraph into lowercase by using the `lower()` method.

In NLP, text normalization deals with the randomness and converts text into a standard form that improves the overall performance of NLP solutions. It also reduces the size of the document term matrix by converting the words into their root word. In the upcoming sections, we will focus on basic text preprocessing operations.

Tokenization

Tokenization is the initial step in text analysis. Tokenization is defined as breaking down text paragraphs into smaller parts or tokens such as sentences or words and ignoring punctuation marks. Tokenization can be of two types: sentence tokenization and word tokenization. A sentence tokenizer splits a paragraph into sentences and word tokenization splits a text into words or tokens.

Let's tokenize a paragraph using NLTK and spaCy:

1. Before tokenization, import NLTK and download the required files:

```
# Loading NLTK module
import nltk

# downloading punkt
nltk.download('punkt')

# downloading stopwords
nltk.download('stopwords')

# downloading wordnet
nltk.download('wordnet')

# downloading average_perception_tagger
nltk.download('averaged_perceptron_tagger')
```

2. Now, we will tokenize paragraphs into sentences using the `sent_tokenize()` method of NLTK:

   ```
   # Sentence Tokenization
   from nltk.tokenize import sent_tokenize

   paragraph="""Taj Mahal is one of the beautiful monuments. It is one
   of the wonders of the world. It was built by Shah Jahan in 1631 in
   memory of his third beloved wife Mumtaj Mahal."""

   tokenized_sentences=sent_tokenize(paragraph)
   print(tokenized_sentences)
   ```

   This results in the following output:

   ```
   ['Taj Mahal is one of the beautiful monument.', 'It is one of the
   wonders of the world.', 'It was built by Shah Jahan in 1631 in
   memory of his third beloved wife Mumtaj Mahal.']
   ```

In the preceding example, we have taken a paragraph and passed it as a parameter to the `sent_tokenize()` method. The output of this method will be a list of sentences.

Let's tokenize the paragraph into sentences using spaCy:

```
# Import spacy
import spacy

# Loading english language model
nlp = spacy.load("en")

# Build the nlp pipe using 'sentencizer'
sent_pipe = nlp.create_pipe('sentencizer')

# Append the sentencizer pipe to the nlp pipeline
nlp.add_pipe(sent_pipe)
paragraph = """Taj Mahal is one of the beautiful monuments. It is one of
the wonders of the world. It was built by Shah Jahan in 1631 in memory of
his third beloved wife Mumtaj Mahal."""

# Create nlp Object to handle linguistic annotations in a documents.
nlp_doc = nlp(paragraph)

# Generate list of tokenized sentence
tokenized_sentences = []
for sentence in nlp_doc.sents:
    tokenized_sentences.append(sentence.text)
print(tokenized_sentences)
```

This results in the following output:

```
['Taj Mahal is one of the beautiful monument.', 'It is one of the wonders
of the world.', 'It was built by Shah Jahan in 1631 in memory of his third
beloved wife Mumtaj Mahal.']
```

In the preceding example, first, we have imported the English language model and instantiated it. After this, we created the NLP pipe using `sentencizer` and added it to the pipeline. Finally, we created the NLP object and iterated through the `sents` attribute of the NLP object to create a list of tokenized sentences.

Let's tokenize paragraphs into words using the `word_tokenize()` function of NLTK:

```
# Import nltk word_tokenize method
from nltk.tokenize import word_tokenize

# Split paragraph into words
tokenized_words=word_tokenize(paragraph)
print(tokenized_words)
```

This results in the following output:

```
['Taj', 'Mahal', 'is', 'one', 'of', 'the', 'beautiful', 'monument', '.',
'It', 'is', 'one', 'of', 'the', 'wonders', 'of', 'the', 'world', '.', 'It',
'was', 'built', 'by', 'Shah', 'Jahan', 'in', '1631', 'in', 'memory', 'of',
'his', 'third', 'beloved', 'wife', 'Mumtaj', 'Mahal', '.']
```

In the preceding example, we have taken a paragraph and passed it as a parameter to the `word_tokenize()` method. The output of this method will be a list of words.

Let's tokenize the paragraph into words using spaCy:

```
# Import spacy
import spacy

# Loading english language model
nlp = spacy.load("en")

paragraph = """Taj Mahal is one of the beautiful monuments. It is one of
the wonders of the world. It was built by Shah Jahan in 1631 in memory of
his third beloved wife Mumtaj Mahal."""

# Create nlp Object to handle linguistic annotations in a documents.
my_doc = nlp(paragraph)

# tokenize paragraph into words
tokenized_words = []
for token in my_doc:
    tokenized_words.append(token.text)

print(tokenized_words)
```

This results in the following output:

```
['Taj', 'Mahal', 'is', 'one', 'of', 'the', 'beautiful', 'monument', '.',
'It', 'is', 'one', 'of', 'the', 'wonders', 'of', 'the', 'world', '.', 'It',
'was', 'built', 'by', 'Shah', 'Jahan', 'in', '1631', 'in', 'memory', 'of',
'his', 'third', 'beloved', 'wife', 'Mumtaj', 'Mahal', '.']
```

In the preceding example, first, we imported the English language model and instantiated it. After this, we created a text paragraph. Finally, we created the NLP object using text paragraphs and iterated it to create a list of tokenized words.

Let's create the frequency distribution of tokenized words:

```
# Import frequency distribution
from nltk.probability import FreqDist

# Find frequency distribution of paragraph
fdist = FreqDist(tokenized_words)

# Check top 5 common words
fdist.most_common(5)
```

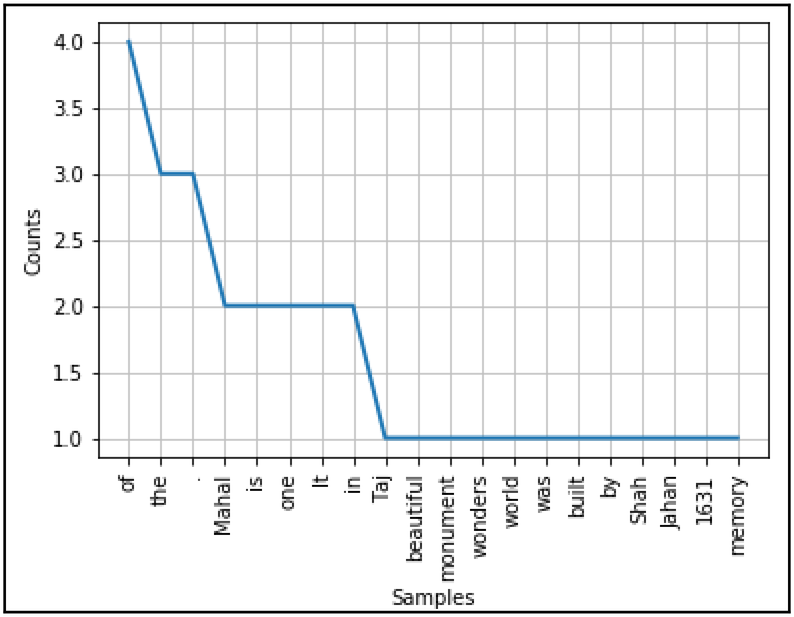
This results in the following output:

```
[('of', 4), ('the', 3), ('.', 3), ('Mahal', 2), ('is', 2)]
```

Let's create a frequency distribution plot using matplotlib:

```
# Import matplotlib
import matplotlib.pyplot as plt

# Plot Frequency Distribution
fdist.plot(20, cumulative=False)
plt.show()
```

This results in the following output:



In the preceding example, we have generated the frequency distribution of tokens using the `FreqDist` class. After sentence and word tokenization, we will learn how to remove stopwords from the given text.

Removing stopwords

Stopwords are counted as noise in text analysis. Any text paragraph has to have verbs, articles, and propositions. These are all considered stop words. Stop words are necessary for human conversation but they don't make many contributions in text analysis. Removing stopwords from text is called noise elimination.

Let's see how to remove stopwords using NLTK:

```
# import the nltk stopwords
from nltk.corpus import stopwords

# Load english stopwords list
stopwords_set=set(stopwords.words("english"))

# Removing stopwords from text
```

```
filtered_word_list=[]
for word in tokenized_words:
    # filter stopwords
    if word not in stopwords_set:
        filtered_word_list.append(word)

# print tokenized words
print("Tokenized Word List:", tokenized_words)

# print filtered words
print("Filtered Word List:", filtered_word_list)
```

This results in the following output:

```
Tokenized Word List: ['Taj', 'Mahal', 'is', 'one', 'of', 'the',
'beautiful', 'monuments', '.', 'It', 'is', 'one', 'of', 'the', 'wonders',
'of', 'the', 'world', '.', 'It', 'was', 'built', 'by', 'Shah', 'Jahan',
'in', '1631', 'in', 'memory', 'of', 'his', 'third', 'beloved', 'wife',
'Mumtaj', 'Mahal', '.']

Filtered Word List: ['Taj', 'Mahal', 'one', 'beautiful', 'monuments', '.',
'It', 'one', 'wonders', 'world', '.', 'It', 'built', 'Shah', 'Jahan',
'1631', 'memory', 'third', 'beloved', 'wife', 'Mumtaj', 'Mahal', '.']
```

In the preceding example, first, we imported the stopwords and loaded the English word list. After this, we iterated the tokenized word list that we generated in the previous section using a `for` loop and filtered the tokenized words from the stop word list using the `if` condition. We saved the filtered words in the `fltered_word_list` list object.

Let's see how to remove stopwords using spaCy:

```
# Import spacy
import spacy

# Loading english language model
nlp = spacy.load("en")

# text paragraph
paragraph = """Taj Mahal is one of the beautiful monuments. It is one of
the wonders of the world. It was built by Shah Jahan in 1631 in memory of
his third beloved wife Mumtaj Mahal."""

# Create nlp Object to handle linguistic annotations in a documents.
my_doc = nlp(paragraph)

# Removing stopwords from text
filtered_token_list = []
for token in my_doc:
     # filter stopwords
     if token.is_stop==False:
         filtered_token_list.append(token)

print("Filtered Word List:",filtered_token_list)
```

This results in the following output:

```
Filtered Sentence: [Taj, Mahal, beautiful, monument, ., wonders, world, .,
built, Shah, Jahan, 1631, memory, beloved, wife, Mumtaj, Mahal, .]
```

In the preceding example, first, we imported the stopwords and loaded the English word list into the stopwords variable. After this, we iterated the NLP object using a `for` loop and filtered each word with the property `"is_stop"` from the stop word list using the `if` condition. We appended the filtered words in the `fltered_token_list` list object. In this section, we have looked at removing stopwords. Now, it's time to learn about stemming and lemmatization to find the root word.

Stemming and lemmatization

Stemming is another step in text analysis for normalization at the language level. The stemming process replaces a word with its root word. It chops off the prefixes and suffixes. For example, the word connect is the root word for connecting, connected, and connection. All the mentioned words have a common root: **connect**. Such differences between word spellings make it difficult to analyze text data.

Lemmatization is another type of lexicon normalization, which converts a word into its root word. It is closely related to stemming. The main difference is that lemmatization considers the context of the word while normalization is performed, but stemmer doesn't consider the contextual knowledge of the word. Lemmatization is more sophisticated than a stemmer. For example, the word "geese" lemmatizes as "goose." Lemmatization reduces words to their valid lemma using a dictionary. Lemmatization considers the part of speech near the words for normalization; that is why it is difficult to implement and slower, while stemmers are easier to implement and faster but with less accuracy.

Let's see how to get stemmed and lemmatized using NLTK:

```
# Import Lemmatizer
from nltk.stem.wordnet import WordNetLemmatizer

# Import Porter Stemmer
from nltk.stem.porter import PorterStemmer
```

```
# Create lemmatizer object
lemmatizer = WordNetLemmatizer()

# Create stemmer object
stemmer = PorterStemmer()

# take a sample word
sample_word = "crying"
print("Lemmatized Sample Word:", lemmatizer.lemmatize(sample_word, "v"))

print("Stemmed Sample Word:", stemmer.stem(sample_word))
```

This results in the following output:

```
Lemmatized Sample Word: cry
Stemmed Sample Word: cri
```

In the preceding example, first, we imported `WordNetLemmatizer` for lemmatization and instantiated its object. Similarly, we imported `PorterStemmer` to stem an instantiate of its object. After this, we got the lemma using the `lemmatize()` function and the stemmed word using the `stem()` function.

Let's see how to get lemmatized words using spaCy:

```
# Import english language model
import spacy

# Loading english language model
nlp = spacy.load("en")

# Create nlp Object to handle linguistic annotations in documents.
words = nlp("cry cries crying")

# Find lemmatized word
for w in words:
    print('Original Word: ', w.text)
    print('Lemmatized Word: ',w.lemma_)
```

This results in the following output:

```
Original Word:  cry
Lemmatized Word:  cry
Original Word:  cries
Lemmatized Word:  cry
Original Word:  crying
Lemmatized Word:  cry
```

In the preceding example, first, we imported the English language model and instantiated it. After this, we created the NLP object and iterated it using a `for` loop. In the loop, we got the text value and its lemma value using the `text` and `lemma_` properties. In this section, we have looked at stemming and lemmatization. Now, we will learn PoS tagging in the given document.

POS tagging

PoS stands for part of speech. The main objective of POS tagging is to discover the syntactic type of words, such as nouns, pronouns, adjectives, verbs, adverbs, and prepositions. PoS tagging finds the relationship among words within a sentence.

Let's see how to get POS tags for words using NLTK:

```
# import Word Tokenizer and PoS Tagger
from nltk.tokenize import word_tokenize
from nltk import pos_tag

# Sample sentence
sentence = "Taj Mahal is one of the beautiful monument."

# Tokenize the sentence
sent_tokens = word_tokenize(sentence)

# Create PoS tags
sent_pos = pos_tag(sent_tokens)

# Print tokens with PoS
print(sent_pos)
```

This results in the following output:

```
[('Taj', 'NNP'), ('Mahal', 'NNP'), ('is', 'VBZ'), ('one', 'CD'), ('of',
'IN'), ('the', 'DT'), ('beautiful', 'JJ'), ('monument', 'NN'), ('.', '.')]
```

In the preceding example, first, we imported `word_tokenize` and `pos_tag`. After this, we took a text paragraph and passed it as a parameter to the `word_tokenize()` method. The output of this method will be a list of words. After this, generate PoS tags for each token using the `pos_tag()` function.

Let's see how to get POS tags for words using spaCy:

```
# Import spacy
import spacy
```

```
# Loading small english language model
nlp = spacy.load("en_core_web_sm")

# Create nlp Object to handle linguistic annotations in a documents.
sentence = nlp(u"Taj Mahal is one of the beautiful monument.")

for token in sentence:
  print(token.text, token.pos_)
```

This results in the following output:

```
Taj PROPN
Mahal PROPN
is VERB
one NUM
of ADP
the DET
beautiful ADJ
monument NOUN
. PUNCT
```

In the preceding example, first, we imported the English language model and instantiated it. After this, we created the NLP object and iterated it using a `for` loop. In the loop, we got the text value and its lemma value using the `text` and `pos_` properties. In this section, we have looked at PoS tags. Now, it's time to jump to recognizing named entities in the text.

Recognizing entities

Entity recognition means extracting or detecting entities in the given text. It is also known as **Named Entity Recognition** (**NER**). An entity can be defined as an object, such as a location, people, an organization, or a date. Entity recognition is one of the advanced topics of NLP. It is used to extract important information from text.

Let's see how to get entities from text using spaCy:

```
# Import spacy
import spacy

# Load English model for tokenizer, tagger, parser, and NER
nlp = spacy.load('en')

# Sample paragraph
paragraph = """Taj Mahal is one of the beautiful monuments. It is one of
the wonders of the world. It was built by Shah Jahan in 1631 in memory of
his third beloved wife Mumtaj Mahal."""
```

```
# Create nlp Object to handle linguistic annotations in documents.
docs=nlp(paragraph)
entities=[(i.text, i.label_) for i in docs.ents]
print(entities)
```

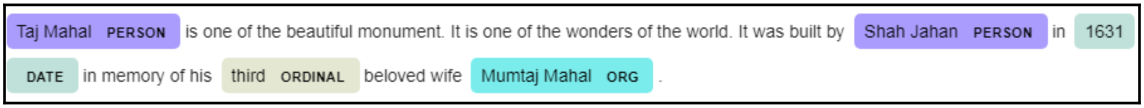
This results in the following output:

```
[('Taj Mahal', 'PERSON'), ('Shah Jahan', 'PERSON'), ('1631', 'DATE'),
('third', 'ORDINAL'), ('Mumtaj Mahal', 'PERSON')]
```

In the preceding example, first, we imported spaCy and loaded the English language model. After this, we created the NLP object and iterated it using a `for` loop. In the loop, we got the text value and its entity type value using the `text` and `label_` properties. Let's visualize the entities in the text using a spaCy display class:

```
# Import display for visualizing the Entities
from spacy import displacy

# Visualize the entities using render function
displacy.render(docs, style = "ent",jupyter = True)
```

This results in the following output:



Taj Mahal PERSON is one of the beautiful monument. It is one of the wonders of the world. It was built by Shah Jahan PERSON in 1631 DATE in memory of his third ORDINAL beloved wife Mumtaj Mahal ORG .

In the preceding example, we imported the display class and called its `render()` method with a NLP text object, `style` as `ent`, and `jupyter` as `True`.

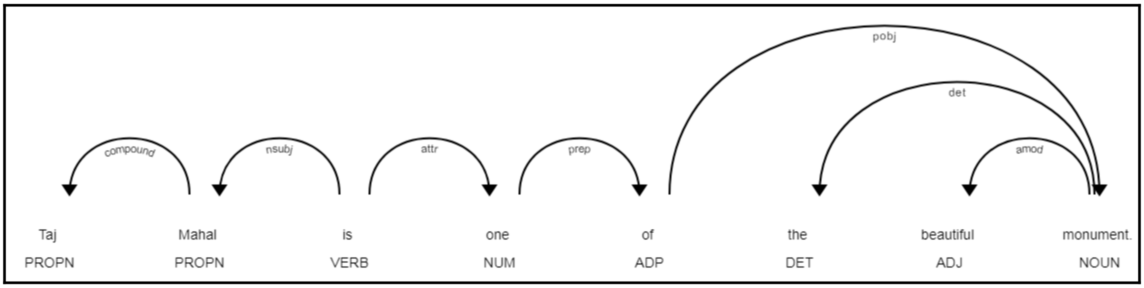
Dependency parsing

Dependency parsing finds the relationship among words – how words are related to each other. It helps computers to understand sentences for analysis; for example, "Taj Mahal is one of the most beautiful monuments." We can't understand this sentence just by analyzing words. We need to dig down and understand the word order, sentence structure, and parts of speech:

```
# Import spacy
import spacy

# Load English model for tokenizer, tagger, parser, and NER
nlp = spacy.load('en')
```

```
# Create nlp Object to handle linguistic annotations in a documents.
docs=nlp(sentence)

# Visualize the using render function
displacy.render(docs, style="dep", jupyter= True, options={'distance':
150})
```

This results in the following output:



In the preceding example, we have imported the display class and called its `render()` method with a NLP text object, `style` as '`dep`', `jupyter` as `True`, and `options` as a dictionary with a distance key and a value of 150. Now, we will see how to visualize text data using a word cloud, based on the word's frequency in the text.

Creating a word cloud

As a data analyst, you need to identify the most frequent words and represent them in graphical form to the top management. A word cloud is used to represent a word-frequency plot. It represents the frequency by the size of the word, that is, the more frequent word looks larger in size and less frequent words looks smaller in size. It is also known as a tag cloud. We can create a word cloud using the `wordcloud` library in Python. We can install it using the following commands:

```
pip install wordcloud
```

Or, alternatively, this one:

```
conda install -c conda-forge wordcloud
```

Let's learn how to create a word cloud:

1. Import libraries and load a stopwords list:

```
# importing all necessary modules
from wordcloud import WordCloud
from wordcloud import STOPWORDS
import matplotlib.pyplot as plt

stopword_list = set(STOPWORDS)

paragraph="""Taj Mahal is one of the beautiful monuments. It is one
of the wonders of the world. It was built by Shah Jahan in 1631 in
memory of his third beloved wife Mumtaj Mahal."""
```

In the preceding example, we imported `WordCloud`, `STOPWORDS`, and `matplotlib.pyplot` classes. We also created the stopword set and defined the paragraph text.

2. Create and generate a word cloud:

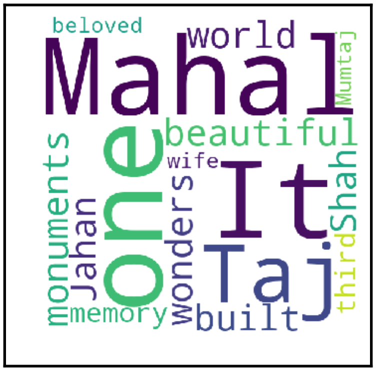
```
word_cloud = WordCloud(width = 550, height = 550,
background_color ='white',
stopwords = stopword_list,
min_font_size = 10).generate(paragraph)
```

After this, the `WordCloud` object with the parameters `width`, `height`, `background_color`, `stopwords`, and `min_font_size` are created and generated the cloud on the paragraph text string.

3. Visualize the word cloud:

```
0.# Visualize the WordCloud Plot
# Set wordcloud figure size
plt.figure(figsize = (8, 6))

# Show image
plt.imshow(word_cloud)

# Remove Axis
plt.axis("off")

# show plot
plt.show()
```

This results in the following output:



In the preceding example, we visualized the word cloud using `matplotlib.pyplot`. Let's learn how to convert text documents into a numeric vector using Bag of Words.

Bag of Words

Bag of Words (**BoW**) is one of the most basic, simplest, and popular feature engineering techniques for converting text into a numeric vector. It works in two steps: collecting vocabulary words and counting their presence or frequency in the text. It does not consider the document structure and contextual information. Let's take the following three documents and understand BoW:

Document 1: I like pizza.

Document 2: I do not like burgers.

Document 3: Pizza and burgers both are junk food.

Now, we will create the **Document Term Matrix** (**DTM**). This matrix consists of the document at rows, words at the column, and the frequency at cell values.

	I	like	pizza	do	not	burgers	and	both	are	junk	food
Doc-1	1	1	1	0	0	0	0	0	0	0	0
Doc-2	1	1	0	1	1	1	0	0	0	0	0
Doc-3	0	0	1	0	0	1	1	1	1	1	1

In the preceding example, we generated the DTM using a single keyword known as a unigram. We can also use a combination of continuous two keywords, known as the bigram model, and three keywords, known as the trigram model. The generalized form is known as the n-gram model.

In Python, scikit-learn offers `CountVectorizer` for generating the BoW DTM. We'll see in the *Sentiment analysis using text classification* section how to generate it using scikit-learn.

TF-IDF

TF-IDF stands for **Term Frequency-Inverse Document Frequency.** It has two segments: **Term Frequency** (**TF**) and **Inverse Document Frequency** (**IDF**). TF only counts the occurrence of words in each document. It is equivalent to BoW. TF does not consider the context of words and is biased toward longer documents. **IDF** computes values that correspond to the amount of information kept by a word.

$$IDF\,(Word) = log_2\left[\frac{Number\ of\ Documents}{Number\ of\ Documents\ that\ Contains\ the\ Word}\right]$$

TF-IDF is the dot product of both segments – TF and IDF. TF-IDF normalizes the document weights. A higher value of TF-IDF for a word represents a higher occurrence in that document. Let's take the following three documents:

Document 1: I like pizza.

Document 2: I do not like burgers.

Document 3: Pizza and burgers both are junk food.

Now, we will create the DTM. This matrix consists of the document name in the row headers, the words in the column headers, and the TF-IDF values in the cells:

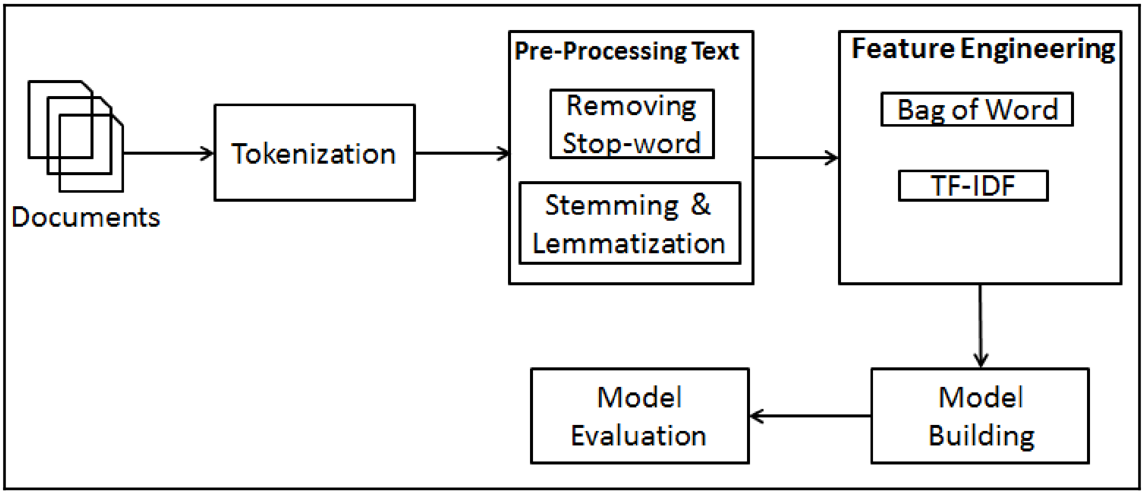
	I	like	pizza	do	not	burgers	and	both	are	junk	food
Doc-1	0.58	0.58	0.58	0	0	0	0	0	0	0	0
Doc-2	0.58	0.58	0	1.58	1.58	0.58	0	0	0	0	0
Doc-3	0	0	0.58	0	0	0.58	1.58	1.58	1.58	1.58	1.58

In Python, scikit-learn offers `TfidfVectorizer` for generating the TF-IDF DTM. Let's see in the upcoming section how to generate it using scikit-learn.

Sentiment analysis using text classification

A business or data analyst needs to understand customer feedback and reviews about a specific product. What did customers like or dislike? And how are sales going? As a business analyst, you need to analyze these things with reasonable accuracy and quantify customer reviews, feedback, opinions, and tweets to understand the target audience. Sentiment analysis extracts the core information from the text and provides people's perception of products, services, brands, and political and social topics. Sentiment analysis is used to understand customers' and people's mindset. It is not only used in marketing, we can also use it in politics, public administration, policy-making, information security, and research. It helps us to understand the polarity of people's feedback. Sentiment analysis also covers words, tone, and writing style.

Text classification can be one of the approaches used for sentiment analysis. It is a supervised method used to detect a class of web content, news articles, blogs, tweets, and sentiments. The classification has a huge number of applications, from marketing, finance, e-commerce, and security. First, we preprocess the text, then we find the features of the preprocessed text, and then we feed features and the labels to the machine learning algorithm to do the classification. The following diagram explains the full idea of sentiment analysis using text classification:



Let's classify the sentiments for Amazon Alexa product reviews. We can get data from the Kaggle website (`https://www.kaggle.com/sid321axn/amazon-alexa-reviews`).

The Alexa product reviews data is a tab-separated values file (TSV file). This data has five columns or attributes – **rating**, **date**, **variation**, **verified_reviews**, and **feedback**.

The `rating` column indicates the user ratings for Alexa products. The date column is the date on which the review was given by the user. The `variation` column represents the product model name. `verified_reviews` has the actual user review about the product.

The rating denotes the rating given by each user to the product. The date is the date of the review, and variation describes the model name. `verified_reviews` contains the text review written by the user, and the feedback column represents the sentiment score, where 1 denotes positive and 0 denotes negative sentiment.

Classification using BoW

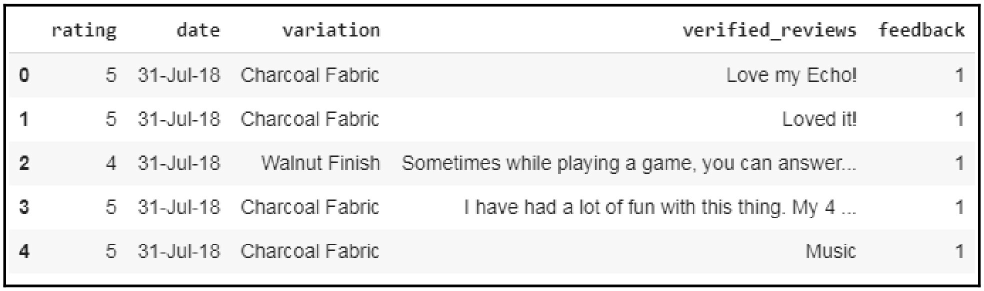
In this subsection, we will perform sentiment analysis and text classification based on BoW. Here, a bag of words is generated using the `scikit-learn` library. Let's see how we perform sentiment analysis using BoW features in the following steps:

1. Load the dataset:

   The first step to build a machine learning model is to load the dataset. Let's first read the data using the pandas `read_csv()` function:

```
# Import libraries
import pandas as pd

# read the dataset
df=pd.read_csv('amazon_alexa.tsv', sep='\t')

# Show top 5-records
df.head()
```

   This results in the following output:



	rating	date	variation	verified_reviews	feedback
0	5	31-Jul-18	Charcoal Fabric	Love my Echo!	1
1	5	31-Jul-18	Charcoal Fabric	Loved it!	1
2	4	31-Jul-18	Walnut Finish	Sometimes while playing a game, you can answer...	1
3	5	31-Jul-18	Charcoal Fabric	I have had a lot of fun with this thing. My 4 ...	1
4	5	31-Jul-18	Charcoal Fabric	Music	1

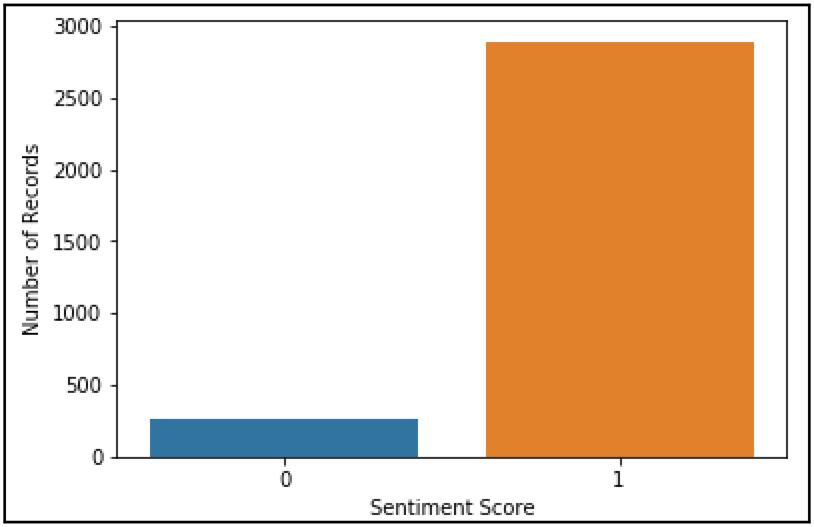
   In the preceding output dataframe, we have seen that the Alexa review dataset has five columns: **rating**, **date**, **variation**, **verified_reviews**, and **feedback**.

2. Explore the dataset.

   Let's plot the **feedback** column count to see how many positive and negative reviews the dataset has:

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

# Count plot
sns.countplot(x='feedback', data=df)

# Set X-axis and Y-axis labels
plt.xlabel('Sentiment Score')
plt.ylabel('Number of Records')

# Show the plot using show() function
plt.show()
```

   This results in the following output:



   In the preceding code, we drew the bar chart for the feedback column using the seaborn `countplot()` function. This function counts and plots the values of the **feedback** column. In this plot, we can observe that 2,900 reviews are positive and 250 reviews are negative feedback.

3. Generating features using `CountVectorizer`:

   Let's generate a BoW matrix for the customer reviews using scikit-learn's `CountVectorizer`:

```
# Import CountVectorizer and RegexTokenizer
from nltk.tokenize import RegexpTokenizer
from sklearn.feature_extraction.text import CountVectorizer

# Create Regex tokenizer for removing special symbols and numeric
values
regex_tokenizer = RegexpTokenizer(r'[a-zA-Z]+')

# Initialize CountVectorizer object
count_vectorizer = CountVectorizer(lowercase=True,
stop_words='english',
ngram_range = (1,1),
tokenizer = regex_tokenizer.tokenize)

# Fit and transform the dataset
count_vectors = count_vectorizer.fit_transform(
df['verified_reviews'])
```

   In the preceding code, we created a `RegexTokenizer` object with an input regular expression that removes the special characters and symbols. After this, the `CountVectorizer` object was created and performed the fit and transform operation on verified reviews. Here, `CountVectorizer` takes parameters such as `lowercase` for converting keywords into lowercase, `stop_words` for specifying a language-specific stopwords list, `ngram_range` for specifying the unigram, bigram, or trigram, and `tokenizer` is used to pass the `tokenizer` object. The `RegexTokenizer` object is passed to the `tokenizer` parameter. Finally, we called the `fit_transform()` function that converts text reviews into a DTM as per specified parameters.

4. Split train and test set:

   Let's split the feature set and target column into `feature_train`, `feature_test`, `target_train`, and `target_test` using `train_test_split()`. `train_test_split()` takes dependent, independent dataframes, `test_size` and `random_state`. Here, `test_size` will decide the ratio of the train-test split (that is, `test_size 0.3` means 30% for the testing set and the remaining 70% will be the training set), and `random_state` is used as a seed value for reproducing the same data split each time. If `random_state` is `None`, then it will randomly split the records each time, which will give different performance measures:

```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Partition data into training and testing set
feature_train, feature_test, target_train, target_test =
train_test_split(count_vectors, df['feedback'], test_size=0.3,
random_state=1)
```

   In the preceding code, we are partitioning the feature set and target column into `feature_train`, `feature_test`, `target_train`, and `target_test` using the `train_test_split()` method.

5. Classification Model Building using Logistic Regression:

   In this section, we will build the logistic regression model to classify the review sentiments using BoW (or `CountVectorizer`). Let's create the logistic regression model:

```
# import logistic regression scikit-learn model
from sklearn.linear_model import LogisticRegression

# Create logistic regression model object
logreg = LogisticRegression(solver='lbfgs')

# fit the model with data
logreg.fit(feature_train,target_train)

# Forecast the target variable for given test dataset
predictions = logreg.predict(feature_test)
```

In the preceding code, we imported `LogisticRegression` and created the `LogisticRegression` object. After creating the model object, we performed the `fit()` operation on the training data and `predict()` to forecast the sentiment for the test dataset.

6. Evaluate the Classification Model:

   Let's evaluate the classification model using the `metrics` class and its methods – `accuracy_score`, `precision_score`, and `recall_score`:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

# Assess model performance using accuracy measure
print("Logistic Regression Model
Accuracy:",accuracy_score(target_test, predictions))

# Calculate model precision
print("Logistic Regression Model
Precision:",precision_score(target_test, predictions))

# Calculate model recall
print("Logistic Regression Model Recall:",recall_score(target_test,
predictions))

# Calculate model f1 score
print("Logistic Regression Model F1-Score:",f1_score(target_test,
predictions))
```

   This results in the following output:

```
Logistic Regression Model Accuracy: 0.9428571428571428
Logistic Regression Model Precision: 0.952433628318584
Logistic Regression Model Recall: 0.9873853211009175
Logistic Regression Model F1-Score: 0.9695945945945945
```

In the preceding code, we have evaluated the model performance using accuracy, precision, recall, and f1-score using the `scikit-learn metrics` function. All the measures are greater than 94%, so we can say that our model is performing well and classifying both the sentiment levels with a good amount of precision and recall.

Classification using TF-IDF

In this subsection, we will perform sentiment analysis and text classification based on TF-IDF. Here, TF-IDF is generated using the `scikit-learn` library. Let's see how we perform sentiment analysis using TF-IDF features using the following steps:

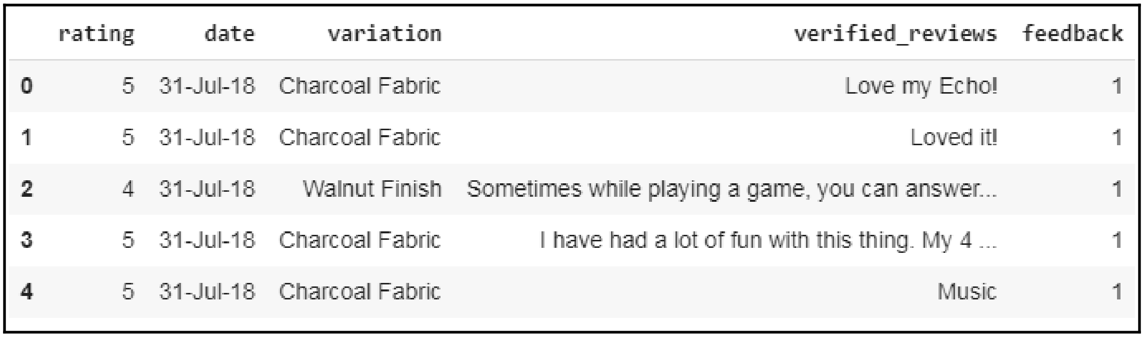
1. Load the dataset:

   The first step for building a machine learning model is to load the dataset.

   Let's first read the data using the pandas `read_csv()` function:

```
# Import libraries
import pandas as pd

# read the dataset
df=pd.read_csv('amazon_alexa.tsv', sep='\t')

# Show top 5-records
df.head()
```

   This results in the following output:



	rating	date	variation	verified_reviews	feedback
0	5	31-Jul-18	Charcoal Fabric	Love my Echo!	1
1	5	31-Jul-18	Charcoal Fabric	Loved it!	1
2	4	31-Jul-18	Walnut Finish	Sometimes while playing a game, you can answer...	1
3	5	31-Jul-18	Charcoal Fabric	I have had a lot of fun with this thing. My 4 ...	1
4	5	31-Jul-18	Charcoal Fabric	Music	1

   In the preceding output dataframe, we have seen that the Alexa review dataset has five columns: **rating**, **date**, **variation**, **verified_reviews**, and **feedback**.

2. Feature generation using `TfidfVectorizer`:

   Let's generate a TF-IDF matrix for the customer reviews using scikit-learn's `TfidfVectorizer`:

```
# Import TfidfVectorizer and RegexTokenizer
from nltk.tokenize import RegexpTokenizer
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
# Create Regex tokenizer for removing special symbols and numeric
values
regex_tokenizer = RegexpTokenizer(r'[a-zA-Z]+')

# Initialize TfidfVectorizer object
tfidf = TfidfVectorizer(lowercase=True, stop_words
='english',ngram_range = (1,1),tokenizer =
regex_tokenizer.tokenize)

# Fit and transform the dataset
text_tfidf = tfidf.fit_transform(df['verified_reviews'])
```

In the preceding code, we created a `RegexTokenizer` object with an input regular expression that removes the special characters and symbols. After this, the `TfidfVectorizer` object was created and performed the fit and transform operation on verified reviews. Here, `TfidfVectorizer` takes parameters such as `lowercase` for converting keywords into lowercase, `stop_words` for a specified language-specific stopwords list, `ngram_range` for specifying the unigram, bigram, or trigram, and `tokenizer` is used to pass the `tokenizer` object. The `RegexTokenizer` object is passed to the `tokenizer` parameter. Finally, we called the `fit_transform()` function that converts text reviews into a DTM as per specified parameters.

3. Split the training and testing datasets:

Let's split the feature set and target column into `feature_train`, `feature_test`, `target_train`, and `target_test` using `train_test_split()`. `train_test_split()` takes dependent, independent dataframes, `test_size` and `random_state`. Let's split the dataset into a training and testing set:

```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Partition data into training and testing set
from sklearn.model_selection import train_test_split

feature_train, feature_test, target_train, target_test =
train_test_split(text_tfidf, df['feedback'], test_size=0.3,
random_state=1)
```

In the preceding code, we partition the feature set and target column into `feature_train`, `feature_test`, `target_train`, and `target_test` using the `train_test_split()` method.

4. Classification model building using logistic regression:

   In this section, we will build the logistic regression model to classify the review sentiments using TF-IDF. Let's create the logistic regression model:

```
# import logistic regression scikit-learn model
from sklearn.linear_model import LogisticRegression

# instantiate the model
logreg = LogisticRegression(solver='lbfgs')

# fit the model with data
logreg.fit(feature_train,target_train)

# Forecast the target variable for given test dataset
predictions = logreg.predict(feature_test)
```

   In the preceding code, we imported `LogisticRegression` and created the `LogisticRegression` object. After creating the model object, we performed a `fit()` operation on the training data and `predict()` to forecast the sentiment for the test dataset.

5. Evaluate the classification model:

   Let's evaluate the classification model using the `metrics` class and its methods – `accuracy_score`, `precision_score`, and `recall_score`:

```
# Import metrics module for performance evaluation
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

# Assess model performance using accuracy measure
print("Logistic Regression Model
Accuracy:",accuracy_score(target_test, predictions))

# Calculate model precision
print("Logistic Regression Model
Precision:",precision_score(target_test, predictions))

# Calculate model recall
```

```
print("Logistic Regression Model Recall:",recall_score(target_test,
predictions))

# Calculate model f1 score
print("Logistic Regression Model F1-Score:",f1_score(target_test,
predictions))
```

This results in the following output:

```
Logistic Regression Model Accuracy: 0.9238095238095239
Logistic Regression Model Precision: 0.923728813559322
Logistic Regression Model Recall: 1.0
Logistic Regression Model F1-Score: 0.960352422907489
```

In the preceding code, we evaluated the model performance using accuracy, precision, recall, and f1-score using the scikit-learn `metrics` function. All the measures are greater than 94%, so we can say that our model is performing well and classifying both sentiment levels with a good amount of precision and recall. In this section, we have looked at sentiment analysis using text classification. Text classification is performed using BoW and TF-IDF features. In the next section, we will learn how to find similarities between two pieces of text, such as sentences or paragraphs.

Text similarity

Text similarity is the process of determining the two closest texts. Text similarity is very helpful in finding similar documents, questions, and queries. For example, a search engine such as Google uses similarity to find document relevance, and Q&A systems such as StackOverflow or a consumer service system use similar questions. There are two common metrics used for text similarity, namely Jaccard and cosine similarity.

We can also use the similarity method available in spaCy. The `nlp` object's `similarity` method returns a score between two sentences. Let's look at the following example:

```
# Import spacy
import spacy

# Load English model for tokenizer, tagger, parser, and NER
nlp = spacy.load('en')

# Create documents
doc1 = nlp(u'I love pets.')
doc2 = nlp(u'I hate pets')
```

```
# Find similarity
print(doc1.similarity(doc2))
```

This results in the following output:

```
0.724494176985974

<ipython-input-32-f157deaa344d>:12: UserWarning: [W007] The model you're
using has no word vectors loaded, so the result of the Doc.similarity
method will be based on the tagger, parser and NER, which may not give
useful similarity judgements. This may happen if you're using one of the
small models, e.g. `en_core_web_sm`, which don't ship with word vectors and
only use context-sensitive tensors. You can always add your own word
vectors, or use one of the larger models instead if available.
```

In the preceding code block, we have found the similarity between two sentences using spaCy's `similarity()` function. Spacy's similarity function does not give better results with small models (such as the `en_core_web_sm` and `en` models); that's why you will get a warning: **UserWarning: [W007].** To remove this warning, use larger models such as `en_core_web_lg`.

Jaccard similarity

Jaccard similarity calculates the similarity between two sets by the ratio of common words (intersection) to totally unique words (union) in both sets. It takes a list of unique words in each sentence or document. It is useful where the repetition of words does not matter. Jaccard similarity ranges from 0-100%; the higher the percentage, the more similar the two populations:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Let's look at a Jaccard similarity example:

```
def jaccard_similarity(sent1, sent2):
    """Find text similarity using jaccard similarity"""
    # Tokenize sentences
    token1 = set(sent1.split())
    token2 = set(sent2.split())
    # intersection between tokens of two sentences
    intersection_tokens = token1.intersection(token2)
    # Union between tokens of two sentences
    union_tokens=token1.union(token2)
    # Cosine Similarity
    sim_= float(len(intersection_tokens) / len(union_tokens))
    return sim_

jaccard_similarity('I love pets.','I hate pets.')
```

This results in the following output:

```
0.5
```

In the preceding example, we have created a function, `jaccard_similarity()`, which takes two arguments, `sent1` and `sent2`. It will find the ratio between the intersection of keywords and the union of keywords between two sentences.

Cosine similarity

Cosine similarity computes the cosine of the angle between two multidimensional projected vectors. It indicates how two documents are related to each other. Two vectors can be made of the bag of words or TF-IDF or any equivalent vector of the document. It is useful where the duplication of words matters. Cosine similarity can measure text similarity irrespective of the size of documents.

$$Cosine\ Similarity(A, B) = \frac{A.B}{|A|\,.\,|B|}$$

Let's look at a cosine similarity example:

```
# Let's import text feature extraction TfidfVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
docs=['I love pets.','I hate pets.']

# Initialize TfidfVectorizer object
tfidf= TfidfVectorizer()

# Fit and transform the given data
tfidf_vector = tfidf.fit_transform(docs)

# Import cosine_similarity metrics
from sklearn.metrics.pairwise import cosine_similarity

# compute similarity using cosine similarity
cos_sim=cosine_similarity(tfidf_vector, tfidf_vector)

print(cos_sim)
```

This results in the following output:

```
[[1. 0.33609693]
[0.33609693 1. ]]
```

In the preceding example, first, we import `TfidfVectorizer` and generate the TF-IDF vector for given documents. After this, we apply the `cosine_similarity()` metric on the document list and get similarity metrics.

Summary

In this chapter, we explored text analysis using NLTK and spaCy. The main focus was on text preprocessing, sentiment analysis, and text similarity. The chapter started with text preprocessing tasks such as text normalization, tokenization, removing stopwords, stemming, and lemmatization. We also focused on how to create a word cloud, recognize entities in a given text, and find dependencies among tokens. In later sections, we focused on BoW, TFIDF, sentiment analysis, and text classification.

The next chapter, Chapter 13, *Analyzing Image Data*, focuses on image processing, basic image processing operations, and face detection using OpenCV. The chapter starts with image color models, and image operations such as drawing on an image, resizing an image, and flipping and blurring an image. In later sections, the focus will be on face detection in a given input image.

13
Analyzing Image Data

We are in the age of information, where every movement will generate data in a variety of formats, such as text, images, geospatial data, and videos. Smartphones have reached rural areas of the world and people are capturing activities, especially in images and videos, and sharing them on social media platforms. This is how lots of big chunks of data are generated and most of the data is in image and video formats. Industry and research institutes want to analyze image and video datasets to generate value and make automated solutions to reduce costs. Image processing and computer vision are fields that explore and develop image- and video-based solutions. There are lots of opportunities for research, innovation, and start-ups in the area of computer vision. In this chapter, we focus on the basics of image processing to build your fundamental knowledge in the computer vision area.

Image processing is a subset of computer vision. Computer vision is an advanced and more powerful field within machine learning and artificial intelligence. Computer vision offers enormous applications, such as detecting objects, classifying images and objects, image captioning, and image segmentation. An image can be defined as two-dimensional signals in signal processing, a set of points in 2D or 3D in geometry, and a two-dimensional or three-dimensional NumPy array in Python. Image processing refers to processing image data and performing operations such as drawing, writing, resizing, flipping, blurring, changing the brightness, and detecting faces. In this chapter, we will focus on all these image processing operations in detail.

We will cover the following topics in this chapter:

- Installing OpenCV
- Understanding image data
- Color models
- Drawing on images
- Writing on images
- Resizing images
- Flipping images

- Changing the brightness
- Blurring an image
- Face detection

Technical requirements

This chapter has the following technical requirements:

- You can find the code, face classifier file, and the datasets at the following Github link: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter13`.
- All the code blocks are available in the `ch13.ipynb` file.
- This chapter uses `.jpg`/`.jpeg` files (`google.jpg`, `image.jpg`, `messi.png`, `nature.jpeg`, `barcelona.jpeg`, and `tajmahal.jpg`) for practice purposes.
- This chapter uses one face classifier XML file (`haarcascade_frontalface_default.xml`).
- In this chapter, we will use the OpenCV, NumPy, and matplotlib Python libraries.

Installing OpenCV

OpenCV is an open source library for computer vision operations such as image and video analysis. OpenCV is primarily developed by Intel in C++ and offers interfaces with Python, Java, and Matlab. OpenCV has the following features:

- It is an open source image processing Python library.
- OpenCV is the core Python library for image processing and computer vision.
- OpenCV is easy to learn and deploy with web and mobile applications.
- OpenCV in Python is an API and wrapper around its C++ core implementation.
- It is fast due to background C++ code.

We can install OpenCV using the following command:

```
pip install opencv-python
```

Using the preceding pip command, we can easily install OpenCV. OpenCV is the most popular library for image processing and computer vision tasks. It offers various use cases related to image analysis operations such as improving image quality, filtering and transforming images, drawing on images, changing colors, detecting faces and objects, identifying human actions, tracking objects, analyzing motion, and finding similar images. After installing the OpenCV library, it's time to understand the basics of image processing.

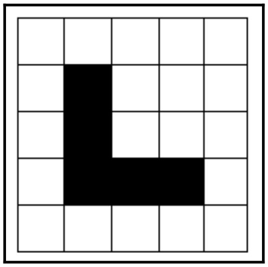
Understanding image data

Image data is a two-dimensional array or function `f(x,y)` with spatial coordinates. The amplitude of the `coordinate(x,y)` is known as intensity. In Python, an image is a 2D or 3D NumPy array with pixel values. Pixels are the smallest, core tiny picture elements, which decide the image quality. A large number of pixels results in a higher resolution. Also, there are various image formats available, such as `.jpeg`, `.png`, `.gif`, and `.tiff`. These file formats are helpful in organizing and maintaining digital image files. Before analyzing image data, we need to understand the types of images. Image data can be of three types:

- Binary
- Grayscale
- Color

Binary images

Binary image pixels have only two colors, generally black and white. Binary image pixels take only binary values 0 or 1.



The preceding image is an example of a binary image. It has only two colors, black and white. It does not use shades of black and white.

Grayscale images

A grayscale image looks like a black and white image. It is represented by 8 bits per pixel, that is, 256 intensity values or tones ranging from 0 to 255. These 256 shades move from pure black to pure white; 0 represents pure black while 255 represents the color white.

The preceding image is a grayscale image. It is a black and white image where shades move from pure black to pure white.

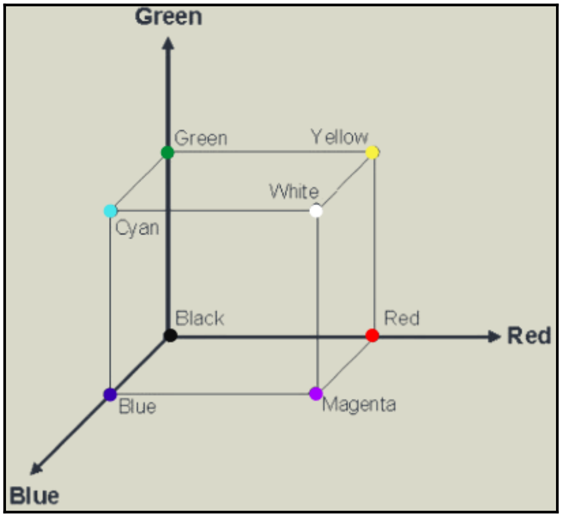
Color images

Color images are a mixture of the primary colors red, blue, and green. These primary colors have the capability to form new colors by blending in certain proportions. Each color uses eight bits (intensity values between 0-255), that is, 24 bits per pixel. Let's take an example of a color image:

In the preceding image file, we can see most of the color shades with different intensities. After understanding the type of images, it's time to understand color models such as RGB, CMYK, HSV, HSL, and grayscale. Let's jump to color models.

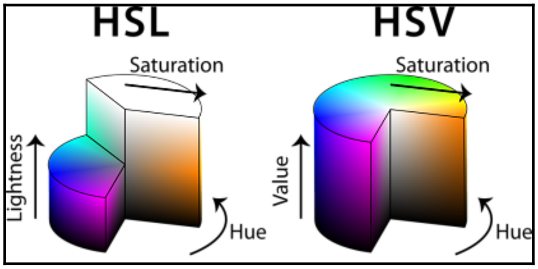
Color models

Color models are a structure for processing and measuring the combination of primary colors. They help us to explain how colors will display on the computer screen or on paper. Color models can be of two types: additive or subtractive. Additive models are used for computer screens, for example, the RGB (red, green, and blue) model, and subtractive models are used for printing images, for example, the CMYK (cyan, magenta, yellow, and black) model:



There are lots of models other than RGB and CMYK, such as HSV, HSL, and Gray Scale. HSV is an acronym for hue, saturation, and value. It is a three-dimensional color model, which is an improved version of the RGB model. In the HSV model, the top of the center axis is white, the bottom is black, and the remaining colors lie in between. Here, the hue is the angle, saturation is the distance from the center axis, and value is the distance from the bottom of the axis.

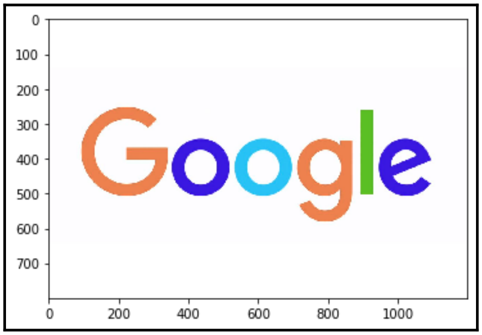
HSL is an acronym for hue, saturation, and lightness. The main difference between HSV and HSL is the amount of lightness and the value of colors from the center axis.



Let's learn how to read and display the image file:

```
# Import cv2 latest version of OpenCV library
import cv2

# Import numeric python (NumPy) library
import numpy as np

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# magic function to render the figure in a notebook
%matplotlib inline

# Read image using imread() function
image = cv2.imread('google.jpg')

# Let's check image data type
print('Image Type:',type(image))

# Let's check dimension of image
print('Image Dimension:',image.shape)

# Let's show the image
plt.imshow(image)
plt.show()
```

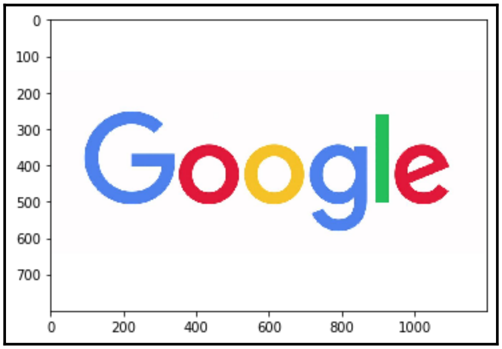
This results in the following output:



In the preceding example, we imported cv2, NumPy, and matplotlib.pyplot. cv2 is for image processing, NumPy is for arrays, and matplotlib.pyplot is for displaying an image. We read the image using the imread() function and returned an array of images. We can check its type using the type() function and its shape using the shape attribute of the NumPy array. We can display the image using the show() function of the matpltlib.pyplot module. The preceding image is not showing the correct colors of the Google logo image. This is because imread() reads images in the BGR color model. Lets convert BGR to the RGB color model using the cvtColor() function and passing tthe flag cv2.COLOR_BGR2RGB:

```
# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(rgb_image)
plt.show()
```

This results in the following output:



Here, you can see the correct image in RGB format.

Let's write the image file on a local disk using the `imwrite()` function:

```
# Write image using imwrite()
cv2.imwrite('image.jpg',image)

Output: True
```

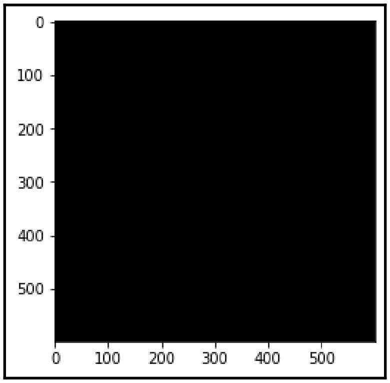
In the preceding code block, we have written the image file on a local disk with the image name `image.jpg`. After understanding color models, it's time to learn how to draw elements on an image.

Drawing on images

Let's learn how to draw different figure shapes, such as a line, square, or triangle, on an image using OpenCV. When we draw any shape on an image, we need to take care of the coordinates, color, and thickness of the shape. Let's first create a blank image with a white or black background:

```
# Import cv2 latest version of OpenCV library
import cv2

# Import numeric python (NumPy) library
import numpy as np

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# Magic function to render the figure in a notebook
%matplotlib inline

# Let's create a black image
image_shape=(600,600,3)
black_image = np.zeros(shape=image_shape,dtype=np.int16)

# Show the image
plt.imshow(black_image)
```

This results in the following output:

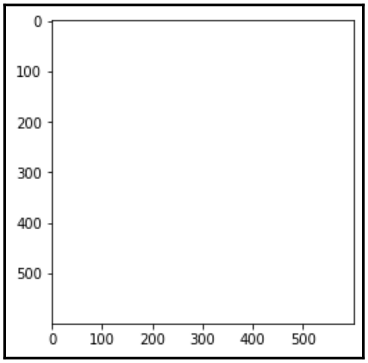


In the preceding example, we created a blank image with a black background using the `zeros()` function of the NumPy module. The `zeros()` function creates an array of the given size and fills the matrix with zeros.

Let's create a blank image with a white background:

```
# Create a white image
image_shape=(600,600,3)
white_image = np.zeros(shape=image_shape,dtype=np.int16)

# Set every pixel of the image to 255
white_image.fill(255)

# Show the image
plt.imshow(white_image)
```

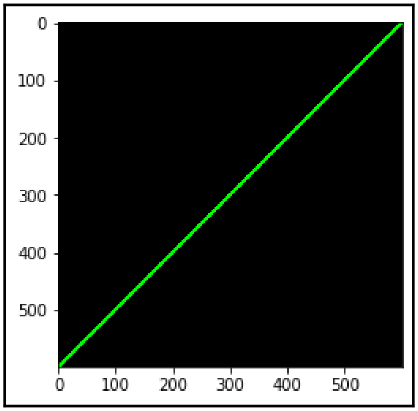
This results in the following output:



In the preceding example, we created a blank image with a white background using the `zeros()` function of the NumPy module and filled the image with 255 for each pixel. The `zeros()` function creates an array of the given size and fills the matrix with zeros. The `fill()` function assigns a given value to all the elements of the matrix. Let's draw a line using OpenCV on a black image:

```
# Draw a line on black image
line = cv2.line(black_image,(599,0),(0,599),(0,255,0),4)

# Show image
plt.imshow(line)
```

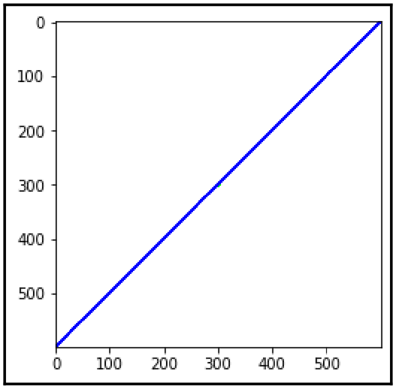
This results in the following output:



In the preceding example, we drew the green line on the black image using the `line()` function. The `line()` function takes the following arguments: image file, `start_point`, `end_point`, color, and thickness. In our example, the start and endpoints are (599,0) and (0,599), the color tuple is (0,255,0), and the thickness is 4. Similarly, we can create a line on a white image. Let's see the following example:

```
# Let's draw a blue line on white image

line = cv2.line(white_image,(599,0),(0,599),(0,0,255),4)
# Show the image
plt.imshow(line)
```

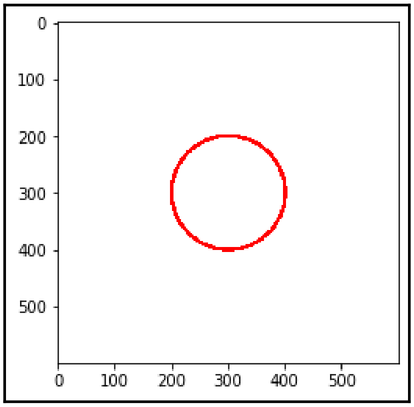
This results in the following output:



Let's see an example of drawing a circle on a white image:

```
# Let's create a white image
img_shape=(600,600,3)
white_image = np.zeros(shape=image_shape,dtype=np.int16)

# Set every pixel of the image to 255
white_image.fill(255)

# Draw a red circle on white image
circle=cv2.circle(white_image,(300, 300), 100, (255,0,0),6)

# Show the image
plt.imshow(circle)
```

This results in the following output:

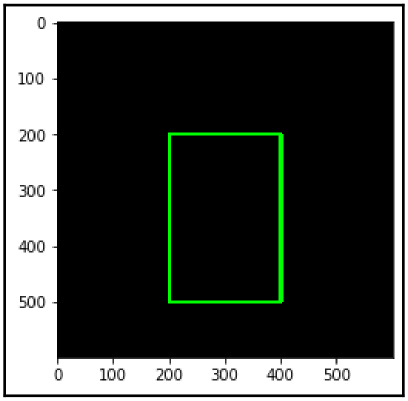


In the preceding example, we created a white image and drew a circle using the `circle()` function. The `circle()` function takes the following arguments: image, `center_coordinates`, radius, color, and thickness. In our example, the center is (300, 300), the radius is 100, a color tuple is (255,0,0), and the thickness is 6.

Let's see an example of drawing a rectangle on a black image:

```
# Let's create a black image
img_shape=(600,600,3)
black_image = np.zeros(shape=image_shape,dtype=np.int16)

# Draw a green rectangle on black image
rectangle= cv2.rectangle(black_image,(200,200),(400,500),(0,255,0),5)

# Show the image
plt.imshow(rectangle)
```

This results in the following output:

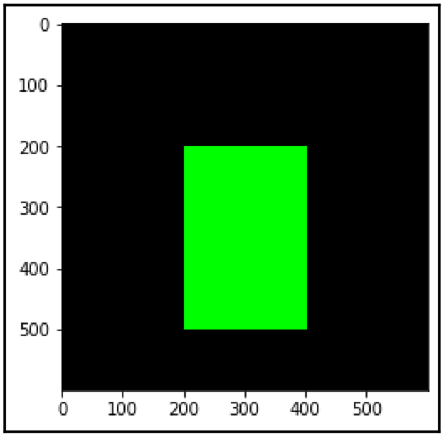


In the preceding example, we created a black image and drew a rectangle using the `rectangle()` function. The `rectangle()` function takes the following arguments: image, `start_point`, `end_point`, color, and thickness. Here, thickness also takes an argument `-1`, the `-1` px value will fill the rectangle shape with the specified color. Let's see an example of a filled rectangle:

```
# Let's create a black image
img_shape=(600,600,3)
black_image = np.zeros(shape=image_shape,dtype=np.int16)

# Draw a green filled rectangle on black image
rectangle= cv2.rectangle(black_image,(200,200),(400,500),(0,255,0),-1)
```

```
# Show the image
plt.imshow(rectangle)
```

This results in the following output:



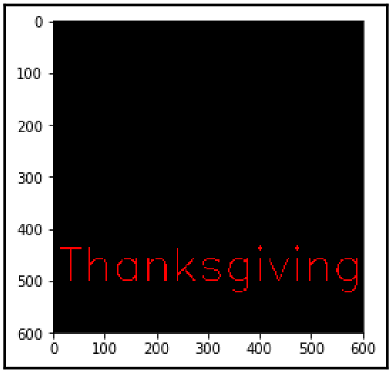
In the preceding example, we filled the rectangle by passing thickness values as -1 px. In a nutshell, we can say that the line takes mainly the start and endpoints as the input, the rectangle takes the top-left and the bottom-right coordinates, and the circle takes center coordinates and radius values.

Writing on images

In the previous section, we created various shapes on images. Now, we will learn how to write text on images. Writing text on an image is similar to drawing shapes. Let's see an example of writing on an image:

```
# Let's create a black image
img_shape=(600,800,3)
black_image = np.zeros(shape=image_shape,dtype=np.int16)

# Write on black image
text = cv2.putText(black_image,'Thanksgiving',(10,500),
cv2.FONT_HERSHEY_SIMPLEX, 3,(255,0,0),2,cv2.LINE_AA)

# Display the image
plt.imshow(text)
```

This results in the following output:



In the preceding example, we created a blank image with the color black. We have written text on an image using the `putText()` function. The `putText()` function will take the following arguments: image, text, coordinates of the bottom-left corner, font, `fontScale`, color, thickness, and `linetype`.

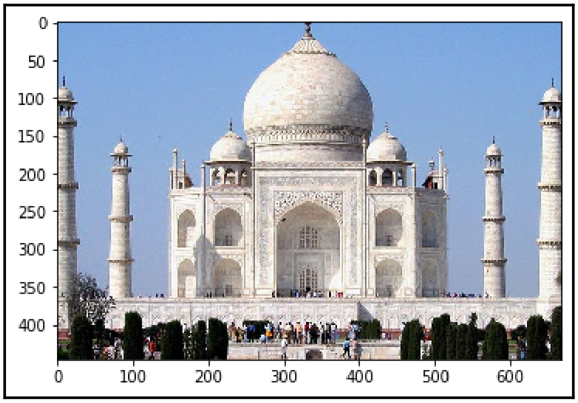
Resizing images

Resizing an image means changing the dimension or scaling of a given image. Scaling or resizing is done either from the width, height, or both. One of the applications of resizing images is training deep learning models where reduced image sizes can speed up the training. Training a deep learning model is out of the scope of this book. If you are interested, then you can refer to any deep learning book from Packt Publishing. Let's see an example of resizing an image:

```
# Import cv2 module
import cv2

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# magic function to render the figure in a notebook
%matplotlib inline

# read image
image = cv2.imread('tajmahal.jpg')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
```

```
# Display the image
plt.imshow(rgb_image)
```

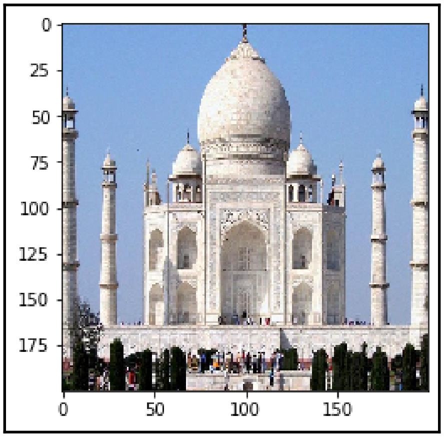
This results in the following output:



In the preceding code, we read the image and converted it from BGR into the RGB space. Let's resize it now using the `resize()` function:

```
# Resize the image
image_resized = cv2.resize(rgb_image, (200, 200))
interpolation = cv2.INTER_NEAREST

# Display the image
plt.imshow(image_resized)
```

This results in the following output:



In the preceding example, we read the image, converted BGR to RGB color using the `cvtColor()` function, and resized the image using the `resize()` function. The `resize()` function takes the following arguments: image, size, and interpolation. Interpolation is used to scale moire-free images. Interpolation takes one of the following flags: `INTER_NEAREST` (for nearest-neighbor interpolation), `INTER_LINEAR` (bilinear interpolation), and `INTER_AREA` (resampling using pixel area relation).

Flipping images

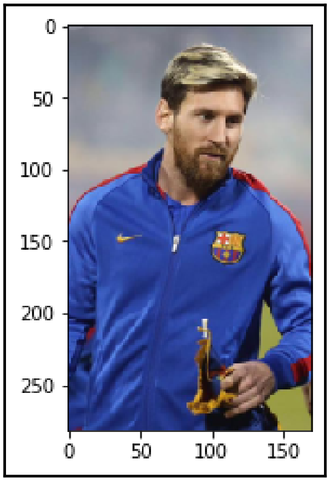
Flipping an image is equivalent to a mirror effect. Let's learn how to flip an image across the *x* axis (vertical flipping), *y* axis (horizontal flipping), or both axes. OpenCV offers the `flip()` function to flip an image. The `flip()` function will take two arguments: image and flipcode. The image is a NumPy array of pixel values and the flipcode used defines the type of flip, such as horizontal, vertical, or both. The following flipcode values are for different types of flips:

- Flipcode > 0 is for a horizontal flip.
- Flipcode = 0 is for a vertical flip.
- Flipcode < 0 is for both a horizontal and vertical flip.

Let's see an example of flipping an image:

```
# Import OpenCV module
import cv2

# Import NumPy
import numpy as np

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# magic function to render the figure in a notebook
%matplotlib inline

# Read image
image = cv2.imread('messi.png')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(rgb_image)
```

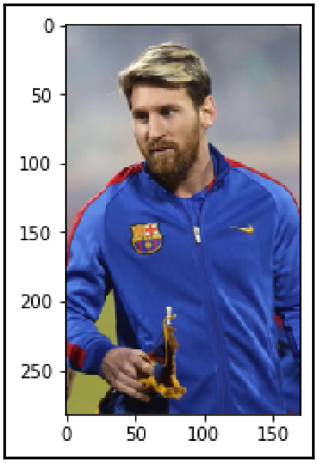
This results in the following output:



This is the original image, of Lionel Messi. Let's flip it horizontally using the `flip()` function by passing 1 as the flipcode in the `flip()` function:

```
# Flipping image (Horizontal flipping)
image_flip = cv2.flip(rgb_image, 1)

# Display the image
plt.imshow(image_flip)
```

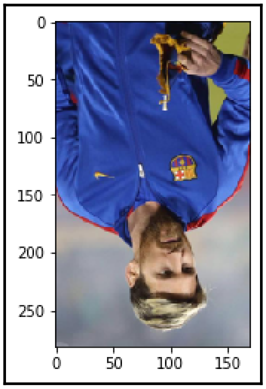
This results in the following output:



This is the horizontally flipped image. Let's flip the original image vertically:

```
# Flipping image (Vertical flipping)
image_flip = cv2.flip(rgb_image,0)

# Display the image
plt.imshow(image_flip)
```

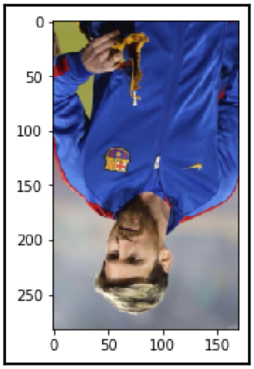
This results in the following output:



You can see the vertically flipped image. Let's flip the original image on both axes:

```
# Flipping image (Horizontal and vertical flipping)
image_flip = cv2.flip(rgb_image, -1)

# Display the image
plt.imshow(image_flip)
```

This results in the following output:



You can see the vertically and horizontally flipped image. After flipping the image, let's learn how to change the brightness of the image in the next section.

Changing the brightness

Brightness is a comparative term that is determined by visual perception. Sometimes it is difficult to perceive the brightness. The value of pixel intensity can help us to find a brighter image. For example, if two pixels have the intensity values 110 and 230, then the latter one is brighter.

In OpenCV, adjusting image brightness is a very basic operation. Brightness can be controlled by changing the intensity of each pixel in an image:

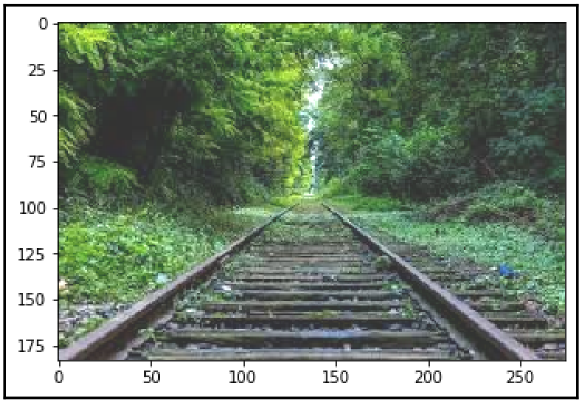
```
# Import cv2 latest version of OpenCV library
import cv2

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# Magic function to render the figure in a notebook
%matplotlib inline

# Read image
image = cv2.imread('nature.jpeg')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(rgb_image)
```

This results in the following output:

In the preceding code example, we have read the image and converted the BGR color model-based image into an RGB color model-based image. Let's change the brightness of the image in the following code block:

```
# set weightage for alpha and betaboth the matrix
alpha_=1
beta_=50

# Add weight to the original image to change the brightness
image_change=cv2.addWeighted(rgb_image, alpha_,
np.zeros(image.shape,image.dtype),0, beta_)

# Display the image
plt.imshow(image_change)
```

This results in the following output:



In the preceding example, we added the two matrices with the given weightage; alpha and beta using the `addWeighted()` function. `addWeighted()` takes the following arguments: `first_image`, `alpha`, `second_image`, `gamma`, and `beta`. In our example, the argument `first_image` input image and the argument `second_image` is the null matrix. The values of `alpha` and `beta` are the weights for both matrices and `gamma` is 0.

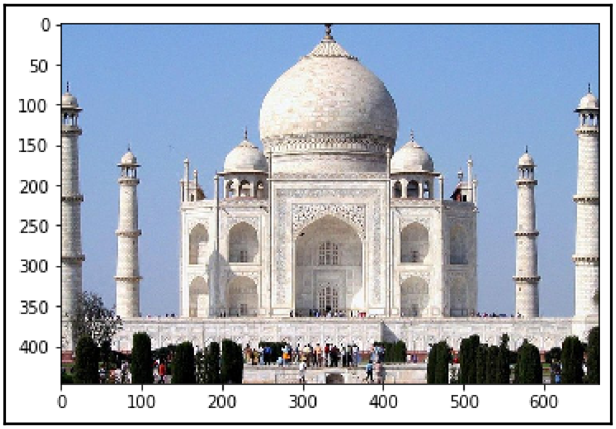
Blurring an image

Blurring is one of the crucial steps of image preprocessing. In preprocessing, the removal of noise impacts the performance of algorithms. Blurring is the process of reducing noise in image data to achieve better accuracy. Blurring also helps us to take charge of handling pixel intensity.

Let's see an example of blurring an image:

```
# Import OpenCV module
import cv2

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# Magic function to render the figure in a notebook
%matplotlib inline

# Read image
image = cv2.imread('tajmahal.jpg')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Display the image
plt.imshow(rgb_image)
```

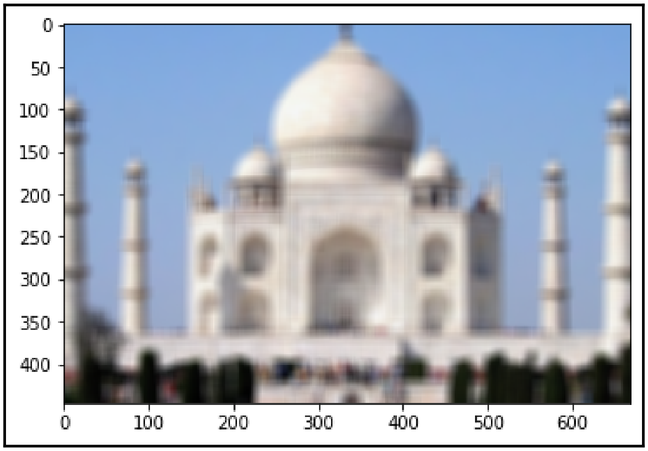
This results in the following output:



In the preceding code sample, we read the image and converted it from a BGR to RGB based image. Let's blur it using the `blur()` function. Blur takes two arguments: image and kernel size. The `blur()` function uses the average blurring method:

```
# Blur the image using blur() function
image_blur = cv2.blur(rgb_image,(15,15))

# Display the image
plt.imshow(image_blur)
```

This results in the following output:



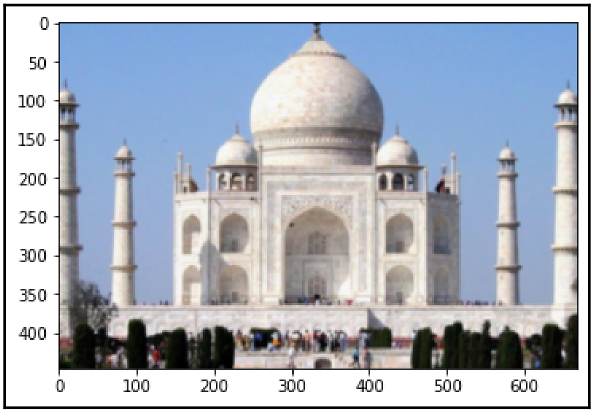
In the preceding example, we read the image, converted BGR to RGB color using the `cvtColor()` function, and displayed the image. Here, we blurred the image using the `blur()` function. The `blur()` function applies average blurring, which uses a normalized box filter. The `blur()` function takes the following arguments: image and kernel size.

We have seen a blurred image using average blurring. Let's explore blurring using Gaussian blurring. In this blurring, the Gaussian kernel is used instead of a box filter. `GaussianBlur()` will take the image and kernel size. The kernel size will be a tuple of the width and height. Both width and height must be a positive and odd number:

```
# Import cv2 module
import cv2

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# magic function to render the figure in a notebook
%matplotlib inline

# read image
image = cv2.imread('tajmahal.jpg')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Blurring the image using Gaussian Blur
image_blur = cv2.GaussianBlur(rgb_image, (7,7), 0)
```

```
# Display the image
plt.imshow(image_blur)
```

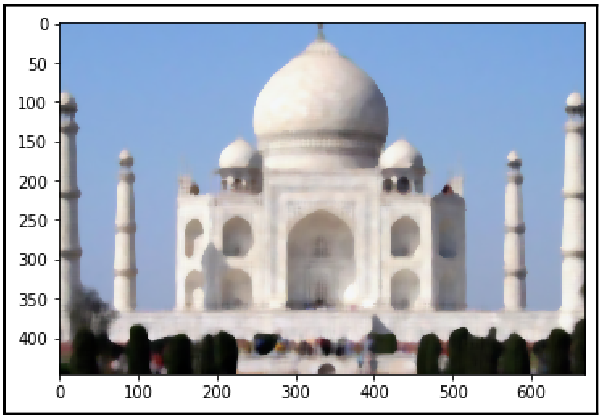
This results in the following output:



Let's explore the median blurring of the given image. Median blur takes pixels in the kernel area and replaces the central element with the median value. `medianBlur()` will take image and kernel size as an argument. It is recommended that the kernel size should be an odd number and greater than 1, for example, 3, 5, 7, 9, 11, and so on:

```
# Import cv2 module
import cv2

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# Convert image color space BGR to RGB
%matplotlib inline

# read image
image = cv2.imread('tajmahal.jpg')

# Convert image color space BGR to RGB
rgb_image=cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Blurring the image using Median blurring
image_blur = cv2.medianBlur(rgb_image,11)

# Display the image
plt.imshow(image_blur)
```

This results in the following output:



In the preceding code block, we blurred the image using median blurring. Here, we used the `medianBlur()` method for median blurring and we can observe the blurred image in the output. In this section, we discussed average blurring, Gaussian blurring, and median blurring techniques. In the next section, we will learn how to detect human faces in images.

Face detection

Nowadays, everyone is using Facebook and you all must have seen facial recognition in an image on Facebook. Facial recognition identifies who a face belongs to and face detection only finds faces in an image, that is, face detection does not determine to whom the detected face belongs. Face detection in a given input image is quite a popular functionality in lots of applications; for example, counting the number of people in an image. In face detection, the algorithm tries to find human faces in a digital image.

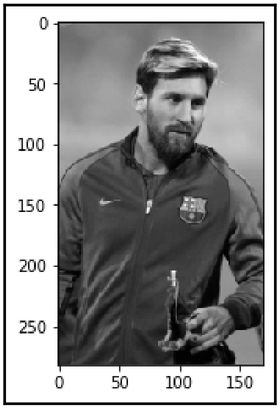
Face detection is a kind of classification problem. We can classify images into two classes, face or not face. We need lots of images to train such a model for classification. Thankfully, OpenCV offers pre-trained models such as the Haar Feature-Based Cascade Classifier and the **Local Binary Pattern** (**LBP**) classifier, trained on thousands of images. In our example, we will use Haar feature extraction to detect a face. Let's see how to capture a face in an image using OpenCV:

1. Read the image and convert it into grayscale:

```
# Import cv2 latest version of OpenCV library
import cv2
```

```
# Import numeric python (NumPy) library
import numpy as np

# Import matplotlib for showing the image
import matplotlib.pyplot as plt

# magic function to render the figure in a notebook
%matplotlib inline

# Read image
image= cv2.imread('messi.png')

# Convert image color space BGR to grayscale
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Displaying the grayscale image
plt.imshow(image_gray, cmap='gray')
```

This results in the following output:



In the preceding code example, we read the Lionel Messi image and converted it into a grayscale image using the `cvtColor()` function.

Let's find the faces in the generated gray image:

2. Load the Haar cascade face classifier file:

```
# Load the haar cascade face classifier file
haar_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
```

3. Get coordinates for all the faces in the image:

```
# Get the faces coordinates for all the faces in the image
faces_cordinates = haar_cascade.detectMultiScale(image_gray,
scaleFactor = 1.3, minNeighbors = 7);
```

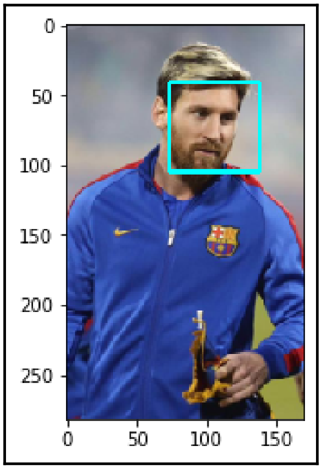
4. Draw a rectangle on detected faces:

```
# Draw rectangle on detected faces
for (p,q,r,s) in faces_cordinates:
    cv2.rectangle(image, (p, q), (p+r, q+s), (255,255,0), 2)
```

5. Convert image color space BGR to RGB and display the image:

```
# Convert image color space BGR to RGB
image_rgb=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display face detected image
plt.imshow(image_rgb)
```

This results in the following output:

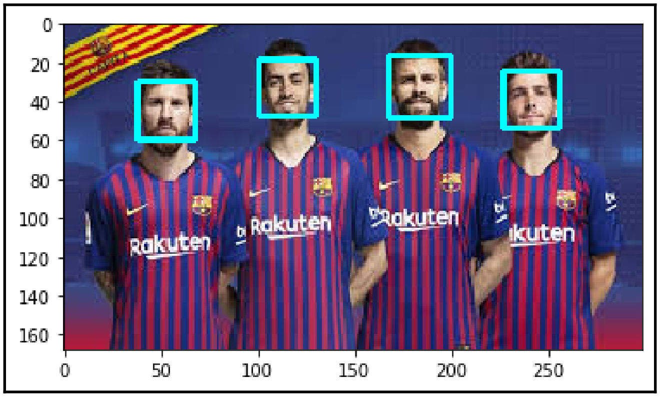


In the preceding example, we converted the BGR image to a grayscale image. OpenCV has pre-trained classifiers for face, eye, and smile detection. We can use a pre-trained face cascade classifier XML file (`haarcascade_frontalface_default.xml`). You can get the classifier file (`haarcascade_frontalface_default.xml`) from the official Git repo: `https://github.com/opencv/opencv/tree/master/data/haarcascades` or you can get it from our GitHub repo: `https://github.com/PacktPublishing/Python-Data-Analysis-Third-Edition/tree/master/Chapter13`.

After this, we can pass the image to the cascade classifier and get the face coordinates in the image. We have drawn rectangles on these face coordinates using the `rectangle()` function. Before displaying the output, we need to convert the RGB image to BGR to display it properly. Let's try this example on an image with multiple faces:

```
# Read the image
image= cv2.imread('barcelona.jpeg')

# Convert image BGR to grayscale
image_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Load the haar cascade face classifier file
haar_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# Get the faces coordinates for all the faces in the image
faces_cordinates = haar_cascade.detectMultiScale(image_gray, scaleFactor =
1.3, minNeighbors = 5);

# Draw rectangle on detected faces
for (x1,y1,x2,y2) in faces_cordinates:
cv2.rectangle(image, (x1, y1), (x1+x2, y1+y2), (255,255,0), 2)

# Convert image color space BGR to RGB
image_rgb=cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Display face detected the image
plt.imshow(image_rgb)
```

This results in the following output:



In the preceding example, we can see the program has detected all the faces in the image.

Summary

In this chapter, we discussed image processing using OpenCV. The main focus of the chapter was on basic image processing operations and face detection. The chapter started with an introduction to types of images and image color models. In later sections, the focus was on image operations such as drawing, resizing, flipping, and blurring an image. In the last section, we discussed face detection in a given input image

The next chapter, Chapter 14, *Parallel Computing Using Dask*, will focus on parallel computation on basic data science Python libraries such as Pandas, NumPy, and scikit-learn using Dask. The chapter will start with Dask data types such as dataframes, arrays, and bags. In later sections, we'll shift focus from dataFrames and arrays to delayed, preprocessing, and machine learning algorithms in parallel using Dask.

14
Parallel Computing Using Dask

Dask is one of the simplest ways to process your data in a parallel manner. The platform is for pandas lovers who struggle with large datasets. Dask offers scalability in a similar manner to Hadoop and Spark and the same flexibility that Airflow and Luigi provide. Dask can be used to work on pandas DataFrames and Numpy arrays that cannot fit into RAM. It splits these data structures and processes them in parallel while making minimal code changes. It utilizes your laptop power and has the ability to run locally. We can also deploy it on large distributed systems as we deploy Python applications. Dask can execute data in parallel and processes it in less time. It also scales the computation power of your workstation without migrating to a larger or distributed environment.

The main objective of this chapter is to learn how to perform flexible parallel computation on large datasets using Dask. The platform provides three data types for parallel execution: Dask Arrays, Dask DataFrames, and Dask Bags. The Dask array is like a NumPy array, while Dask DataFrames are like pandas DataFrames. Both can execute data in parallel. A Dask Bag is a wrapper for Python objects so that they can perform operations simultaneously. Another concept we'll cover in this chapter is Dask Delayed, which parallelizes code. Dask also offers data preprocessing and machine learning model development in parallel mode.

In this chapter, we will cover the following topics:

- Parallel computing using Dask
- Dask data types
- Dask Delayed
- Preprocessing data at scale
- Machine learning at scale

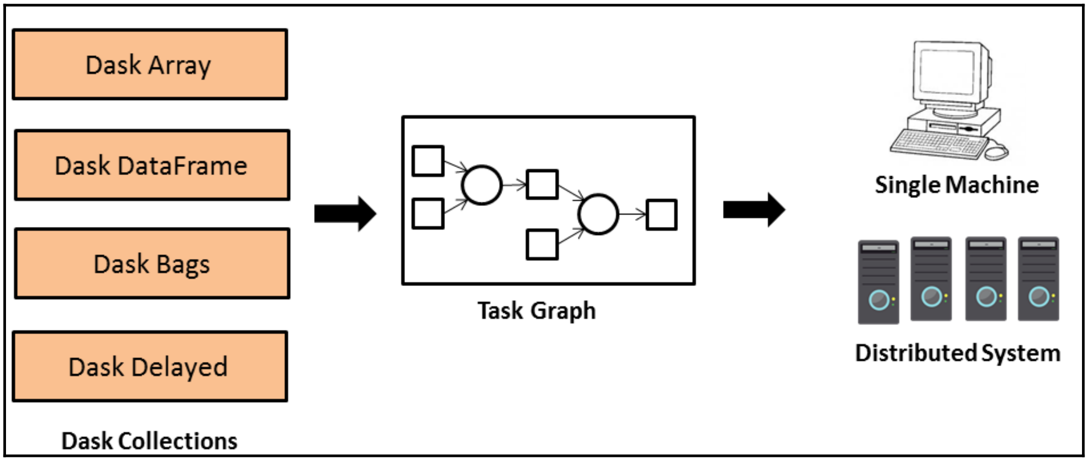
Let's get started!

Parallel computing using Dask

Python is one of the most popular programming languages among data professionals. Python data science libraries such as Numpy, Pandas, Scipy, and Scikit-learn can sequentially perform data science tasks. However, with large datasets, these libraries will become very slow due to not being scalable beyond a single machine. This is where Dask comes into the picture. Dask helps data professionals handle datasets that are larger than the RAM size on a single machine. Dask utilizes the multiple cores of a processor or uses it as a distributed computed environment. Dask has the following qualities:

- It is familiar with existing Python libraries
- It offers flexible task scheduling
- It offers a single and distributed environment for parallel computation
- It performs fast operations with lower latency and overhead
- It can scale up and scale down

Dask offers similar concepts to pandas, NumPy, and Scikit-learn, which makes it easier to learn. It is an open source parallel computing Python library that runs on top of pandas, Numpy, and Scikit-learn across multiple cores of a CPU or multiple systems. For example, if a laptop has a quad-core processor, then Dask will use 4 cores for processing the data. If the data won't fit in the RAM, it will be partitioned into chunks before processing. Dask scales up the pandas and NumPy capacity to deal with moderately large datasets. Let's understand how Dask perform operations in parallel by looking at the following diagram:



Dask creates a task graph to execute a program in parallel mode. In the task graph, nodes represent the task, and the edges between the nodes represent the dependency of one task over another.

Let's install the Dask library on our local system. By default, Anaconda has Dask installed already, but if you want to reinstall or update Dask, you can use the following command:

```
conda install dask
```

We can also install it using the `pip` command, as shown here:

```
pip install dask
```

With that, we have learned how to install the `dask` library for parallel and fast execution. Now, let's look at the core data types of the Dask library.

Dask data types

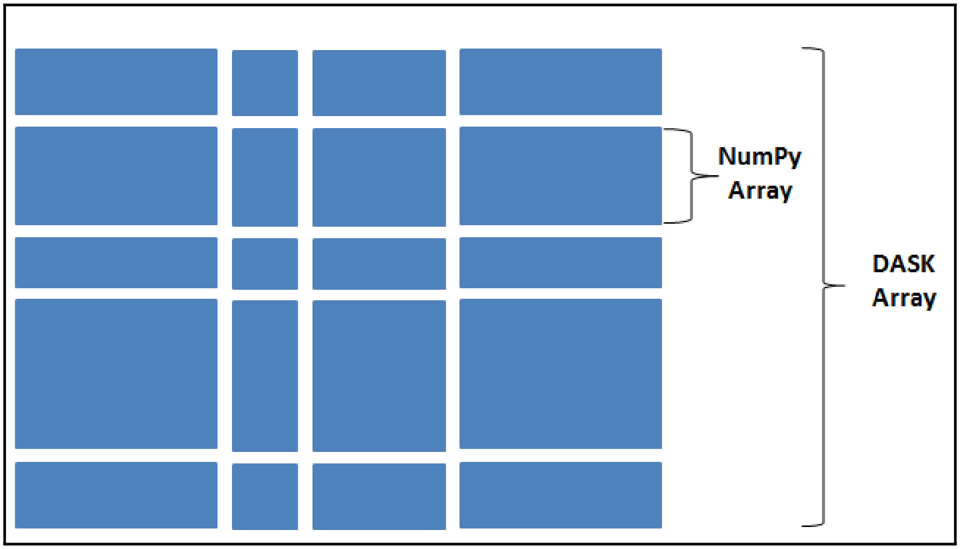
In computer programming, data types are basic building blocks for writing any kind of functionality. They help us work with different types of variables. Data types are the kind of values that are stored in variables. They can be primary and secondary.

Primary data types are the basic data types such as int, float, and char, while secondary data types are developed using primary data types such as lists, arrays, strings, and DataFrames. Dask offers three data structures for parallel operations: DataFrames, Bags, and Arrays. These data structures split data into multiple partitions and distribute them to multiple nodes in the cluster. A Dask DataFrame is a combination of multiple small pandas DataFrames and it operates in a similar manner. Dask Arrays are like NumPy arrays and support all the operations of Numpy. Finally, Dask Bags are used to process large Python objects.

Now, it's time to explore these data types. We'll start with Dask Arrays.

Dask Arrays

A Dask Array is an abstraction of the NumPy n-dimensional array, processed in parallel and partitioned into multiple sub-arrays. These small arrays can be on local or distributed remote machines. Dask Arrays can compute large-sized arraysby utilizing all the available cores in the system. They can be applied to statistics, optimization, bioinformatics, business domains, environmental science, and many more fields. They also support lots of NumPy operations, such as arithmetic and scalar operations, aggregation operations, matrices, and linear algebra operations. However, they do not support unknown shapes. Also, the `tolist` and `sort` operations are difficult to perform in parallel. Let's understand how Dask Arrays decompose data into a NumPy array and execute them in parallel by taking a look at the following diagram:



As we can see, there are multiple blocks of different shapes, all of which represent NumPy arrays. These arrays form a Dask Array and can be executed on multiple machines. Let's create an array using Dask:

```
# import Dask Array
import dask.array as da

# Create Dask Array using arange() function and generate values from 0 to
17
a = da.arange(18, chunks=4)

# Compute the array
a.compute()
```

This results in the following output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,17])
```

In the preceding example, we used the `compute()` function to get the final output. The `da.arange()` function will only create the computational graph, while the `compute()` function is used to execute that graph. We have generated 18 values with a chunk size of 4 using the `da.arange()` function. Let's also check the chunks in each partition:

```
# Check the chunk size
a.chunks
```

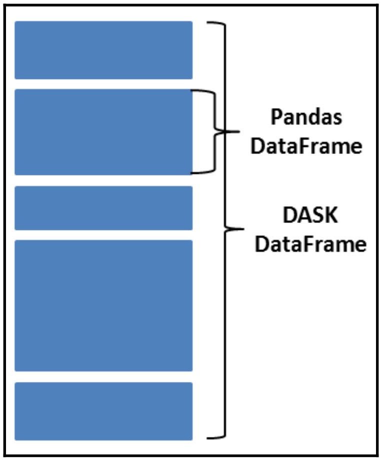
This results in the following output:

```
((4, 4, 4, 4, 2),)
```

In the preceding example, an array with 18 values was partitioned into five parts with a chunk size of 4, where these initial chunks have 4 values each and the last one has 2 values.

Dask DataFrames

Dask DataFrames are abstractions of pandas DataFrames. They are processed in parallel and partitioned into multiple smaller pandas DataFrames, as shown in the following diagram:



These small DataFrames can be stored on local or distributed remote machines. Dask DataFrames can compute large-sized DataFrames by utilizing all the available cores in the system. They coordinate the DataFrames using indexing and support standard pandas operations such as `groupby`, `join`, and `time series`. Dask DataFrames perform operations such as element-wise, row-wise, `isin()`, and date faster compared to `set_index()` and `join()` on index operations. Now, let's experiment with the performance or execution speed of Dask:

```
# Read csv file using pandas
import pandas as pd
%time temp = pd.read_csv("HR_comma_sep.csv")
```

This results in the following output:

```
CPU times: user 17.1 ms, sys: 8.34 ms, total: 25.4 ms

Wall time: 36.3 ms
```

In the preceding code, we tested the read time of a file using the pandas `read_csv()` function. Now, let's test the read time for the Dask `read_csv()` function:

```
# Read csv file using Dask

import dask.dataframe as dd

%time df = dd.read_csv("HR_comma_sep.csv")
```

This results in the following output:

```
CPU times: user 18.8 ms, sys: 5.08 ms, total: 23.9 ms

Wall time: 25.8 ms
```

In both examples, we can observe that the execution time for data reading is reduced when using the Dask `read_csv()` function.

DataFrame Indexing

Dask DataFrames support two types of index: label-based and positional indexing. The main problem with Dask Indexing is that it does not maintain the partition's information. This means it is difficult to perform row indexing; only column indexing is possible. `DataFrame.iloc` only supports integer-based indexing, while `DataFrame.loc` supports label-based indexing. `DataFrame.iloc` only selects columns.

Let's perform these indexing operations on a Dask DataFrame:

1. First, we must create a DataFrame and perform column indexing:

   ```
   # Import Dask and Pandas DataFrame
   import dask.dataframe as dd
   import pandas as pd

   # Create Pandas DataFrame
   df = pd.DataFrame({"P": [10, 20, 30], "Q": [40, 50, 60]},
   index=['p', 'q', 'r'])

   # Create Dask DataFrame
   ddf = dd.from_pandas(df, npartitions=2)

   # Check top records
   ddf.head()
   ```

   This results in the following output:

   ```
   P Q

   p 10 40

   q 20 50

   r 30 60
   ```

   In the preceding example, we created a pandas DataFrame (with `p`, `q`, and `r` indexes and `P` and `Q` columns) and converted it into a Dask DataFrame.

2. The column selection process in Dask is similar to what we do in pandas. Let's select a single column in our Dask DataFrame:

   ```
   # Single Column Selection
   ddf['P']
   ```

   This results in the following output:

   ```
   Dask Series Structure:

   npartitions=1

   p int64

   r ...
   Name: P, dtype: int64
   Dask Name: getitem, 2 tasks
   ```

In the preceding code, we selected a single column by passing the name of the column. For multiple column selection, we need to pass a list of columns.

3. Let's select multiple columns in our Dask DataFrame:

```
# Multiple Column Selection
ddf[['Q', 'P']]
```

This results in the following output:

```
Dask DataFrame Structure:

Q P

npartitions=1

p int64 int64

r ... ...

Dask Name: getitem, 2 tasks
```

Here, we have selected two columns from the list of columns available.

4. Now, let's create a DataFrame with an integer index:

```
# Import Dask and Pandas DataFrame
import dask.dataframe as dd
import pandas as pd

# Create Pandas DataFrame
df = pd.DataFrame({"X": [11, 12, 13], "Y": [41, 51, 61]})

# Create Dask DataFrame
ddf = dd.from_pandas(df, npartitions=2)

# Check top records
ddf.head()
```

This results in the following output:

```
X Y

0 11 41

1 12 51

2 13 61
```

In the preceding code, we created a pandas DataFrame and converted it into a Dask DataFrame using the `from_pandas()` function.

5. Let's select the required column using a positional integer index:

```
ddf.iloc[:, [1, 0]].compute()
```

This results in the following output:

```
Y X
0 41 11

1 51 12

2 61 13
```

In the preceding code, we swapped the column's location using `iloc` while using a positional integer index.

6. If we try to select all the rows, we will get a `NotImplementedError`, as shown here:

```
ddf.iloc[0:4, [1, 0]].compute()
```

This results in the following output:

```
NotImplementedError: 'DataFrame.iloc' only supports selecting
columns. It must be used like 'df.iloc[:, column_indexer]'.
```

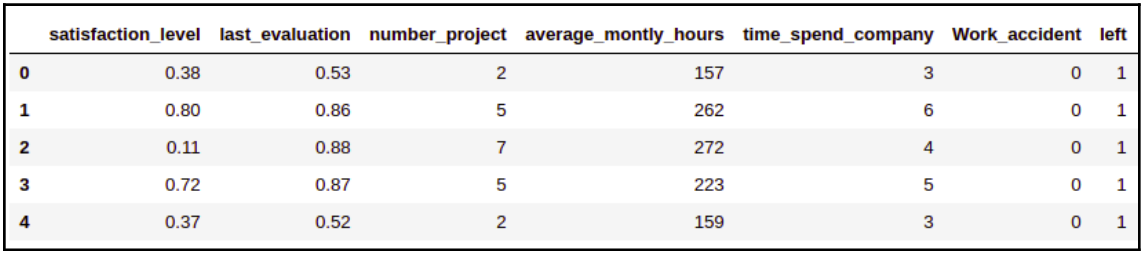
In the preceding code block, we can see that the `DataFrame.iloc` only supports selecting columns.

Filter data

We can filter the data from a Dask DataFrame similar to how we would do this for a pandas DataFrame. Let's take a look at the following example:

```
# Import Dask DataFrame
import dask.dataframe as dd

# Read CSV file
ddf = dd.read_csv('HR_comma_sep.csv')

# See top 5 records
ddf.head(5)
```

This results in the following output:

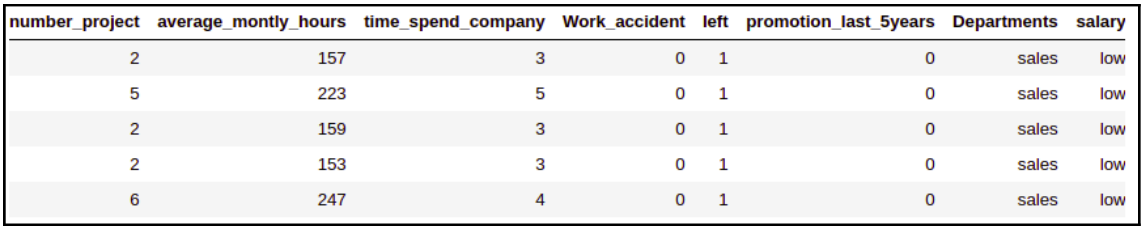


	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left
0	0.38	0.53	2	157	3	0	1
1	0.80	0.86	5	262	6	0	1
2	0.11	0.88	7	272	4	0	1
3	0.72	0.87	5	223	5	0	1
4	0.37	0.52	2	159	3	0	1

In the preceding code, we read the human resource CSV file using the `read_csv()` function into the Dask DataFrame. This output is only showing some of the columns. However, when you run the notebook for yourself, you will be able to see all the available columns. Let's filter the low-salary employees in the dataset:

```
# Filter employee with low salary
ddf2 = ddf[ddf.salary == 'low']

ddf2.compute().head()
```

This results in the following output:



number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments	salary
2	157	3	0	1	0	sales	low
5	223	5	0	1	0	sales	low
2	159	3	0	1	0	sales	low
2	153	3	0	1	0	sales	low
6	247	4	0	1	0	sales	low

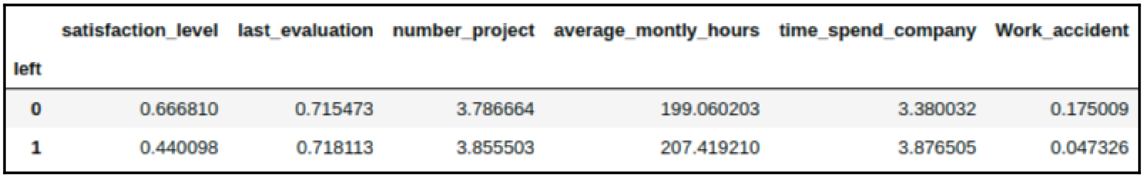
In the preceding code, we filtered the low-salary employees through the condition into the brackets.

Groupby

The `groupby` operation is used to aggregate similar items. First, it splits the data based on the values, finds an aggregate of similar values, and combines the aggregated results. This can be seen in the following code:

```
# Find the average values of all the columns for employee left or stayed
ddf.groupby('left').mean().compute()
```

This results in the following output:



	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident
left						
0	0.666810	0.715473	3.786664	199.060203	3.380032	0.175009
1	0.440098	0.718113	3.855503	207.419210	3.876505	0.047326

In the preceding example, we grouped the data based on the left column (it shows an employee who stayed or left the company) and aggregated it by the mean value.

Converting a pandas DataFrame into a Dask DataFrame

Dask DataFrames are implemented based on pandas DataFrames. For data analysts, it is necessary to learn how to convert a Dask DataFrame into a pandas DataFrame. Take a look at the following code:

```
# Import Dask DataFrame
from dask import dataframe as dd

# Convert pandas dataframe to dask dataframe
ddf = dd.from_pandas(pd_df,chunksize=4)

type(ddf)
```

This results in the following output:

```
dask.dataframe.core.DataFrame
```

Here, we have used the `from_pandas()` method to convert a pandas DataFrame into a Dask DataFrame.

Converting a Dask DataFrame into a pandas DataFrame

In the previous subsection, we converted a pandas DataFrame into a Dask DataFrame. Similarly, we can convert a Dask DataFrame into a pandas DataFrame using the `compute()` method, as shown here:

```
# Convert dask DataFrame to pandas DataFrame
pd_df = df.compute()

type(pd_df)
```

This results in the following output:

```
pandas.core.frame.DataFrame
```

Now, let's learn about another important topic: Dask Bags.

Dask Bags

A Dask Bag is an abstraction over generic Python objects. It performs `map`, `filter`, `fold`, and `groupby` operations in the parallel interface of smaller Python objects using a Python iterator. This execution is similar to PyToolz or the PySpark RDD. Dask Bags are more suitable for unstructured and semi-structured datasets such as text, JSON, and log files. They perform multiprocessing for computation for faster processing but will not perform well with inter-worker communication. Bags are immutable types of structures that cannot be changed and are slower compared to Dask Arrays and DataFrames. Bags also perform slowly on the `groupby` operation, so it is recommended that you use `foldby` instead of `groupby`.

Now, let's create various Dask Bag objects and perform operations on them.

Creating a Dask Bag using Python iterable items

Let's create some Dask Bag objects using Python iterable items:

```
# Import dask bag
import dask.bag as db

# Create a bag of list items
items_bag = db.from_sequence([1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
npartitions=3)

# Take initial two items
items_bag.take(2)
```

This results in the following output:

```
(1, 2)
```

In the preceding code, we created a bag of list items using the `from_sequence()` method. The `from_Sequence()` method takes a list and places it into `npartitions` (a number of partitions). Let's filter odd numbers from the list:

```
# Filter the bag of list items
```

```
items_square=items_bag.filter(lambda x: x if x % 2 != 0 else None)

# Compute the results
items_square.compute()
```

This results in the following output:

```
[1, 3, 5, 7, 9]
```

In the preceding code, we filtered the odd numbers from the bag of lists using the `filter()` method. Now, let's square each item of the bag using the `map` function:

```
# Square the bag of list items
items_square=items_b.map(lambda x: x**2)

# Compute the results
items_square.compute()
```

This results in the following output:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

In the preceding code, we used the `map()` function to map the bag items. We mapped these items to their square value.

Creating a Dask Bag using a text file

We can create a Dask Bag using a text file by using the `read_text()` method, as follows:

```
# Import dask bag
import dask.bag as db

# Create a bag of text file
text = db.read_text('sample.txt')

# Show initial 2 items from text
text.take(2)
```

This results in the following output:

```
('Hi! how are you? \n', '\n')
```

In the preceding code, we read a text file into a `dask bag` object by using the `read_text()` method. This allowed us to show the two initial items in the Dask Bag.

Storing a Dask Bag in a text file

Let's store a Dask Bag in a text file:

```
# Convert dask bag object into text file
text.to_textfiles('/path/to/data/*.text.gz')
```

This results in the following output:

```
['/path/to/data/0.text.gz']
```

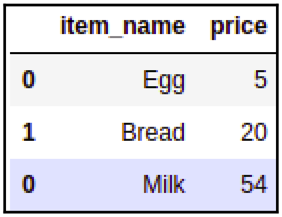
In the preceding code, `to_textfiles()` converted the `bag` object into a text file.

Storing a Dask Bag in a DataFrame

Let's store a Dask Bag in a DataFrame:

```
# Import dask bag
import dask.bag as db

# Create a bag of dictionary items
dict_bag = db.from_sequence([{'item_name': 'Egg', 'price': 5},
{'item_name': 'Bread', 'price': 20},
{'item_name': 'Milk', 'price': 54}],
npartitions=2)

# Convert bag object into dataframe
df = dict_bag.to_dataframe()

# Execute the graph results
df.compute()
```

This results in the following output:



	item_name	price
0	Egg	5
1	Bread	20
0	Milk	54

In the preceding example, we created a Dask Bag of dictionary items and converted it into a Dask DataFrame using the `to_dataframe()` method. In the next section, we'll look at Dask Delayed.

Dask Delayed

Dask Delayed is an approach we can use to parallelize code. It can delay the dependent function calls in task graphs and provides complete user control over parallel processes while improving performance. Its lazy computation helps us control the execution of functions. However, this differs from the execution timings of functions for parallel execution.

Let's understand the concept of Dask Delayed by looking at an example:

```
# Import dask delayed and compute
from dask import delayed, compute

# Create delayed function
@delayed
def cube(item):
    return item ** 3

# Create delayed function
@delayed
def average(items):
    return sum(items)/len(items)

# create a list
item_list = [2, 3, 4]

# Compute cube of given item list
cube_list= [cube(i) for i in item_list]

# Compute average of cube_list
computation_graph = average(cube_list)

# Compute the results
computation_graph.compute()
```

This results in the following output:

```
33.0
```

In the preceding example, two methods, `cube` and `average`, were annotated with `@dask.delayed`. A list of three numbers was created and a cube containing every value was computed. After computing the cube of list values, we calculated the average of all the values. All these operations are lazy in nature and are computed later when the output is expected from the programmer and the flow of execution is stored in a computational graph. We executed this using the `compute()` method. Here, all the cube operations will execute in a parallel fashion.

Now, we will visualize the computational graph. However, before we can do this, we need to install the Graphviz editor.

On Windows, we can install Graphviz using `pip`. We must also set the path in an environment variable:

```
 pip install graphviz
```

On Mac, we can install it using `brew`, as follows:

```
brew install graphviz
```

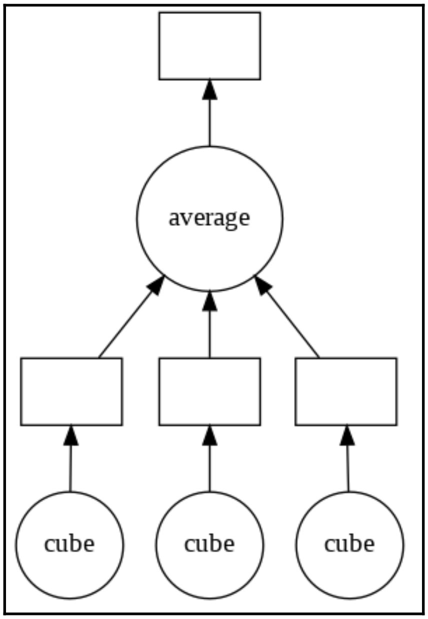
On Ubuntu, we need to install it on a Terminal using the `sudo apt-get` command:

```
sudo apt-get install graphviz
```

Now, let's visualize the computational graph:

```
# Compute the results
computation_graph.visualize()
```

This results in the following output:



In the preceding example, we printed a computational graph using the `visualize()` method. In this graph, all the cube operations were executed in a parallel fashion and their result was consumed by the `average()` function.

Preprocessing data at scale

Dask preprocessing offers scikit-learn functionalities such as scalers, encoders, and train/test splits. These preprocessing functionalities work well with Dask DataFrames and Arrays since they can fit and transform data in parallel. In this section, we will discuss feature scaling and feature encoding.

Feature scaling in Dask

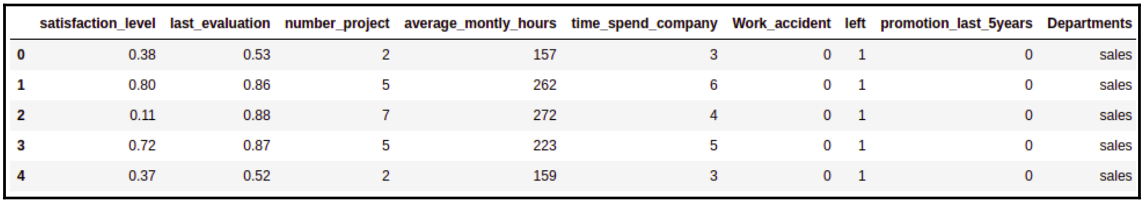
As we discussed in Chapter 7, *Cleaning Messy Data*, feature scaling, also known as feature normalization, is used to scale the features at the same level. It can handle issues regarding different column ranges and units. Dask also offers scaling methods that have parallel execution capacity. It uses most of the methods that scikit-learn offers:

Scaler	**Description**
MinMaxScaler	Transforms features by scaling each feature to a given range
RobustScaler	Scales features using statistics that are robust to outliers
StandardScaler	Standardizes features by removing the mean and scaling them to unit variance

Let's scale the `last_evaluation` (employee performance score) column of the human resource dataset:

```
# Import Dask DataFrame
import dask.dataframe as dd

# Read CSV file
ddf = dd.read_csv('HR_comma_sep.csv')

# See top 5 records
ddf.head(5)
```

This results in the following output:



	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments
0	0.38	0.53	2	157	3	0	1	0	sales
1	0.80	0.86	5	262	6	0	1	0	sales
2	0.11	0.88	7	272	4	0	1	0	sales
3	0.72	0.87	5	223	5	0	1	0	sales
4	0.37	0.52	2	159	3	0	1	0	sales

In the preceding code, we read the human resource CSV file using the `read_csv()` function into a Dask DataFrame. The preceding output only shows some of the columns that are available. However, when you run the notebook for yourself, you'll be able to see all the columns in the dataset. Now, let's scale the `last_evalaution` column (last evaluated performance score):

```
# Import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

# Instantiate the MinMaxScaler Object
scaler = MinMaxScaler(feature_range=(0, 100))

# Fit the data on Scaler
scaler.fit(ddf[['last_evaluation']])

# Transform the data
performance_score=scaler.transform(ddf[['last_evaluation']])

# Let's see the scaled performance score
performance_score
```

This results in the following output:

```
array([[26.5625],

[78.125 ],

[81.25 ],

...,

[26.5625],

[93.75 ],

[25. ]])
```

In the preceding example, we scaled the `last_evaluation` (last evaluated performance score) column. We scaled it from a range of 0-1 range to a range of 0-100. Next, we will look at feature encoding in Dask.

Feature encoding in Dask

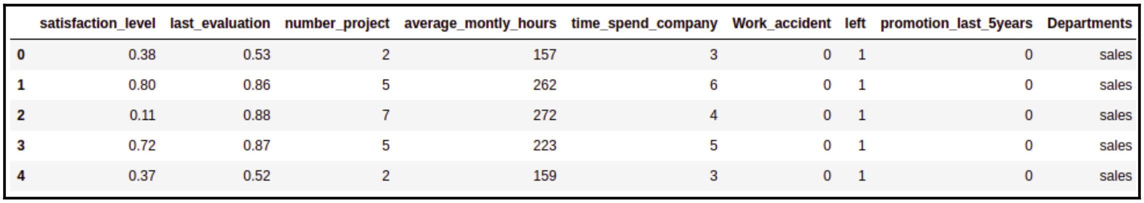
As we discussed in Chapter 7, *Cleaning Messy Data*, feature encoding is a very useful technique for handling categorical features. Dask also offers encoding methods that have parallel execution capacity. It uses most of the methods that scikit-learn offers:

Encoder	**Description**
LabelEncoder	Encodes labels with a value between 0 and 1 that's less than the number of classes available.
OneHotEncoder	Encodes categorical integer features as a one-hot encoding.
OrdinalEncoder	Encodes a categorical column as an ordinal variable.

Let's try using these methods:

```
# Import Dask DataFrame
import dask.dataframe as dd

# Read CSV file
ddf = dd.read_csv('HR_comma_sep.csv')

# See top 5 records
ddf.head(5)
```

This results in the following output:

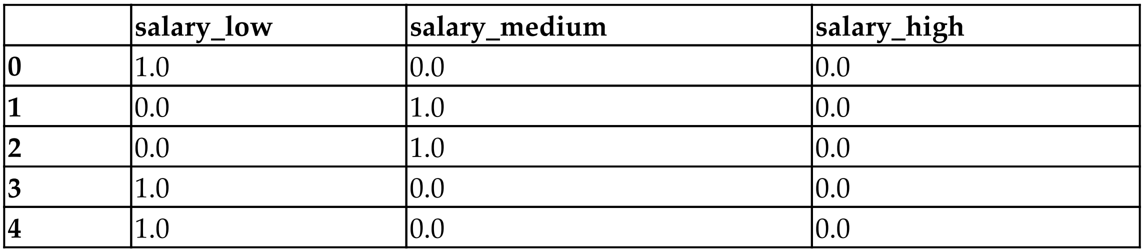


	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments
0	0.38	0.53	2	157	3	0	1	0	sales
1	0.80	0.86	5	262	6	0	1	0	sales
2	0.11	0.88	7	272	4	0	1	0	sales
3	0.72	0.87	5	223	5	0	1	0	sales
4	0.37	0.52	2	159	3	0	1	0	sales

In the preceding code, we read the human resource CSV file using the `read_csv()` function into a Dask DataFrame. The preceding output only shows some of the columns that are available. However, when you run the notebook for yourself, you'll be able to see all the columns in the dataset. Now, let's scale the `last_evalaution` column (last evaluated performance score):

```
# Import Onehot Encoder
from dask_ml.preprocessing import Categorizer
from dask_ml.preprocessing import OneHotEncoder
from sklearn.pipeline import make_pipeline

# Create pipeline with Categorizer and OneHotEncoder
pipe = make_pipeline(Categorizer(), OneHotEncoder())

# Fit and transform the Categorizer and OneHotEncoder
pipe.fit(ddf[['salary',]])
result=pipe.transform(ddf[['salary',]])

# See top 5 records
result.head()
```

This results in the following output:



	salary_low	salary_medium	salary_high
0	1.0	0.0	0.0
1	0.0	1.0	0.0
2	0.0	1.0	0.0
3	1.0	0.0	0.0
4	1.0	0.0	0.0

In the preceding example, the `scikit-learn` pipeline was created using `Categorizer()` and `OneHotEncoder()`. The Salary column of the Human Resource data was then encoded using the `fit()` and `transform()` methods. Note that the categorizer will convert the columns of a DataFrame into categorical data types.

Similarly, we can also encode the Salary column using the ordinal encoder. Let's take a look at an example:

```
# Import Onehot Encoder
from dask_ml.preprocessing import Categorizer
from dask_ml.preprocessing import OrdinalEncoder
from sklearn.pipeline import make_pipeline

# Create pipeline with Categorizer and OrdinalEncoder
```

```
pipe = make_pipeline(Categorizer(), OrdinalEncoder())

# Fit and transform the Categorizer and OneHotEncoder
pipe.fit(ddf[['salary',]])
result=pipe.transform(ddf[['salary',]])

# Let's see encoded results
result.head()
```

This results in the following output:

```
salary

0 0

1 1

2 1

3 0

4 0
```

In the preceding example, the scikit-learn pipeline was created using `Categorizer()` and `OrdinalEncoder()`. The Salary column of the Human Resource data was then encoded using the `fit()` and `transform()` methods. Note that the categorizer will convert the columns of a DataFrame in categorical data types.

Machine learning at scale

Dask offers Dask-ML services for large-scale machine learning operations using Python. Dask-ML decreases the model training time for medium-sized datasets and experiments with hyperparameter tuning. It offers scikit-learn-like machine learning algorithms for ML operations.

We can scale scikit-learn in three different ways: parallelize scikit-learn using `joblib` by using random forest and SVC; reimplement algorithms using Dask Arrays using generalized linear models, preprocessing, and clustering; and partner it with distributed libraries such as XGBoost and Tensorflow.

Let's start by looking at parallel computing using scikit-learn.

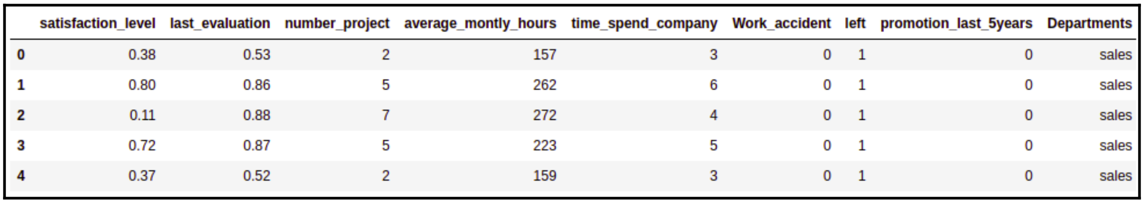
Parallel computing using scikit-learn

To perform parallel computing using scikit-learn on a single CPU, we need to use `joblib`. This makes scikit-learn operations parallel computable. The `joblib` library performs parallelization on Python jobs. Dask can help us perform parallel operations on multiple scikit-learn estimators. Let's take a look:

1. First, we need to read the dataset. We can load the dataset using a pandas DataFrame, like so:

```
# Import Dask DataFrame
import pandas as pd


# Read CSV file
df = pd.read_csv('HR_comma_sep.csv')


# See top 5 records
df.head(5)
```

This results in the following output:



	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments
0	0.38	0.53	2	157	3	0	1	0	sales
1	0.80	0.86	5	262	6	0	1	0	sales
2	0.11	0.88	7	272	4	0	1	0	sales
3	0.72	0.87	5	223	5	0	1	0	sales
4	0.37	0.52	2	159	3	0	1	0	sales

In the preceding code, we read the human resource CSV file using the `read_csv()` function into a Dask DataFrame. The preceding output only shows some of the columns that are available. However, when you run the notebook for yourself, you will be able to see all the columns in the dataset. Now, let's scale the `last_evalaution` column (last evaluated performance score).

2. Next, we must select the dependent and independent columns. To do this, select the columns and divide the data into dependent and independent variables, as follows:

```
# select the feature and target columns
data=df[['satisfaction_level', 'last_evaluation']]

label=df['left']
```

3. Create a scheduler and generate the model in parallel. Import the `dask.distributed` client to create a scheduler and worker on a local machine:

```
# Import client
from dask.distributed import Client

# Instantiate the Client
client = Client()
```

4. The next step is to create a parallel backend using `sklearn.externals.joblib` and write the normal `scikit-learn` code:

```
# import dask_ml.joblib
from sklearn.externals.joblib import parallel_backend

with parallel_backend('dask'):
    # Write normal scikit-learn code here
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.metrics import accuracy_score
    from sklearn.model_selection import train_test_split

    # Divide the data into two parts: training and testing set
    X_train, X_test, y_train, y_test = train_test_split(data,label,
    test_size=0.2,
    random_state=0)
    # Instantiate RandomForest Model
    model = RandomForestClassifier()

    # Fit the model
    model.fit(X_train,y_train)

    # Predict the classes
    y_pred = model.predict(X_test)

    # Find model accuracy
    print("Accuracy:",accuracy_score(y_test, y_pred))
```

This results in the following output:

```
Accuracy: 0.92
```

The preceding parallel generated random forest model has given us 92% accuracy, which is very good.

Reimplementing ML algorithms for Dask

Some machine learning algorithms have been reimplemented by the Dask development team using Dask Arrays and DataFrames. The following algorithms have been reimplemented:

- Linear machine learning models such as linear regression and logistic regression
- Preprocessing with scalers and encoders
- Unsupervised algorithms such as k-means clustering and spectral clustering

In the following subsection, we will build a logistic regression model and perform clustering on the dataset.

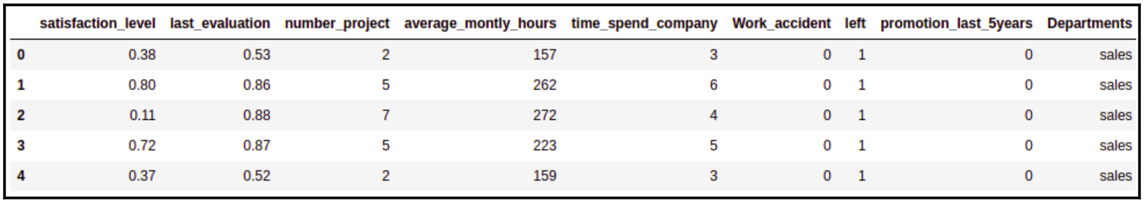
Logistic regression

Let's build a classifier using logistic regression:

1. Load the dataset into a Dask DataFrame, as follows:

```
# Read CSV file using Dask
import dask.dataframe as dd

# Read Human Resource Data
ddf = dd.read_csv("HR_comma_sep.csv")

# Let's see top 5 records
ddf.head()
```

This results in the following output:



	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments
0	0.38	0.53	2	157	3	0	1	0	sales
1	0.80	0.86	5	262	6	0	1	0	sales
2	0.11	0.88	7	272	4	0	1	0	sales
3	0.72	0.87	5	223	5	0	1	0	sales
4	0.37	0.52	2	159	3	0	1	0	sales

In the preceding code, we read the human resource CSV file using the `read_csv()` function into a Dask DataFrame. The preceding output only shows some of the columns that are available. However, you run the notebook for yourself, you will be able to see all the columns in the dataset. Now, let's scale the `last_evalaution` column (last evaluated performance score).

2. Next, select the required column for classification and divide it into dependent and independent variables:

   ```
   data=ddf[['satisfaction_level','last_evaluation']].to_dask_array(le
   ngths=True)

   label=ddf['left'].to_dask_array(lengths=True)
   ```

3. Now, let's create a `LogisticRegression` model. First, import `LogisticRegression` and `train_test_split`. Once you've imported the required libraries, divide the dataset into two parts; that is, training and testing datasets:

   ```
   # Import Dask based LogisticRegression
   from dask_ml.linear_model import LogisticRegression

   # Import Dask based train_test_split
   from dask_ml.model_selection import train_test_split

   # Split data into training and testing set
   X_train, X_test, y_train, y_test = train_test_split(data, label)
   ```

4. Instantiate the model and fit it to a training dataset. Now, you can predict the test data and compute the model's accuracy, as follows:

   ```
   # Create logistic regression model
   model = LogisticRegression()

   # Fit the model
   model.fit(X_train,y_train)

   # Predict the classes
   y_pred = model.predict(X_test)

   # Find model accuracy
   print("Accuracy:",accuracy_score(y_test, y_pred))
   ```

   This results in the following output:

   ```
   Accuracy: 0.7753333333333333
   ```

   As we can see, the model is offering an accuracy of 77.5%, which is considered good.

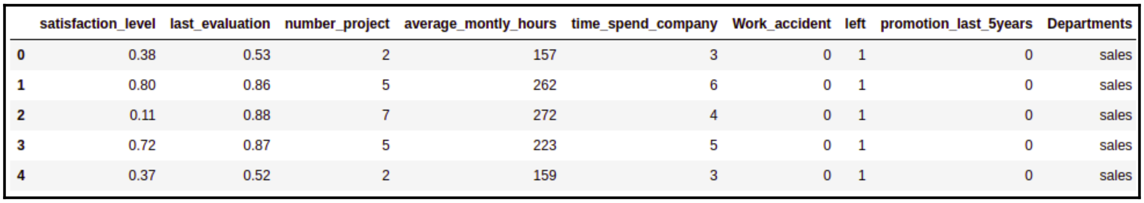
Clustering

The developers of Dask have also reimplemented various k-means clustering algorithms. Let's perform clustering using Dask:

1. Read the human resource data into a Dask DataFrame, as follows:

```
# Read CSV file using Dask
import dask.dataframe as dd

# Read Human Resource Data
ddf = dd.read_csv("HR_comma_sep.csv")

# Let's see top 5 records
ddf.head()
```

This results in the following output:



	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	Departments
0	0.38	0.53	2	157	3	0	1	0	sales
1	0.80	0.86	5	262	6	0	1	0	sales
2	0.11	0.88	7	272	4	0	1	0	sales
3	0.72	0.87	5	223	5	0	1	0	sales
4	0.37	0.52	2	159	3	0	1	0	sales

In the preceding code, we read the human resource CSV file using the `read_csv()` function into a Dask DataFrame. The preceding output only shows some of the columns that are available. However, when you run the notebook for yourself, you will be able to see all the columns in the dataset. Now, let's scale the `last_evalaution` column (last evaluated performance score).

2. Next, select the required column for k-means clustering. We have selected the `satisfaction_level` and `last_evaluation` columns here:

```
data=ddf[['satisfaction_level',
'last_evaluation']].to_dask_array(lengths=True)
```

3. Now, let's create a k-means clustering model. First, import k-means. Once you've imported the required libraries, fit them onto the dataset and get the necessary labels. We can find these labels by using the `compute()` method:

```
# Import Dask based Kmeans
from dask_ml.cluster import KMeans

# Create the Kmeans model
```

```
model=KMeans(n_clusters=3)

# Fit the model
model.fit(data)

# Predict the classes
label=model.labels_

# Compute the results
label.compute()
```

This results in the following output:

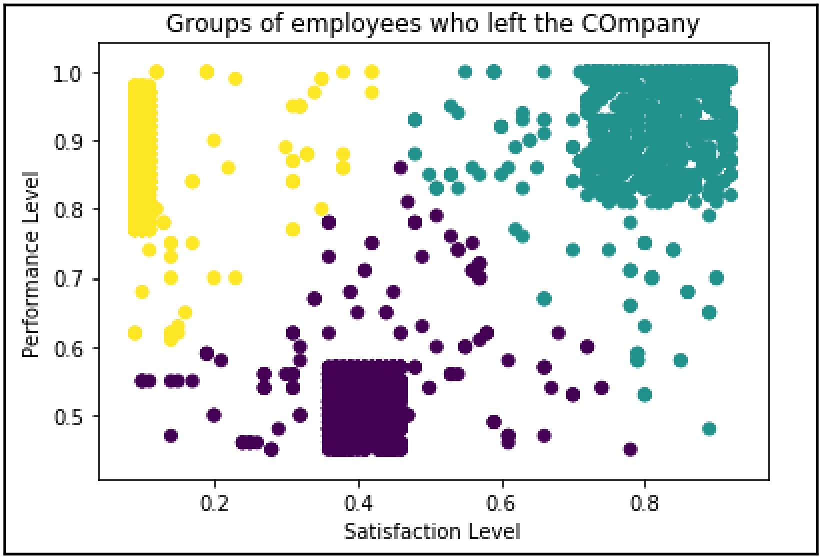
```
array([0, 1, 2, ..., 0, 2, 0], dtype=int32)
```

In the preceding code, we created the k-means model with three clusters, fitted the model, and predicted the labels for the cluster.

4. Now, we will visualize the k-means results using the `matplotlib` library:

```
# Import matplotlib.pyplot
import matplotlib.pyplot as plt

# Prepare x,y and cluster_labels
x=data[:,0].compute()
y=data[:,1].compute()
cluster_labels=label.compute()

# Draw scatter plot
plt.scatter(x,y, c=cluster_labels)

# Add label on X-axis
plt.xlabel('Satisfaction Level')

# Add label on X-axis
plt.ylabel('Performance Level')

# Add a title to the graph
plt.title('Groups of employees who left the Company')

# Show the plot
plt.show()
```

This results in the following output:



In the preceding code, we visualized the clusters using `matplotlib.pyplot`. Here, we have plotted the satisfaction score on the X-axis, the performance score on the Y-axis, and distinguished between the clusters by using different colors.

Summary

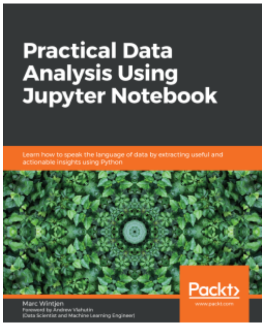
In this chapter, we focused on how to perform parallel computation on basic data science Python libraries such as pandas, Numpy, and scikit-learn. Dask provides a complete abstraction for DataFrames and Arrays for processing moderately large datasets over single/multiple core machines or multiple nodes in a cluster.

We started this chapter by looking at Dask data types such as DataFrames, Arrays, and Bags. After that, we focused on Dask Delayed, preprocessing, and machine learning algorithms in a parallel environment.

This was the last chapter of this book, which means our learning journey ends here. We have focused on core Python libraries for data analysis and machine learning such as pandas, Numpy, Scipy, and scikit-learn. We have also focused on Python libraries that can be used for text analytics, image analytics, and parallel computation such as NLTK, spaCy, OpenCV, and Dask. Of course, your learning process doesn't need to stop here; keep learning new things and about the latest changes. Try to explore and change code based on your business or client needs. You can also start private or personal projects for learning purposes. If you are unable to decide on what kind of project you want to start, you can participate in Kaggle competitions at `http://www.kaggle.com/` and more!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

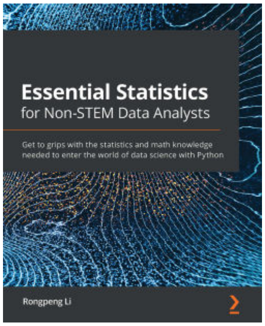

Practical Data Analysis Using Jupyter Notebook
Learn how to speak the language of data by extracting useful and actionable insights using Python
Packt>
www.packt.com
Marc Wintjen
Foreword by Andrew Vlahutin
(Data Scientist and Machine Learning Engineer)

Practical Data Analysis Using Jupyter Notebook

Marc Wintjen

ISBN: 978-1-83882-603-1

- Understand the importance of data literacy and how to communicate effectively using data
- Find out how to use Python packages such as NumPy, pandas, Matplotlib, and the Natural Language Toolkit (NLTK) for data analysis
- Wrangle data and create DataFrames using pandas
- Produce charts and data visualizations using time-series datasets
- Discover relationships and how to join data together using SQL
- Use NLP techniques to work with unstructured data to create sentiment analysis models
- Discover patterns in real-world datasets that provide accurate insights


Essential Statistics
for Non-STEM Data Analysts
Get to grips with the statistics and math knowledge needed to enter the world of data science with Python
Rongpeng Li

Essential Statistics for Non-STEM Data Analysts

Rongpeng Li

ISBN: 978-1-83898-484-7

- Find out how to grab and load data into an analysis environment
- Perform descriptive analysis to extract meaningful summaries from data
- Discover probability, parameter estimation, hypothesis tests, and experiment design best practices
- Get to grips with resampling and bootstrapping in Python
- Delve into statistical tests with variance analysis, time series analysis, and A/B test examples
- Understand the statistics behind popular machine learning algorithms
- Answer questions on statistics for data scientist interviews

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index