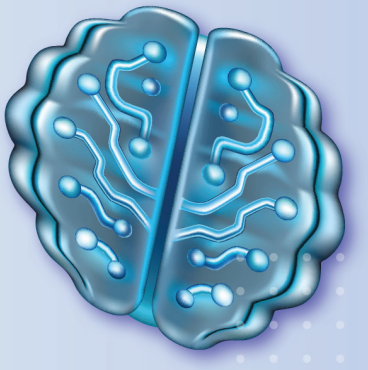
Hands-On AI Development with Python

Build and Deploy Real-World AI, Machine Learning, Deep Learning, and NLP Applications





Vivian Aranha

packt

Hands-On AI Development with Python

Build and Deploy Real-World AI, Machine Learning, Deep Learning, and NLP Applications

Vivian Aranha

‹packt›

Hands-On AI Development with Python

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
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 author, 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.

Portfolio Director: Sunith Shetty
Relationship Lead: Deepesh Patel
Project Manager: Yash Basil
Content Engineer: Mamta Wardhani
Technical Editor: Shweta Amale
Indexer: Manju Arasan
Production Designer: Shantanu Zagade
Growth Lead: Shruthi Shetty

First published: September 2026

Production reference: 1170926

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80808-853-7
www.packtpub.com

Contributors

About the author

Vivian Aranha is an AI educator, technology leader, and founder of School of AI, with over 20 years of industry experience. He earned a Bachelor's degree in Information Technology in 2004 and a Master's degree in Computer Science in 2006. His career spans web technologies, mobile app development for iOS and Android, blockchain solutions, and AI systems and applications.

Vivian has worked with Fortune 500 organizations, including The Washington Post, Delta Air Lines, and IBM. An instructor since 2009, he has trained professionals worldwide and now teaches AI globally. His courses have attracted over 2.5 million enrollments, with more than 500,000 students learning through School of AI, Udemy, Skool, and Maven.

About the reviewer

Bala Kannaiyan is a data engineering leader and pioneer in AI and cloud modernization, with over two decades of IT experience across data management, cloud architecture, and advanced analytics. As a senior leader at a US cloud technology giant, he drives Fortune 500 and Government organizations through large-scale modernization. A Bachelor of Engineering graduate of the prestigious Anna University, he holds numerous certifications in Snowflake, Azure, Databricks, Oracle, and Scrum, has authored BPB's Snowflake SnowPro Core Certification Guide, and has reviewed Packt's *Snowflake: The Ultimate Guide to Snowpark*. A mentor at regional user groups and a non-profit volunteer, he expands access to education. He enjoys traveling and gardening.

Table of Contents

Preface **xvii**

Free benefits with your book ..... **xxii**

Chapter 1: Python for AI and Data Science Foundations **1**

Technical requirements **2**
Introduction to Python and installation **3**
Installing Jupyter Notebook **6**
Writing your first program in Jupyter Notebook • 7
Python basics **7**
Variables and data types • 8
Operators • 8
Control flow • 9
If-else statements • 9
For loop • 10
While loop • 10
Functions • 11
Working with lists **12**
Working with dictionaries **13**
Introduction to NumPy **14**
Creating arrays using NumPy • 14
Array operations • 15
Matrix operations • 15
Introduction to pandas **16**
Creating a CSV file • 17
Loading and inspecting data with pandas • 18
Data manipulation in pandas • 19
Hands-on project: Basic data manipulation and file handling **21**
Introduction to exploratory data analysis **22**
Loading and inspecting data • 22
Inspecting data structures • 24

Handling missing data • 25
Identifying missing data • 26
Techniques to handle missing data • 27
Data transformation and feature engineering 29
Feature scaling • 29
Categorical data encoding • 30
Visualizing data with matplotlib and seaborn 30
Matplotlib basics • 30
Advanced data visualization with seaborn • 34
Descriptive statistics • 35
Hands-on project: Exploratory data analysis of a real dataset 36
Summary 39

Chapter 2: Machine Learning and Classification Models 41

Technical requirements 42
Machine learning and its types 43
Supervised learning and dataset preparation 43
Splitting data into training and test sets • 45
Building a linear regression model • 45
Making predictions • 46
Evaluating model performance • 46
Feature scaling and regularization 47
Hands-on project: Predicting house prices using linear regression 48
Importing libraries • 49
Loading and inspecting the California housing dataset • 50
Perform exploratory data analysis • 51
Preparing features and target variables • 53
Creating the train-test split • 53
Training the linear regression model • 54
Making predictions and evaluating the model • 54
Visualizing actual vs. predicted values • 55
Displaying model coefficients • 56
Understanding classification models in ML 57
What is classification? • 57
Logistic regression for classification • 57

Building a logistic regression classifier **58**
Evaluating the classification model • 58
Confusion matrix • 59
Accuracy • 59
Precision and recall • 59
F1 score • 60
Visualizing the decision boundary • 60
Hands-on project: Spam detection using logistic regression **62**
Importing libraries • 62
Loading and inspecting data • 63
Preparing the features and target • 63
Creating the train-test split • 63
Training the logistic regression model • 64
Making predictions • 64
Evaluating the model • 64
Visualizing the confusion matrix • 65
Summary **66**

Chapter 3: Neural Networks and Natural Language Processing **67**

Technical requirements **68**
What is a neural network? **69**
Introduction to deep learning frameworks • 70
MNIST data overview **70**
Building a simple neural network for MNIST classification **71**
Step 1: Data preprocessing • 72
Step 2: Defining the neural network architecture • 73
Step 3: Compiling the neural network model • 74
Step 4: Training the neural network • 74
Evaluating the neural network **75**
Model evaluation • 75
Making predictions • 75
Understanding activation functions: ReLU and softmax **75**
Hands-on project: Handwritten digit classification using neural networks **78**
Introduction to natural language processing **80**
Sentiment analysis: understanding text classification **81**

Text preprocessing • 81

Tokenization • 81

Stemming and lemmatization • 81

Vectorization • 82

Using pre-trained models for NLP 83

Building a sentiment analysis model with TensorFlow 85

Model evaluation and confusion matrix 87

Hands-on project: Sentiment analysis using pre-trained NLP models 88

Sentiment analysis with TensorFlow and the IMDB dataset • 89

Summary 90

Chapter 4: Deploying an AI Model as a Web Service 91

Technical requirements 92

Installing Flask 93

Creating a basic Flask application 93

Creating a web interface for the model 95

Building a prediction function • 96

Creating a Flask web form for input • 98

Creating the prediction route in Flask • 99

Loading the model and vectorizer • 100

Implementing the prediction logic • 100

Testing the prediction route • 101

Deploying the Flask application to Heroku 102

Setting up Heroku • 102

Getting started with the Heroku CLI • 102

Preparing the application for deployment • 102

Deploying to Heroku • 103

Testing the web service and API • 105

Summary 107

Chapter 5: Supervised Learning and Forecasting Projects 109

Technical requirements 109

Beginner projects 110

Implementing a basic calculator in Python • 110

Defining arithmetic functions • 110

Creating the calculator function • 111
Running the calculator • 112
Building a simple chatbot using predefined responses • 113
Defining the response dictionary • 114
Matching user input to responses • 114
Running the chatbot • 115
Building a spam email detector using scikit-learn • 117
Downloading and preparing the dataset • 117
Importing libraries for spam detection • 117
Loading and splitting the dataset • 118
Training the logistic regression model • 119
Evaluating the model and visualizing results • 122
Machine learning projects..... 124
Predicting house prices with linear regression • 124
Loading and preparing the data • 125
Selecting features and target • 126
Splitting the data into training and test sets • 126
Training the linear regression model • 127
Making predictions about house prices • 127
Evaluating model performance • 127
Displaying model coefficients • 127
Testing the model with new input • 128
Predicting diabetes with logistic regression • 129
Inspecting the diabetes dataset • 130
Selecting features and target variables • 131
Splitting the dataset into training and test sets • 131
Training the logistic regression model • 132
Making predictions with patient data • 132
Evaluating model performance • 132
Visualizing the confusion matrix • 133
Testing the model with a new patient record • 134
Classifying iris flower dataset using decision trees • 135
Importing libraries for decision tree classification • 135
Loading and preparing the Iris dataset • 136
Splitting features and target species • 137

Splitting the dataset into training and test sets • 137
Training the decision tree classifier • 137
Evaluating the model • 138
Visualizing the decision tree • 139
Predicting car prices using random forest • 140
Creating a sample car dataset • 141
Encoding categorical features • 142
Splitting the dataset into training and test sets • 144
Training and evaluating the random forest model • 144
Human activity recognition using smartphones dataset with random forest • 145
Downloading and preparing the HAR dataset • 145
Importing libraries for activity recognition • 145
Preprocessing the activity dataset • 146
Splitting the dataset into training and test sets • 147
Training the random forest classifier • 147
Making predictions on the test set • 147
Evaluating the model • 147
Visualizing the activity confusion matrix • 148
Predicting employee attrition using XGBoost • 149
Preprocessing the attrition dataset • 151
Splitting features and target variable • 152
Splitting the dataset into training and testing sets • 152
Training the XGBoost classifier • 152
Making predictions and evaluating the model • 152
Visualizing the attrition confusion matrix • 153
Predicting customer churn using logistic regression • 154
Importing libraries for churn prediction • 155
Loading and inspecting the churn dataset • 155
Preprocessing and encoding categorical features • 157
Splitting the dataset into training and test sets • 157
Scaling features and training the model • 157
Evaluating the model • 158
Detecting credit card fraud using scikit-learn • 159
Importing libraries for fraud detection • 159
Loading and preparing the fraud dataset • 160

Splitting and scaling the dataset • 160
Initializing and training the random forest classifier • 161
Evaluating the model • 161
Disease prediction using machine learning algorithms • 163
Loading and exploring the heart disease dataset • 163
Preprocessing the heart disease dataset • 164
Splitting the dataset into training and test sets • 165
Training multiple machine learning models • 165
Model selection and evaluation • 166
Visualizing the heart disease confusion matrix • 167
Predicting heart disease risk in new patient data • 168
Forecasting and prediction projects 169
Weather forecasting using historical data • 169
Importing libraries for weather forecasting • 169
Preparing the historical weather data • 169
Selecting features and target variables • 170
Splitting the data into training and test sets • 170
Training the linear regression model • 170
Making predictions with the trained model • 171
Evaluating model performance • 171
Visualizing actual and predicted temperatures • 171
Predicting temperature for new data • 172
Predicting stock prices with linear regression • 173
Creating a sample historical stock price dataset • 173
Preparing features and target variables • 174
Splitting the dataset into training and test sets • 175
Training the linear regression model • 175
Predicting closing prices on test data • 175
Evaluating model performance • 176
Visualizing actual vs. predicted stock prices • 176
Predicting stock price for a future date • 178
Summary 179

Chapter 6: Vision, Language, Recommendations and AI Projects 181

Technical requirements..... 181

Deep learning and computer vision projects 182

Building an image classifier using Keras and TensorFlow • 183

Importing libraries for image classification • 183

Loading and preprocessing the MNIST dataset • 184

Constructing the CNN model • 185

Compiling the model • 186

Training the model • 187

Evaluating the model • 187

Making predictions on test images • 188

Visualizing digit predictions • 188

Classifying dogs vs. cats with a convolutional neural network • 190

Importing libraries for CNN image classification • 191

Preparing the dataset paths • 191

Defining image data generators for augmentation and rescaling • 191

Defining the validation data generator • 192

Loading training and validation data • 192

Defining the CNN model architecture • 193

Compiling the model • 194

Printing the model summary • 194

Training the CNN model • 194

Plotting training and validation accuracy and loss • 195

Making predictions on new images • 196

Interpreting the classification results • 197

Building a basic neural network from scratch with NumPy • 197

Defining activation and loss functions • 198

Initializing the neural network class • 198

Implementing the forward pass • 199

Implementing the backward pass and weight updates • 200

Implementing the training loop • 201

Training with the XOR dataset • 202

Natural language processing (NLP) projects 203

Sentiment analysis on text data using NLTK • 203

Importing libraries for sentiment analysis • 204

Feature extraction for sentiment analysis • 205

Loading and preprocessing the movie reviews dataset • 205

Shuffling the dataset • 206
Preparing the training and testing sets • 206
Training the Naive Bayes classifier • 206
Evaluating the classifier • 206
Displaying the most informative features • 207
Defining the sentiment analysis function • 207
Testing the classifier with custom movie review sentences • 208
Frequency-based text summarization with NLTK • 209
Importing NLTK and downloading resources • 209
Defining the input text • 210
Creating the frequency-based summarization function • 210
Detecting fake product reviews using NLP techniques • 213
Importing libraries for fake review detection • 213
Loading and inspecting the reviews dataset • 214
Extracting features and splitting the dataset • 214
Vectorizing the review text with TF-IDF • 215
Training and evaluating the fake review classifier • 216
Detecting emotions in text using the Natural Language Toolkit • 217
Importing libraries for emotion detection • 217
Downloading required NLTK data • 218
Importing and initializing the sentiment analyzer • 218
Creating sample text for emotion detection • 218
Writing a function for emotion detection • 218
Detecting and printing the emotion • 220
Named entity recognition (NER) with spaCy • 220
Installing spaCy and downloading the pre-trained model • 221
Importing libraries for named entity recognition • 221
Defining the sample text and processing with spaCy • 221
Extracting named entities into a DataFrame • 222
Visualizing named entities with displaCy • 224
Saving extracted entities to a CSV file • 224
Detecting fake news using Naive Bayes • 225
Importing libraries for fake news detection • 226
Loading and inspecting the news dataset • 226
Defining features and splitting the dataset • 227

Vectorizing news text with TF-IDF • 227
Training the multinomial Naive Bayes classifier • 228
Evaluating the model • 228
Building a resume scanner using keyword extraction • 229
Importing libraries for resume scanning • 229
Creating the sample resume dataset • 230
Defining the job description • 230
Vectorizing resumes and computing similarity scores • 231
Filtering and displaying matching resumes • 232
Automatic essay grading using BERT • 233
Importing libraries for essay grading • 233
Loading and exploring the dataset • 234
Confirming the selected columns • 234
Preprocessing the data • 235
Creating a custom dataset class • 235
Splitting the dataset into training and validation sets • 236
Defining the essay grading model • 237
Initializing the model, optimizer, and loss function • 237
Training the essay grading model • 237
Evaluating the essay grading model • 238
Setting up the training loop • 239
Testing the essay grading model • 239
Visualizing training and validation loss • 241
Recommendation system projects..... 243
Building a movie recommendation system using cosine similarity • 243
Importing libraries for movie recommendation • 243
Creating the sample movie dataset • 244
Converting the dataset to a DataFrame • 244
Defining the TF-IDF vectorizer • 244
Transforming the genre column • 245
Computing the cosine similarity matrix • 245
Defining the recommendation function • 245
Testing the recommendation system • 246
Building a book recommendation system using collaborative filtering • 247
Importing libraries for book recommendation • 247

Creating a sample book ratings dataset • 247
Building the user-book matrix • 248
Computing user similarity scores • 248
Defining the book recommendation function • 249
Generating book recommendations for users • 250
Predicting movie ratings using collaborative filtering • 251
Downloading and preparing the MovieLens dataset • 251
Importing libraries for rating prediction • 252
Loading and previewing the dataset • 252
Preprocessing the dataset for the surprise library • 253
Splitting the dataset into training and test sets • 253
Building and training the SVD model • 253
Evaluating the model with RMSE • 254
Predicting a specific user and movie • 254
Visualizing the distribution of movie ratings • 254
AI and game projects 255
Building a tic-tac-toe AI with the minimax algorithm • 255
Representing the Tic-Tac-Toe board • 256
Printing the Tic-Tac-Toe board • 256
Checking for a winner • 256
Checking if the board is full • 257
Evaluating the board for the minimax algorithm • 257
Implementing the minimax algorithm • 258
Finding the best move for the AI • 259
Implementing the main game loop • 260
Building a simple personal assistant using Python speech libraries • 263
Importing libraries for the personal assistant • 264
Initializing the speech engine • 265
Creating the speak function • 265
Taking voice commands • 265
Creating the respond function • 266
Running the assistant • 268
Starting the application • 268
Summary 269

Chapter 7: Unlock the Code Bundle and the PDF Version **271**

Unlock this book's free benefits in three easy steps **272**

Other Books You May Enjoy **276**

Index **279**

Preface

Artificial intelligence has transformed from a theoretical discipline confined to research laboratories into a practical, accessible toolkit that powers intelligent systems across industries. Python has emerged as the language of choice for AI development, offering a rich ecosystem of libraries, frameworks, and tools that democratize the creation of intelligent applications. Yet the journey from understanding AI concepts to building production-ready systems remains challenging for many developers and data scientists.

This book addresses that challenge head-on. It provides a comprehensive, hands-on approach to learning practical AI development with Python, starting from foundational concepts and progressing systematically to advanced applications. The strength of this book lies in its balance between theoretical understanding and practical implementation. Each concept is introduced, explained, and immediately applied through code examples and real-world projects.

Throughout this journey, you will discover that artificial intelligence is not an isolated skill, but rather a convergence of programming, mathematics, data manipulation, and software engineering practices. By mastering these interconnected skills, you will be equipped to build intelligent systems that solve real problems.

Key features of this book include:

- Hands-on projects that reinforce learning through practical implementation, from beginner to advanced levels
- Real-world datasets and scenarios that mirror actual AI development challenges
- Step-by-step guidance through complex topics, making advanced concepts accessible
- Integration of multiple AI domains including machine learning, deep learning, NLP, and computer vision
- Production-ready deployment strategies using Flask and cloud platforms
- Best practices and practical patterns distilled from real-world experience

In writing this book, I have drawn from years of experience building and deploying AI systems in diverse domains. The projects and examples reflect actual challenges faced during real AI development, ensuring that the knowledge you gain is immediately applicable.

Who this book is for

This book is designed for:

- Software developers seeking to integrate AI capabilities into their applications
- Data scientists transitioning from analysis to model building and deployment
- Students and self-taught practitioners beginning their journey into artificial intelligence
- Technical professionals and engineers looking to implement intelligent systems
- Teams seeking to establish AI competencies within their organizations

To get the most from this book, you should have basic familiarity with Python programming (variables, functions, and control flow). While some mathematical concepts are referenced, the book emphasizes practical implementation over mathematical derivation. Prior experience with machine learning is not required; the book builds knowledge progressively.

What this book covers

Chapter 1, *Python for AI and Data Science Foundations*, covers fundamental Python constructs, data structures, and essential libraries (NumPy, Pandas, Matplotlib, Seaborn). You will learn to load, clean, and visualize data, and perform exploratory data analysis on real datasets.

Chapter 2, *Machine Learning and Classification Models*, introduces machine learning fundamentals and covers supervised learning, regression, and classification. Hands-on projects guide you through building predictive models for house prices and spam detection, with emphasis on model evaluation techniques.

Chapter 3, *Neural Networks and Natural Language Processing*, covers neural network architecture, backpropagation, and deep learning with TensorFlow. You will build models for image classification (MNIST) and sentiment analysis using pre-trained NLP models.

Chapter 4, *Deploying an AI Model as a Web Service*, teaches you to deploy AI models as REST APIs using Flask and cloud platforms like Heroku, making your models accessible to end users.

Chapter 5, *Supervised Learning and Forecasting Projects*, consolidates earlier knowledge through beginner and intermediate projects: a command-line calculator, a rule-based chatbot, and a spam email detector.

Chapter 6, *Vision, Language, Recommendations and AI Projects*, extends to advanced projects including image classification with CNNs, natural language processing applications, recommendation systems, and AI-driven games, using frameworks like PyTorch and Hugging Face Transformers.

To get the most out of this book

To benefit fully from this book, follow these recommendations:

- Install all required libraries in advance using the pip commands provided at the beginning of each chapter
- Work through code examples interactively in Jupyter Notebook or your preferred IDE, rather than passively reading
- Modify and experiment with code examples to deepen your understanding
- Complete all hands-on projects, attempting to solve problems independently before reviewing solutions
- Access the GitHub repository to download datasets and complete code examples

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



2. Search for this book

3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to the *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Execute the `train_model()` function."

A block of code is set as follows:

```
import pandas as pd
import numpy as np

data = pd.read_csv('dataset.csv')
print(data.head())
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Use the **confusion matrix** to evaluate classifier performance."

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

```
python <file_name>
```

Note

Warnings or important notes appear like this.

Tip

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 or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

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 reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

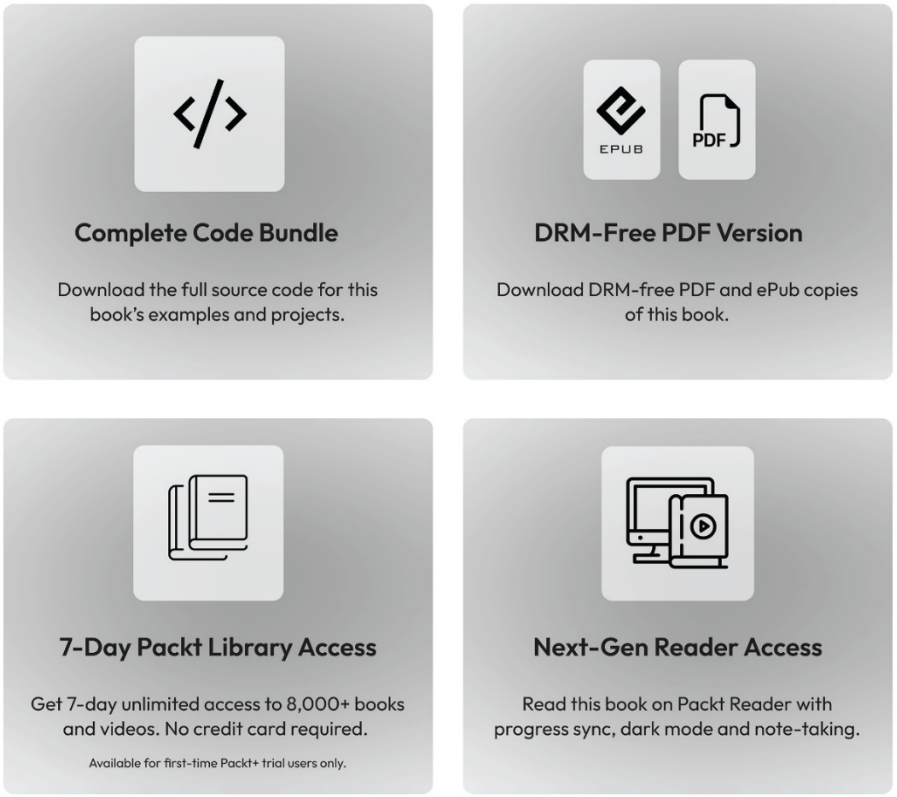
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

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 `http://authors.packt.com/`.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.



Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *Hands-On AI Development with Python*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



`https://packt.link/r/1808088530`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Python for AI and Data Science Foundations

A solid understanding of Python forms the backbone of modern artificial intelligence and data science, enabling practitioners to translate complex ideas into functional solutions. Covering foundational programming concepts alongside essential tools and libraries, Python serves as the primary language for building, testing, and deploying AI-driven applications. Mastery of these basics empowers individuals to efficiently manipulate data, automate tasks, and construct the algorithms that drive intelligent systems.

This chapter focuses on becoming familiar with Python, emphasizing fundamental programming skills that are essential for AI development. It begins with basic Python constructs and progresses toward the use of popular libraries such as NumPy and Pandas for data manipulation.

The topics that will be covered in this chapter are:

- Introduction to Python and installation
- Installing Jupyter Notebooks
- Python basics
- Working with lists
- Working with dictionaries
- Introduction to NumPy
- Introduction to Pandas
- Hands-on project: Basic data manipulation and file handling
- Introduction to exploratory data analysis

- Data transformation and feature engineering
- Visualizing data with Matplotlib and Seaborn
- Hands-on project: Exploratory data analysis on a real dataset

By the end of this chapter, you will be able to:

- Gain an understanding of basic Python syntax and the process of writing simple Python programs
- Use variables, data types, and operators effectively
- Write functions, use loops, and implement control flow using if-else statements
- Read from and write to files
- Receive an introduction to the two core libraries, NumPy and Pandas, which are foundational for AI development
- Load data from various file formats such as CSV and Excel
- Clean and preprocess data by handling missing values and outliers
- Create data visualizations to explore relationships between variables
- Understand basic descriptive statistics and their role in data analysis

Technical requirements

To follow along with the examples and hands-on projects in this chapter, you will need the following:

- Python 3.8 or above
- Jupyter Notebook
- The following Python libraries installed via `pip`:
    - `numpy`
    - `pandas`
    - `matplotlib`
    - `seaborn`
    - `scikit-learn`

You can install these libraries using the following command:

```
pip install numpy pandas matplotlib seaborn scikit-learn
```

Additionally, you will need the following datasets:

- MBA Admission dataset from Kaggle (`https://www.kaggle.com/datasets/nikhil7280/mba-admission`)
- Titanic dataset from Kaggle (`https://www.kaggle.com/datasets/titanic`)

A free Kaggle account is required to download these datasets.

All code examples in this chapter assume that you have Jupyter Notebook installed and running. If you have not yet installed Jupyter Notebook, refer to the *Installing Jupyter Notebooks* section later in this chapter for installation instructions.

> **Note**
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Introduction to Python and installation

Python is described as a high-level, versatile programming language that is widely used in AI and data science because of its simplicity and the extensive ecosystem of libraries available. The official Python website, `https://www.python.org`, provides documentation and information about Python, including sections such as getting started, download, docs, and jobs. However, we will only consider the aspects relevant to this book.

To begin working with Python, installation is required. The recommended approach is to visit `https://www.python.org/downloads`, where a button is available to download the latest version of Python, as shown in *Figure 1.1*:

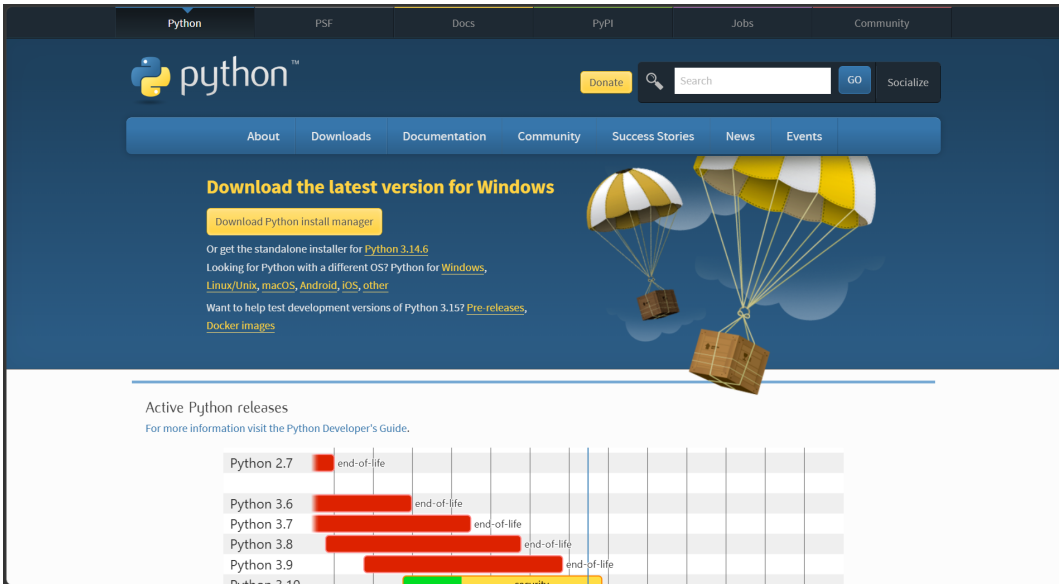


Figure 1.1: Python.org downloads section showing the options available

Note

The Python version can change while you are viewing this chapter.

The website automatically suggests the latest version suitable for the user's operating system, but options for Windows, Linux, macOS, and others are also available.

The screenshot in *Figure 1.2* displays the Python releases for Windows page, where various installers for different Windows architectures (64-bit, 32-bit, ARM64) can be selected:

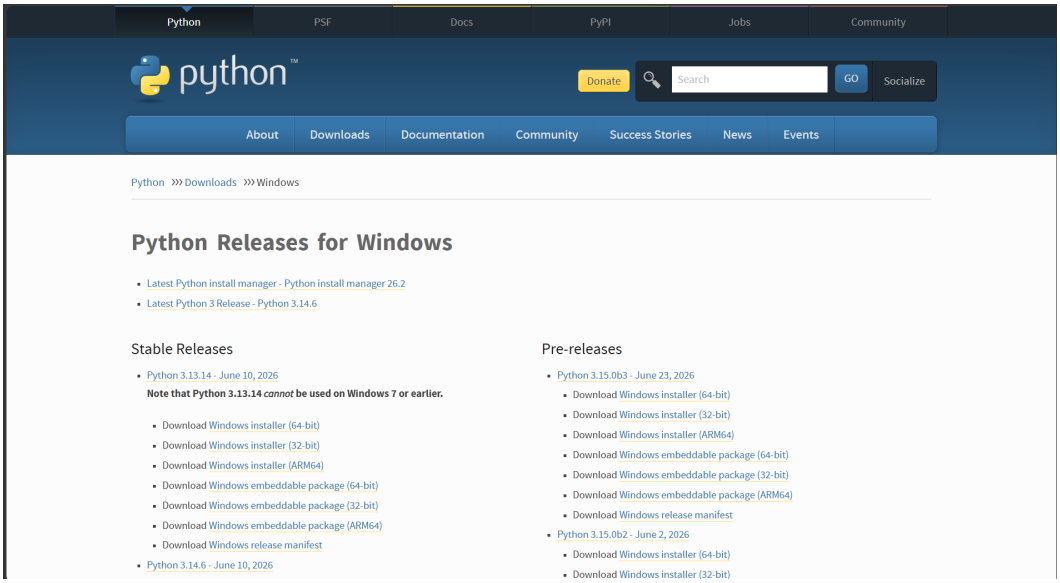


Figure 1.2: Python releases for Windows page showing download links for different Windows installers

Similarly, *Figure 1.3* shows the download page for macOS, where the latest version of Python can be downloaded directly:

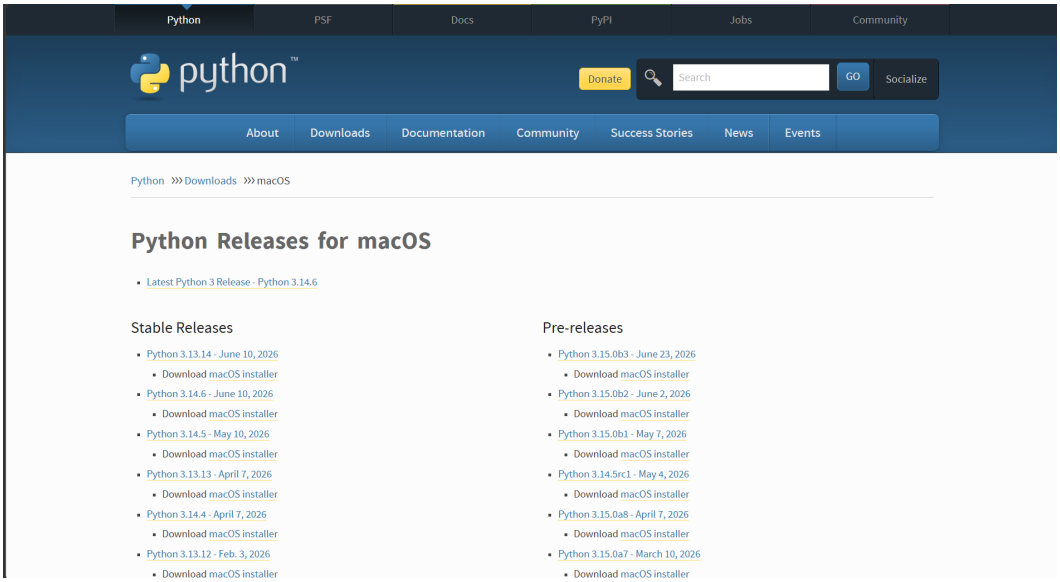


Figure 1.3: Python download page for macOS

Once the installer is downloaded, the installation process proceeds as follows:

1. Click on the downloaded installer to launch it
2. Read the installation guide
3. Continue to the next step
4. Review the history and license agreement
5. Continue and agree to the license
6. Click **Install**
7. Enter the system password if prompted
8. Approve the installation of the software

Wait for the installation to complete and close the installer when finished. After installation, to verify if the Python installation was successful, open the terminal and write the following command to display the installed Python version:

```
python --version
```

If the terminal displays the version of Python downloaded from the website, then the installation is successful.

Installing Jupyter Notebook

The next required tool is **Jupyter Notebooks**. The installation instructions are available at `https://www.jupyter.org/install`. These installation guidelines also include the instructions to download JupyterLab, but our focus is on installing Jupyter Notebooks rather than JupyterLab. Use the following command in the command prompt to install Jupyter Notebooks:

```
pip install notebook
```

Once installed, Jupyter Notebook can be launched by running the following command:

```
jupyter notebook
```

This command opens a new notebook in the browser, running from the local environment. A new folder can be created on the desktop, and a new notebook can be created inside this folder. The Jupyter Notebook interface allows writing and running Python code interactively.

Let's begin using Jupyter Notebook by writing our first program to print hello world.

Writing your first program in Jupyter Notebook

Let's write a simple program to demonstrate basic Python syntax. In the Jupyter Notebook, create a new folder named `code`, and create a new file inside it. Write the following code in the cell of the file:

```
print("Hello World!")
```

This code uses the `print()` function to display the string `"Hello World!"` on the screen. The string is enclosed in double quotes, indicating it is a string data type. The following is the output when you run the code:

```
Hello World!
```

The notebook interface provides several ways to execute the code: the play button above the cell, the **Run** menu with the **Run Selected Cell** option, or the displayed keyboard shortcut.

The string inside the print function can be modified. For example, changing the text to "Hello My World!" and running the cell again will update the output accordingly. The following code prints "Hello My World!" to the output:

```
print("Hello My World!")
```

Further modifications to the string, such as introducing a space within the word "World," will also be reflected in the output after running the cell. The following code prints "Hello My World!" to the output:

```
print("Hello My World!")
```

Each time a modification is made to the code cell, an orange indicator appears next to the cell, signaling that changes have occurred. Additionally, when the cell is executed, a number also appears next to the cell, indicating how many times the cell has been run or how many print statements have been executed.

Now that we have understood the setup and print statements, let's move on to Python basic concepts.

Python basics

Now that you have set up Python and Jupyter Notebook, it's time to explore the foundational building blocks of Python programming. Understanding these basics, starting with how to

store and work with data through variables and data types, is essential for writing effective Python code. The following sections walk through these concepts with practical examples that you can execute in your notebook.

Variables and data types

Variables are used to store data, and Python supports multiple data types such as integers, floats, strings, and Booleans. In Python, there is no need to declare a variable with a specific type or keyword as required in some other programming languages. Instead, a variable can be created and assigned a value directly.

The following code demonstrates the creation of variables with different data types:

```
x = 5
y = 2.5
name = "Vivian"
is_student = False
```

In this example, `x` is an integer, `y` is a float, `name` is a string, and `is_student` is a Boolean. The Boolean type can have values of either `True` or `False`.

Note

Comments in Python are created by placing a hash (#) or pound sign before the text, which is useful for documentation or providing notes to other developers.

Operators

Python provides several **arithmetic operators**, including addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and modulus (%). The following code demonstrates the use of arithmetic operators:

```
# Operators +, -, *, /, **, %

sum = x + y
sub = x - y
product = x * y
div = x / y
power = x ** 2
mod = x % 2

print(sum)
```

```
print(sub)
print(product)
print(div)
print(power)
print(mod)
```

This will produce the output as:

```
25
```

The code adds x and y and stores it in the sum variable, subtracts x and y and stores it in the sub variable, multiplies them and stores the result in the product variable, divides them and stores it in the div variable, raises x to the power of 2 and stores the result in the power variable, and finally calculates the mod of x and y and stores the result in the mod variable.

Control flow

Control flow in Python is managed using `if-else` statements and loops such as the `for` loop and `while` loop. Let's begin with the if-else statements.

If-else statements

Control flow has if-else statements, which execute certain code based on a particular condition. For example, the following code checks whether x is greater than y and prints a message accordingly:

```
if x > y:
    print(f"{x} is greater than {y}")
else:
    print(f"{x} is less than or equal to {y}")
```

The output produced by this is:

```
5 is greater than 2.5
```

If the value of y is increased to 20 and the cell is run again, the output changes to indicate that x is less than or equal to y:

```
5 is less than or equal to 2.5
```

This is how the `if-else` statements execute: if the condition in the if block is `true`, it performs the action; otherwise, it executes the `else` block.

For loop

The `for` loop can be used to iterate over a sequence of numbers. The following code demonstrates a for loop that prints numbers from 0 to 4:

```
for i in range(5):
    print(i)
```

The output produced by this for loop is:

```
0
1
2
3
4
```

This loop iterates through each number from 0 up to, but not including, 5 and prints each number. The value of variable `i` starts at 0 and increments by 1 on each iteration until it reaches 5. When `i` becomes 5, it is out of range, and the loop ends.

While loop

The **while loop** in Python allows repeated execution of a block of code as long as a specified condition remains `true`. The following code demonstrates a while loop where the variable `j` is initialized to 0. The loop continues to execute as long as `j` is less than 5 (`j<5`). Inside the loop, the current value of `j` is printed, and then `j` is incremented by 1 using the increment operator `+=`:

```
j = 0
while j < 5:
    print(j)
    j += 1
```

The output produced is:

```
0
1
2
3
4
```

In this example, the output displays the numbers 0 through 4. The loop checks if `j` is less than 5 before each iteration. When `j` reaches 5, the condition is no longer `true`, so the loop exits without printing 5.

Let's change the condition and start value. For example, if `j` is set to 4 and the loop condition is changed to `j < 8`, the loop prints the numbers 4, 5, 6, and 7. When `j` reaches 8, the condition `j < 8` is no longer satisfied, so the loop exits as follows:

```
j = 4
while j < 8:
    print(j)
    j += 1
print("I am out")
```

The output is as follows:

```
4
5
6
7
I am out
```

After the loop, a message is printed to indicate that the loop has finished.

Functions

Functions in Python enable code reusability by encapsulating logic within callable blocks. A function is created using the `def` keyword, followed by the function name and any parameters inside parentheses. The function block itself is not executed until it is called. The following code defines a function named `greet` that takes a parameter called `name` and returns a greeting string:

```
def greet(name):
    return f"Hello, {name}"
```

When this function is defined, nothing is executed immediately. Functions exist as callable blocks and require an explicit **call** to run their logic. To execute the function and display its result, the function can be called within a print statement, passing a string as the argument:

```
print(greet("Vivian Aranha"))
```

After we call the function, the following output is produced:

```
Hello, Vivian Aranha
```

In this example, the function greet is called with the argument "Vivian Aranha". The function receives this value as the parameter name, constructs the greeting string, and returns it. The print statement then outputs the returned string. We will be using functions a lot throughout this book.

Working with lists

A **list** in Python is a collection of items. Python lists are dynamic, which means they can grow or shrink as needed. For example, a list called `numbers` can be created to store an array-like structure containing the numbers one through five. The following code creates a list of five numbers and prints the item at index zero:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[0])
```

It prints the item at the given index:

```
1
```

In this example, the list contains five items, with indices starting from zero. It prints one as the output because the item at index zero is one. If you want to index the item at index three, use the following:

```
print(numbers[3])
```

The item at index three is the number four. Additionally, there are several common list operations in Python such as:

- `append()` adds an item to the end of the list
- `remove()` deletes a particular item from the list by its value
- `pop()` removes an item from a list by its index
- `slice()` returns a subset of the list

For example, the following code appends the number six to the end of the list numbers:

```
numbers.append(6)
print(numbers)
```

After appending, the list contains one more number at the end:

```
[1, 2, 3, 4, 5, 6]
```

The value appended does not have to be in order. We can append any number; for instance, the following code appends the number seven to the list and prints the result, displaying the updated list:

```
numbers.append(7)
print(numbers)
```

This appends seven to the end of the list as follows:

```
[1, 2, 3, 4, 5, 6, 7]
```

In summary, whatever data is passed to append gets added to the end of the list.

Working with dictionaries

A **dictionary** in Python stores data as key-value pairs, making it useful for organizing related data. A dictionary is defined using curly brackets, with each item consisting of a key and its corresponding value. For example, a variable named `student` can be created as a dictionary containing information about a person:

```
student = {
    "name": "Vivian",
    "age": 40,
    "is_student": True
}
```

In this dictionary, the keys are `"name"`, `"age"`, and `"is_student"`, and their corresponding values are `"Vivian"`, `40`, and `True`. To display data from a dictionary, the key is used instead of an index. For example, accessing the age of the student is done as follows:

```
print(student["age"])
```

It prints the value present in the `"age"` key:

```
40
```

Similarly, accessing the name of the person is done by specifying the `"name"` key:

```
print(student["name"])
```

The output will be:

```
Vivian
```

This approach performs the retrieval of values based on their keys, rather than relying on numerical indices.

Introduction to NumPy

NumPy is a powerful library which is used for numerical computation in Python. NumPy provides support for arrays, matrices, and many mathematical functions. Let's see how to create arrays using NumPy.

Creating arrays using NumPy

To create an array using NumPy, the `numpy` library must be imported first. The following code imports numpy as `np`, then creates a variable named `array` using `np.array` with a list of numbers from 1 to 5. Printing the array displays its contents:

```
import numpy as np
array = np.array([1, 2, 3, 4, 5])
print(array)
```

This code prints the array created:

```
[1 2 3 4 5]
```

Array operations

NumPy arrays support element-wise operations. For example, to add two arrays together, another array can be created and added to the first. The following code creates a second array with values `[5, 4, 3, 2, 1]` and prints the result of adding the two arrays:

```
array_2 = np.array([5, 4, 3, 2, 1])
print(array + array_2)
```

The output prints the addition of the elements of the two arrays:

```
[6 6 6 6 6]
```

The addition operation added the number present at the zero index of `array` with the number present at the zero index of `array_2`; similarly, the number present at index one of `array` got added to the number present at index one of `array_2`. This is repeated for all the other elements of the two arrays.

If both arrays contain the same numbers, such as `[1, 2, 3, 4, 5]`, the addition produces a new array with each element doubled. The following code demonstrates this:

```
array_2 = np.array([1, 2, 3, 4, 5])
print(array + array_2)
```

The output is as follows:

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

Matrix operations

NumPy also supports matrix manipulations, which are useful in machine learning. A **matrix** in NumPy is defined as a two-dimensional array. For example, a 2 x 2 matrix can be created with the following code:

```
matrix = np.array([[1, 2],
                   [3, 4]])
```

To perform matrix multiplication, the `np.dot` function is used. Multiplying a matrix by itself produces the matrix product. The following code demonstrates this operation:

```
print(np.dot(matrix, matrix))
```

This operation produces the product of the matrix with itself:

```
[[ 7 10]
 [ 5 22]]
```

Let us create another matrix, for example with values `[[4, 3], [2, 1]]`, and the product of the two matrices can be computed in the same way:

```
matrix = np.array([[1, 2],
                   [3, 4]])

matrix_2 = np.array([[4, 3],
                     [2, 1]])

print(np.dot(matrix, matrix_2))
```

The output matrix produced will be:

```
[[ 8  5]
 [ 20 13]]
```

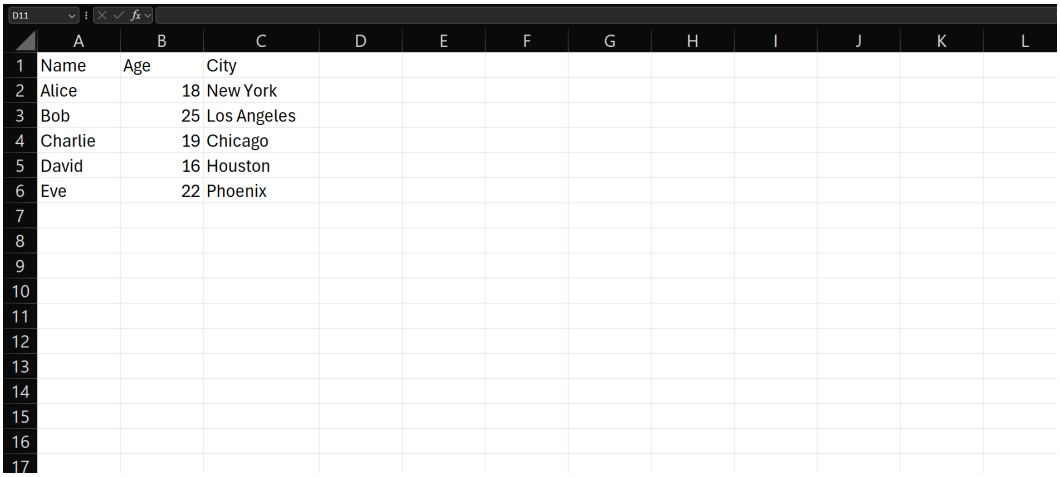
This demonstrates multiplication of two matrices or the product of these two matrices.

Introduction to pandas

Pandas is a library used for data manipulation and analysis. It provides data structures like **DataFrames**, which make working with structured data easy. This is a significant part of AI applications.

Creating a CSV file

To demonstrate loading and inspecting data, let's create a CSV file. Open Excel and create a blank document. Label the columns as **Name**, **Age**, and **City** to structure the data. Next, populate the data in the columns as shown in *Figure 1.4*:



	A	B	C	D	E	F	G	H	I	J	K	L
1	Name	Age	City									
2	Alice	18	New York									
3	Bob	25	Los Angeles									
4	Charlie	19	Chicago									
5	David	16	Houston									
6	Eve	22	Phoenix									
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												

Figure 1.4: Excel window showing a worksheet with the columns labelled and populated with data

The next step is to save the Excel file as a CSV file. To do that, select the **Save As** option from **File**, name the file as you want to (we're naming it `data.csv`), and choose the format to be saved in **CSV (Comma delimited)** format as shown in *Figure 1.5*:

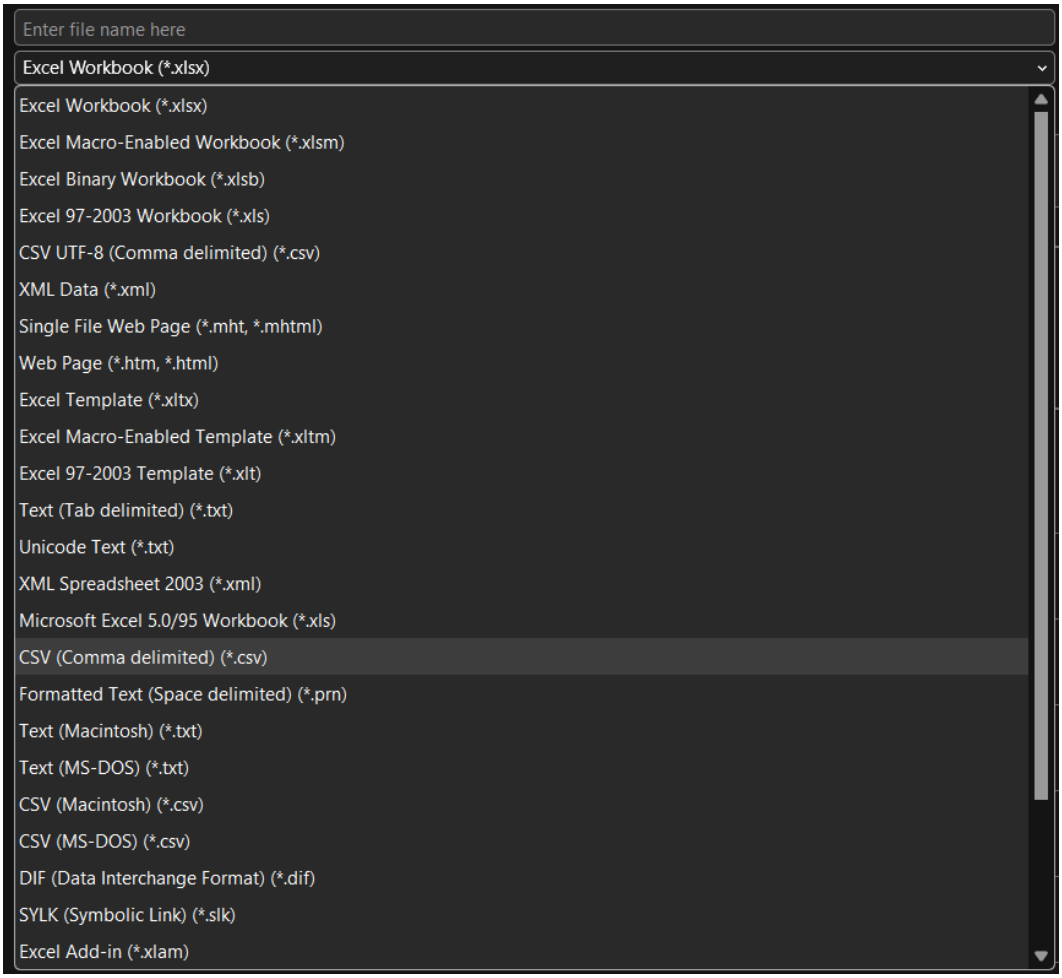


Figure 1.5: Save As dialog box showing the file types available to be saved

This will save the file in the required format.

Loading and inspecting data with pandas

To begin working with the data in Python, the pandas library must be imported. The following code imports pandas and assigns it the alias `pd`:

```
import pandas as pd
```

Next, the data is loaded from the CSV file using the `read_csv()` function. The following code reads the file named `data.csv` and assigns the resulting DataFrame to the variable `data`:

```
data = pd.read_csv('data.csv')
```

Note

When loading data from a CSV file, the file path depends on where your file is stored. If the CSV file is in the same directory as your Jupyter Notebook, you can simply use the filename as `pd.read_csv('data.csv')`. However, if the file is located elsewhere on your computer, you need to provide the full path. On Windows, paths can be specified in two ways: using a raw string such as `pd.read_csv(r"C:\Users\YourName\Documents\project\data.csv")`, or using forward slashes, which work across all operating systems, `pd.read_csv("C:/Users/YourName/Documents/project/data.csv")`.

To inspect the contents of the DataFrame, the `head()` method is used. This method displays the first few rows of the data:

```
print(data.head())
```

The output shows the first rows of the data that were entered in the CSV file, confirming that the data have been read correctly.

Data manipulation in pandas

To perform data manipulation, let's start by selecting a column from the DataFrame that we want to manipulate. For example, to select only the `Name` column, the following code is used:

```
name = data['Name']
print(name)
```

The output will be the column selected:

```
0      Alice
1        Bob
2    Charlie
3      David
4        Eve
Name: Name, dtype: object
```

To filter rows based on a condition, such as selecting only those entries where the `Age` is greater than 18, the following code is used:

```
filtered_data = data[data['Age'] > 18]
print(filtered_data)
```

This code will search the data, and where the condition is true (`Age > 18`), it returns that data:

```
    Name  Age         City
1    Bob   25  Los Angeles
2 Charlie  19      Chicago
4    Eve   22      Phoenix
```

The output shows the filtered data set, which includes only those individuals whose age is greater than 18.

To obtain descriptive statistics about the dataset, the `describe()` method is used. This method provides information such as count, mean, standard deviation, minimum, and maximum values for each numeric column. The following code prints these statistics:

```
print(data.describe())
```

The output description is as follows:

```
             Age
count   5.000000
mean   20.000000
std     3.535534
min    16.000000
25%    18.000000
50%    19.000000
75%    22.000000
max    25.000000
```

This output includes the count of items, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum values for the `Age` column. These statistics are useful for quickly understanding the distribution and summary of the dataset.

Hands-on project: Basic data manipulation and file handling

For this hands-on project, our task is to build a small program that reads from a text file, processes the content, and writes the result to another file. The steps for this exercise are as follows:

1. Read a text file with multiple lines of numbers (for example, `numbers.txt`)
2. Process the data by converting each number to its square
3. Write the output to another file (for example, `output.txt`)

To begin, create a new text file named `numbers.txt` by clicking on **New** and then **Text File**. Name this file `numbers.txt`, populated with numbers such as 1, 2, 3, 4, 5, 10, and 15 as shown in *Figure 1.6*:

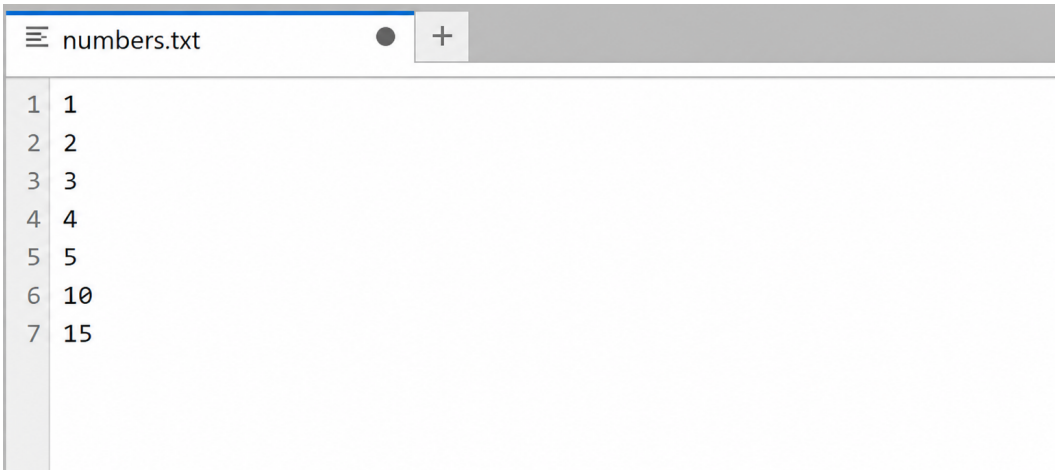


Figure 1.6: Numbers.txt file populated with numerical data

After saving and closing the file, the next step is to open the file in Python. The following code demonstrates how to open the file in **read mode** (`r`) using a context manager `with`, which ensures the file is properly closed after reading:

```
with open('numbers.txt', 'r') as file:
    numbers = file.readlines()
```

The `numbers.txt` file is opened in the `file` variable, which can be used to interact with the file inside. The `numbers` variable now contains a list of lines from the file, each representing a number.

Next step is to process the data by converting each number to its square. To do this, we will use list comprehension. The following code strips whitespace from each line, converts it to an integer, squares it, and appends a newline character:

```
squared_numbers = [str(int(num.strip()) ** 2) + '\n' for num in numbers]
```

Finally, the last step is to write the squared numbers to a new file named `output.txt`. The following code uses the **write mode** (`w`), which will create the `output.txt` file if it does not exist:

```
with open('output.txt', 'w') as file:
    file.writelines(squared_numbers)
```

After running the program, verify the output file `output.txt` that will be created. Open it and check if the numbers in the file are squares of the numbers present in the `numbers.txt` file. The input numbers were 1, 2, 3, 4, 5, 10, and 15; the output file will contain their squares: 1, 4, 9, 16, 25, 100, and 225. This demonstrates successful reading, processing, and writing of data using basic Python file handling and data manipulation techniques.

Introduction to exploratory data analysis

Exploratory data analysis (EDA) is an iterative process to analyze datasets with objectives centered on data exploration, cleaning, and visualization. The process involves loading datasets, handling missing values, and visualizing data using popular libraries such as Pandas, Matplotlib, and Seaborn. This foundational work is essential for understanding data prior to building any machine learning models.

Loading and inspecting data

Pandas is a widely used library that simplifies the process of loading data from different file types, including CSV, Excel, and SQL databases. Let's understand how to load a CSV file.

To obtain a dataset, **Kaggle** (`https://www.kaggle.com`) is used as a source. Kaggle provides access to a wide variety of datasets. The process for obtaining a dataset from Kaggle is as follows:

1. Navigate to the Kaggle website and select **Datasets** from the left menu
2. Create an account, if necessary, as most datasets require an account for download

3. Go to the **Education** section to find relevant datasets
4. Select the **MBA Admission dataset** for this exercise. We are using this one `https://www.kaggle.com/datasets/taweilo/mba-admission-dataset`
5. Download the dataset as a zip file
6. Unzip the downloaded file to extract `MBA.csv`
7. Move the extracted `MBA.csv` file into the project folder on Jupyter Notebook

Figure 1.7 displays the MBA Admission dataset page on Kaggle, showing the dataset description, metadata, and download options:

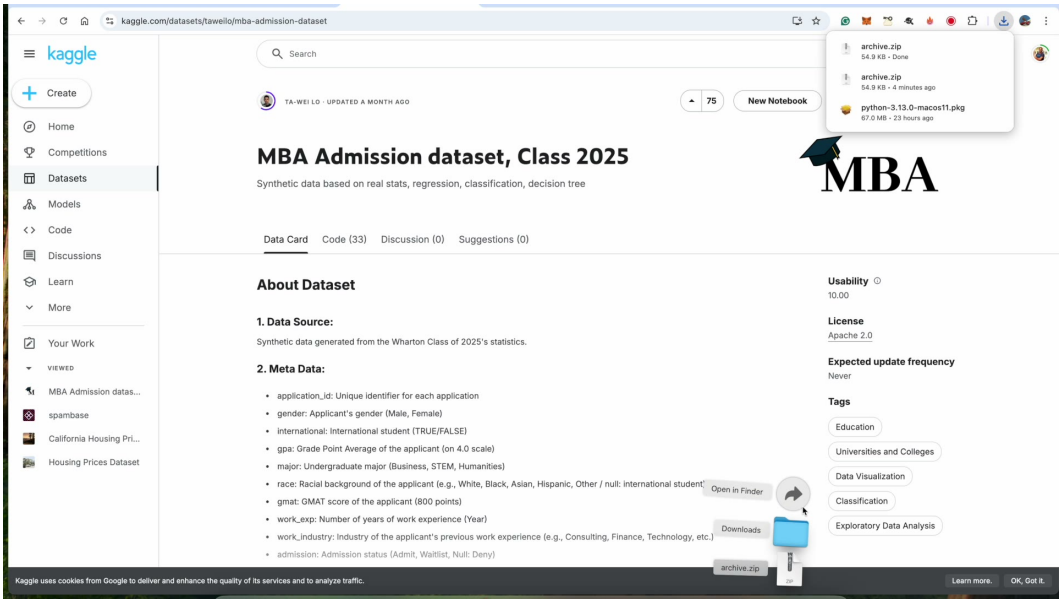


Figure 1.7: Kaggle MBA Admission dataset page with dataset description, metadata, and download options visible

After preparing the dataset, create a new Python 3 notebook. Follow these steps to load and inspect the data:

1. Import the pandas library using the alias `pd`
2. Load the CSV file using `pd.read_csv`, specifying the filename `mba.csv`
3. Inspect the first few rows of the dataset using the `head()` method

The following code imports pandas, loads the CSV file, and displays the first few rows of the dataset:

```
import pandas as pd
data = pd.read_csv('MBA.csv')
data.head()
```

This produces the output as:

	application_id	gender	international	gpa	major	race	gmat	work_exp	work_industry	admission
0	1	Female	False	3.30	Business	Asian	620.0	3.0	Financial Services	Admit
1	2	Male	False	3.28	Humanities	Black	680.0	5.0	Investment Management	NaN
2	3	Female	True	3.30	Business	NaN	710.0	5.0	Technology	Admit
3	4	Male	False	3.47	STEM	Black	690.0	6.0	Technology	NaN
4	5	Male	False	3.35	STEM	Hispanic	590.0	5.0	Consulting	NaN

Figure 1.8: Jupyter Notebook cell displaying the first few cells of the MBA admission dataset

This approach allows for an initial inspection of the dataset's structure and contents.

Inspecting data structures

To quickly understand the structure of a dataset in pandas, several predefined steps can be followed. These steps provide information about the shape, columns, and summary of the data. To determine the shape of the dataset, which indicates the number of rows and columns, the following code is used:

```
print(data.shape)
```

The shape of the current dataset is:

```
(6194, 10)
```

The output shows that the dataset contains 6,194 rows and 10 columns.

To view the column names present in the dataset, the following code is executed:

```
print(data.columns)
```

The output is as follows:

```
Index(['application_id', 'gender', 'international', 'gpa', 'major', 'race',
'gmat', 'work_exp', 'work_industry', 'admission'], dtype='object')
```

This output lists all the columns, including application ID, gender, international status, GPA, major, race, GMAT score, work experience, work industry, and admission status.

To obtain a summary of the entire dataset, including the number of non-null entries and data types for each column, the following code is used:

```
print(data.info())
```

The output is as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6194 entries, 0 to 6193
Data columns (total 10 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   application_id  6194 non-null   int64
 1   gender          6194 non-null   object
 2   international   6194 non-null   bool
 3   gpa             6194 non-null   float64
 4   major           6194 non-null   object
 5   race            4352 non-null   object
 6   gmat            6194 non-null   float64
 7   work_exp        6194 non-null   float64
 8   work_industry   6194 non-null   object
 9   admission       1000 non-null   object
dtypes: bool(1), float64(3), int64(1), object(5)
memory usage: 441.7+ KB
None
```

This summary provides details about each column, such as the number of non-null values and the data type (e.g., int64, object, bool, float64).

Handling missing data

Upon opening the `mba.csv` file, it is apparent that some information is missing in various places. Let us proceed to handling this missing data. Create a new notebook in Jupyter Notebook.

The first step is to load the data. The following code imports pandas and reads the `MBA.csv` file:

```
import pandas as pd
data = pd.read_csv('MBA.csv')
```

To verify that the data has been loaded, the head of the DataFrame is printed:

```
print(data.head())
```

The output is as follows:

	application_id	gender	international	gpa	major	race	gmat	work_exp	work_industry	admission
0	1	Female	False	3.30	Business	Asian	620.0	3.0	Financial Services	Admit
1	2	Male	False	3.28	Humanities	Black	680.0	5.0	Investment Management	NaN
2	3	Female	True	3.30	Business	NaN	710.0	5.0	Technology	Admit
3	4	Male	False	3.47	STEM	Black	690.0	6.0	Technology	NaN
4	5	Male	False	3.35	STEM	Hispanic	590.0	5.0	Consulting	NaN

Observe that in the output table, missing values are visible as `NaN` in the race and admission columns. The next step is to identify the missing data.

Identifying missing data

In real-world datasets, missing values are common, and it is crucial to address them appropriately before conducting further analysis. To identify missing data in the DataFrame, the following code is used:

```
print(data.isnull().sum())
```

The *isnull()* function checks each column for null values, and the `sum()` function returns the sum of missing entries per column. The following output indicates the number of missing values in each column, such as one missing value in `application_id`, one in `gender`, one in `international`, `1842` in race, and `5194` in `admission`:

The output is as follows:

```
application_id       0
gender               0
international        0
gpa                  0
```

```
major               0
race             1842
gmat                0
work_exp            0
work_industry       0
admission        5194
dtype: int64
```

Now we have the amount of missing data; the next step is to identify how to handle this missing data.

Techniques to handle missing data

Handling **missing data** is a critical step in preparing datasets for analysis. There are two primary options for addressing missing values: dropping them or filling them with specific values.

The first option is to drop missing values (using the `dropna()` method), which removes rows and columns containing missing data. The following code creates a new DataFrame, `data_clean`, by dropping all rows with missing values from the original DataFrame:

```
data_clean = data.dropna()
```

This operation results in a dataset where all rows with any missing values have been removed. To verify, run the following code:

```
print(data_clean.isnull().sum())
```

Figure 1.9 shows that all the missing values have been dropped:

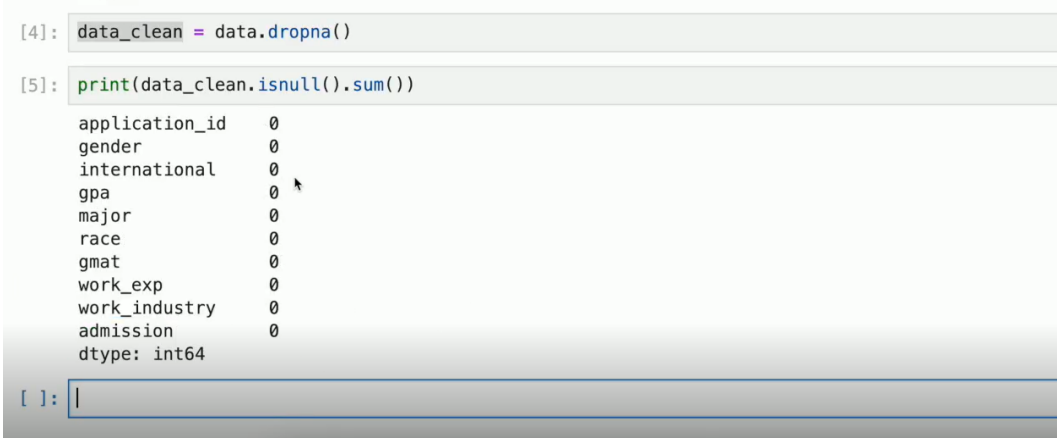


Figure 1.9: All the missing values have been removed from the dataset

The second option is to fill missing values by replacing them (using the `fillna()` method) with specific values, such as the mean, median, or a placeholder. For example, if the work experience column contains missing information, the following code replaces missing values in the work experience column with the mean value of that column:

```
data.fillna({'work_exp':data['work_exp'].mean()}, inplace=True)
```

This command calculates the mean of the work experience values and fills any missing entries in the column with this mean. If there are no missing values in the work experience column, this operation will have no effect. Alternatively, a fixed value can be used as a placeholder. For instance, to assign four years of work experience to all missing entries, the following code can be used:

```
data.fillna({'work_exp': 4}, inplace=True)
```

This approach fills all missing values in the work experience column with value 4.

You might be thinking about when to use each of them. So let me make it clear to you. Use `dropna()` under the following scenarios:

- When missing values represent less than 5 percent of your dataset
- When the missing data occurs randomly across the dataset
- When you have sufficient data and can afford to lose some rows
- When missing values in a specific column are concentrated in a few records
- When the analysis requires complete cases without imputed values

Use `fillna()` under the following scenarios:

- When missing values represent more than 5 percent of your dataset
- When removing rows would result in significant data loss
- When you can reasonably estimate or impute missing values based on the data
- When you need to preserve all observations for analysis or model training
- When domain knowledge suggests appropriate replacement values (mean, median, or forward fill)

These methods provide flexibility in handling missing data, allowing either the removal of incomplete records or the imputation of missing values with statistical measures or fixed placeholders.

Data transformation and feature engineering

Data transformation and feature engineering are essential steps in preparing data for machine learning models. This process includes **feature scaling** and **categorical data encoding**, also known as one-hot encoding.

Data transformation involves modifying data to meet the requirements of a machine learning model. Many algorithms expect the data to be in a specific format or range, so transforming the data ensures consistency and can improve the model's accuracy.

Feature scaling

Feature scaling is used to ensure that all numerical features are on the same scale. This prevents features with larger magnitudes from dominating those with smaller magnitudes. **Normalization** is especially important for algorithms such as k-nearest neighbors (K-NN), support vector machines (SVMs), and gradient descent-based methods, including neural networks.

One common method for feature scaling is **min-max scaling**, which scales features to a specific range, usually 0 to 1. The following code imports the `MinMaxScaler` from scikit-learn's `preprocessing` module:

```
from sklearn.preprocessing import MinMaxScaler
```

To apply min-max scaling, a `MinMaxScaler` object is initialized and used to transform the data. For example, if the dataset contains `'Age'` and `'Income'` as columns, the following code applies min-max normalization to these columns:

```
scaler = MinMaxScaler()
data[['Age', 'Income']] = scaler.fit_transform(data[['Age', 'Income']])
```

This code creates a scaler object and transforms the `'Age'` and `'Income'` columns so that their values are scaled between 0 and 1.

Categorical data encoding

Categorical data encoding is addressed using a technique called **one-hot encoding**. One-hot encoding converts categorical variables into a set of binary columns, where each category is represented as its own column containing 0s and 1s. Each row will have a 1 in the column corresponding to its category and 0s elsewhere.

One-hot encoding is applied when categorical features are present, particularly for algorithms that require numerical input.

The following code demonstrates how to perform one-hot encoding on a dataset using pandas. In this example, the columns of interest are `'Gender'` and `'Purchased'`:

```
data_encoded = pd.get_dummies(data, columns=['Gender', 'Purchased'])
```

This code creates a new DataFrame, `data_encoded`, in which the `'Gender'` and `'Purchased'` columns have been transformed into binary columns representing each category.

Visualizing data with matplotlib and seaborn

Visualization is a way to understand trends, distributions, and relationships in data. This is an important skill for an AI engineer, as visualizations can reveal patterns and insights that may not be immediately obvious from raw data.

Matplotlib basics

To begin working with Matplotlib, a dataset is required. Let's use the dataset we had created earlier `data.csv`, which consisted of columns Name, Age, and City. We'll add two extra columns to the already existing dataset as follows:

Income	Gender
15000	F
30000	M
18000	F
8000	M
28000	F

Table 1.1: Two extra columns to be added to the data.csv file

So, now the CSV file has 5 columns named Name, Age, City, Income, and Gender. Save this CSV file on Jupyter Notebook, and let's get started. To use Matplotlib, the `matplotlib.pyplot` module is imported as `plt`. This allows for the creation of various types of plots. The following code imports the Matplotlib pyplot module:

```
import matplotlib.pyplot as plt
```

Next, we'll create a line plot for age versus income. To do that, we'll use the `plt.plot` function, where the first argument is the age data and the second argument is the income data. This produces a line plot that visualizes the relationship between age and income in the dataset:

```
plt.plot(data['Age'], data['Income'])
```

To enhance the readability and interpretability of the plot, a title can be added to the age versus income visualization.

```
plt.title('Age vs Income')
```

Additionally, labeling the axes clarifies what each axis represents:

```
plt.xlabel('Age')
plt.ylabel('Income')
```

Finally, display the plot using `plt.show()`:

```
plt.show()
```

But we do not have the data in our code yet! Before plotting, the data must be loaded. This is accomplished by importing the pandas library and reading the CSV file containing the dataset as follows:

```
import pandas as pd
data = pd.read_csv('data.csv')
```

The entire code looks like this:

```
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('data.csv')
plt.plot(data['Age'], data['Income'])
plt.title('Age vs Income')
plt.xlabel('Age')
plt.ylabel('Income')
plt.show()
```

The resulting plot displays income as a function of age, with the axes and title clearly labeled. This visualization allows for an immediate understanding of how income varies with age in the dataset. Here's how the resulting plot looks like:

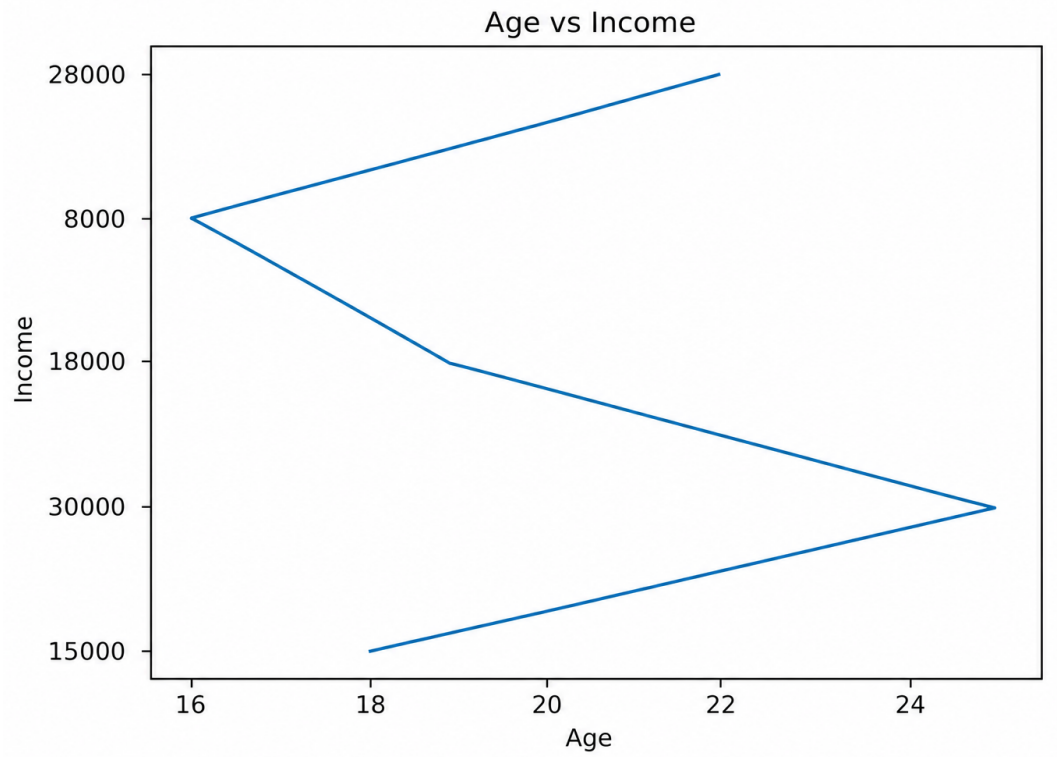


Figure 1.10: Line plot showing the relationship between age and income

To visualize categorical data, such as gender distribution, a bar chart can be created. This is accomplished by using the `value_counts()` method on the gender column to count the occurrences of each category, followed by the plot method with the argument `kind='bar'`. This approach converts the gender data into a count of values and uses that as input for the bar chart.

The following code generates a bar chart showing the distribution of gender in the dataset, assigns a title to the chart, and displays it:

```
data['Gender'].value_counts().plot(kind='bar')
plt.title('Gender distribution')
plt.show()
```

The resulting bar chart displays the counts of females and males in the dataset. In this case, the chart shows that there are three females and two males:

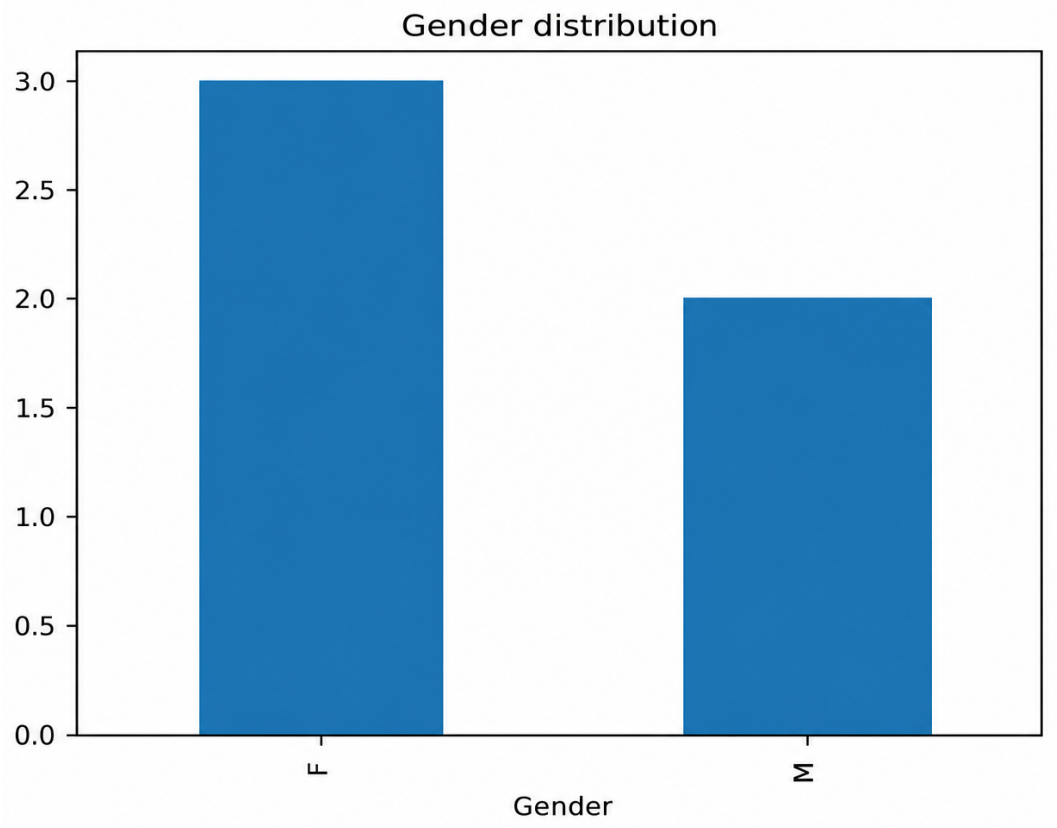


Figure 1.11: Bar chart displaying the distribution of genders in the dataset

The bar chart in *Figure 1.11* shows the gender distribution, with the x-axis labeled by gender categories and the y-axis indicating the count for each category.

Advanced data visualization with seaborn

Seaborn provides more aesthetic and advanced visualization options compared to basic plotting libraries. For example, Seaborn can generate a correlation heatmap, which visually represents the correlation coefficients between variables in a dataset.

Although the current dataset is too small for a meaningful correlation heat map, here's an example of how such a plot would be created. The following code demonstrates how to use

Seaborn to generate a correlation heat map with automatic annotation and a `'coolwarm'` color theme:

```
import seaborn as sns

sns.heatmap(data.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

This code would produce a heat map showing the correlation matrix of the dataset, with annotations for each cell and a color gradient indicating the strength and direction of correlations. However, with a small dataset, this visualization may not be informative.

Instead, a **pair plot** can be created using Seaborn, which is suitable for the available data. A pair plot visualizes the pairwise relationships between selected variables, displaying distributions along the diagonal and scatter plots for each variable pair. The following code creates a pair plot for the columns 'Age', 'Income', and 'Gender':

```
sns.pairplot(data[['Age', 'Income', 'Gender']])
plt.show()
```

This code generates a grid of plots showing the distribution of 'Age' and 'Income', as well as scatter plots of 'Income' versus 'Age', with points colored by 'Gender' if the data supports it. The pair plot provides an overview of how these variables relate to each other in the dataset.

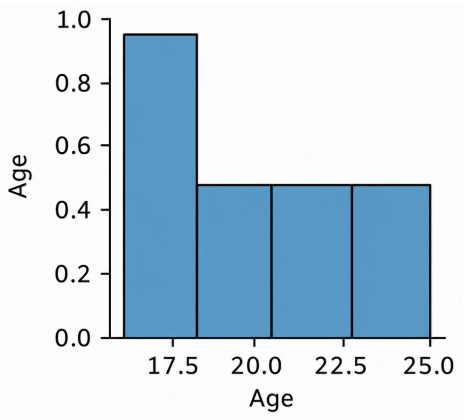


Figure 1.12: Pair plot displaying pairwise relationships between age, income, and gender

Descriptive statistics

Descriptive statistics are of two types: basic descriptive statistics and interpretive statistics.

For **basic descriptive statistics**, it is possible to compute values such as mean, median, mode, and standard deviation to understand the distribution of data. To obtain a summary of these statistics, the `describe()` function can be used on the dataset. This function provides information including mean, median, mode, standard deviation, minimum, and maximum values, as well as other descriptive metrics. Running `data.describe()` will output this summary, offering a comprehensive description of the dataset.

Interpretive statistics involves understanding the meaning of these computed values. The mean, median, and mode indicate the central tendency of the data. The standard deviation measures the spread, showing how the data is distributed. Percentiles, such as the minimum, maximum, 25th, 50th, and 75th percentiles, provide further insight into the distribution of values within the dataset.

Hands-on project: Exploratory data analysis of a real dataset

For this hands-on project, our task is to perform exploratory data analysis on the Titanic dataset to understand the survival rate across various features, including age, gender, and class. The steps for this project are as follows:

- Load the *Titanic dataset* using Pandas
- Clean the data by handling missing values, such as missing age values
- Explore the dataset
- Visualize the data

Let's start by accessing the dataset. Visit `https://www.kaggle.com/datasets`, and search for Titanic. The one we are using here is `https://www.kaggle.com/datasets/heptapod/titanic`. Extract the file and move it to Jupyter Notebook. Rename it to `titanic.csv`. Create a new Notebook for the hands-on project.

Tip

Please note that the dataset consists of a column named `2urvived`, which should have been `Survived`. Please rename the column to `Survived` and follow the steps as mentioned.

The initial requirement is to import the necessary libraries as follows:

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

The first step in the analysis is to load the Titanic dataset. This is accomplished by reading the CSV file named `titanic.csv` using pandas:

```
data = pd.read_csv('titanic.csv')
```

The next step is to handle missing values in the dataset. The focus is on the `'Age'` column. All missing values in the `'Age'` column are replaced with the mean age. This is done by obtaining all the age data, calculating the mean, and filling the missing values with this mean. The operation is performed in place:

```
data.fillna({'Age':data['Age'].mean()}, inplace=True)
```

The analysis then proceeds to examine the survival rate. The following code prints the normalized value counts for the `'Survived'` column, which provides the survival rates:

```
print(data['Survived'].value_counts(normalize=True))
```

The next step is to group the data by gender and passenger class to analyze survival rates across these categories. The code groups the data by `'Sex'` and `'Pclass'`, then calculates the mean of the 'survived' column for each group:

```
print(data.groupby(['Sex', 'Pclass'])['Survived'].mean())
```

The next task is to visualize the data. A bar plot is created using Seaborn to show the survival rate by gender. The x-axis represents gender, and the y-axis represents the survival rate:

```
sns.barplot(x='Sex', y='Survived', data=data)
plt.title('Survival Rate by Gender')
plt.show()
```

The output generated is as follows:

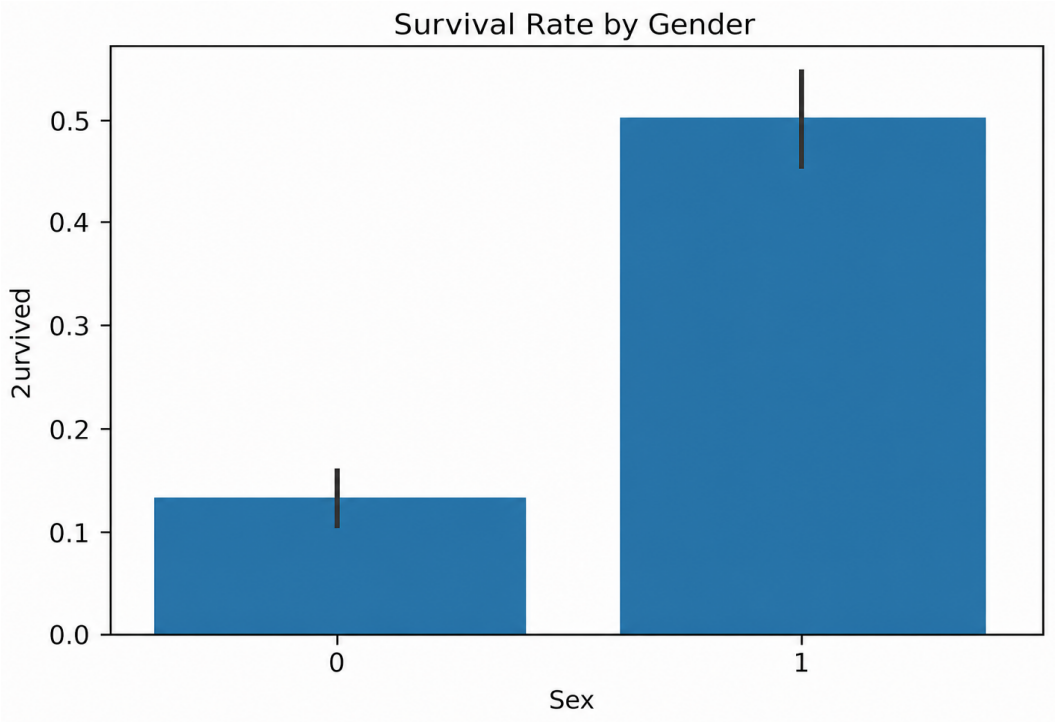


Figure 1.13: Bar chart displaying survival rates by gender for Titanic passengers

The plot in *Figure 1.13* displays the survival rate by gender, with the x-axis labeled as "Sex" and the y-axis labeled as "Survived". The bar for females is around 0.5-0.6 and the bar for males is about 0.15.

The output printed is:

```
Survived
0    0.636364
1    0.363636
Name: proportion, dtype: float64
sex     Pclass
female   1         0.968085
         2         0.921053
         3         0.500000
male     1         0.368852
         2         0.157407
         3         0.135447
Name: Survived, dtype: float64
```

The printed output first displays the survival data, showing the proportion of survivors and non-survivors in the dataset. This is followed by a breakdown of survival rates by gender and passenger class, indicating the number and ratio of females and males, as well as their respective survival rates across different classes.

The bar chart visualizes the survival rate by gender. In this particular dataset, the plot indicates that all females survived, while it appears that no males survived according to the data shown.

The code and output in the notebook cell demonstrate the sequence of steps taken: importing the necessary libraries, loading the Titanic dataset, filling missing age values with the mean, printing normalized survival counts, grouping by gender and passenger class to calculate survival means, and finally plotting the survival rate by gender.

Summary

Python for AI and Data Science Foundations introduces the essential building blocks of Python programming, from basic syntax, variables, and control flow to functions and core data structures such as lists and dictionaries. The chapter guides through practical steps for setting up Python and Jupyter Notebooks and progresses to foundational libraries like NumPy and Pandas for array operations, data manipulation, and handling missing values. Key concepts in data transformation, feature engineering, and visualization using Matplotlib and Seaborn are demonstrated, culminating in hands-on projects that apply these techniques to real-world datasets, including exploratory analysis of the Titanic dataset.

In the next chapter, *Chapter 2, Machine Learning and Classification Models*, we will move from these foundations into machine learning, building your first regression and classification models.

2

Machine Learning and Classification Models

Machine learning and classification models have become central to modern data analysis, offering a systematic approach to uncovering insights and making informed predictions from complex datasets. By combining foundational concepts, data preparation techniques, model building, and evaluation strategies, this field enables the extraction of meaningful patterns and relationships from raw information. As a result, machine learning empowers practitioners to develop predictive tools that support decision-making across diverse domains, from housing markets to healthcare and beyond.

The following topics will be covered in this chapter:

- Machine learning and its types
- Supervised learning and dataset preparation
- Feature scaling and regularization
- Hands-on project: Predicting house prices using linear regression
- Understanding classification models in ML
- Building a logistic regression classifier
- Hands-on project: Spam detection using logistic regression

By the end of this chapter, we will:

- Understand the basic concepts of machine learning
- Differentiate between supervised and unsupervised learning
- Know how to perform a train-test split
- Build a linear regression model for predicting continuous variables

- Evaluate model performance using metrics like Mean Squared Error (MSE) and R-squared
- Understand the basics of classification and how it differs from regression
- Build and train a logistic regression classifier
- Understand and compute performance metrics such as accuracy, precision, recall, F1-score, and confusion matrix
- Visualize the decision boundary of a classification model
- Apply classification algorithms to a real-world dataset

Technical requirements

To follow along with the examples and hands-on projects in this chapter, you will need the following:

- Python 3.8 or above
- Jupyter Notebook
- The following Python libraries are installed via pip:
    - `numpy`
    - `pandas`
    - `matplotlib`
    - `seaborn`
    - `scikit-learn`

You can install these libraries using the following command:

```
pip install numpy pandas matplotlib seaborn scikit-learn
```

Additionally, you will need the following datasets:

- California housing dataset from Kaggle (`https://www.kaggle.com/datasets/camnugent/california-housing-prices`)
- Spam detection dataset from Kaggle (`https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv`)

A free Kaggle account is required to download these datasets.

All code examples in this chapter assume that you have Jupyter Notebook installed and running, with the required libraries available in your Python environment.

Machine learning and its types

Machine learning (ML) is a branch of artificial intelligence (AI) that enables systems to learn from data, identify patterns, and make decisions with minimal human intervention. The main types of machine learning include supervised learning and unsupervised learning:

- In **supervised learning**, the model is trained on labeled data where the correct output is provided. Examples of supervised learning tasks include regression and classification.
- In **unsupervised learning**, the model is trained on data that is not labeled and attempts to identify patterns on its own. Examples of unsupervised learning tasks include clustering and dimensionality reduction.

Linear regression is a supervised learning algorithm used to predict a continuous variable, known as the target, based on one or more input features, which are independent variables. The focus in this chapter is primarily on linear regression.

Supervised learning and dataset preparation

For the data set overview, a data set is selected where the goal is to predict house prices based on features such as the number of rooms, size of the house, and age. The example data set provides features like the number of rooms, house size in square feet, and the age of the house. The target variable is the house price.

Several housing price datasets are available for this purpose on Kaggle. For example, the *California Housing Prices* dataset (`https://www.kaggle.com/datasets/camnugent/california-housing-prices`) is a popular choice. This dataset includes features such as longitude, latitude, the age of the house, and the total number of rooms available. While it does not contain the exact price, it provides the median house value.

Other similar datasets include the *Housing Market and Prices* dataset, which contains attributes such as bedroom count, net square meter, center distance, metro distance, floor, age, and price. These datasets provide a range of values suitable for supervised learning tasks.

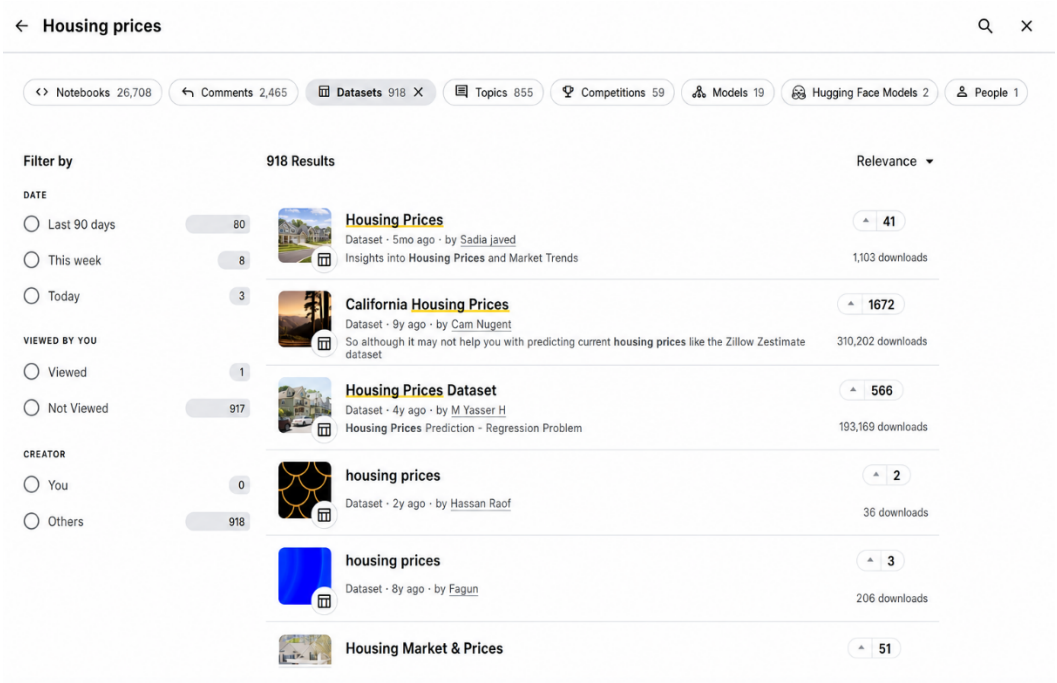


Figure 2.1: Housing prices datasets on Kaggle website

Any of these datasets can be used. To proceed with loading and preparing a dataset, the process involves splitting the data into features and target. The following code imports pandas and assigns it the alias `pd`:

```
import pandas as pd
```

Next, the data set is loaded using the pandas library. The following code demonstrates how to read a CSV file named `house_prices.csv` into a DataFrame:

```
data = pd.read_csv('housing.csv')
```

After loading the data, the next step is to select the relevant features, also known as independent variables. In this example, the features chosen are `'Rooms'`, `'Size'`, and `'Age'`. These are extracted from the DataFrame as follows:

```
X = data[['total_rooms', 'households', 'housing_median_age']]
```

The target variable, which is the value to be predicted, is the `'Price'` column:

```
y = data['median_house_value']
```

This process prepares the data by splitting it into features and target, which is a necessary step before training a machine learning model.

Splitting data into training and test sets

To evaluate the performance of a machine learning model, it is important to divide the data set into a training set and a test set. The training set is used to train the model, while the test set is used to evaluate its performance. The scikit-learn library provides a function called `train_test_split` for this purpose. This function is available in the `sklearn.model_selection` module. The following code imports the required function:

```
from sklearn.model_selection import train_test_split
```

To split the data, the features and target are passed to the function, along with parameters specifying the proportion of the data to be used for testing and a random seed for reproducibility. In this example, 80 percent of the data is used for training and 20 percent for testing, with a random state of 42:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

This code divides the data into four sets: `X_train`, `X_test`, `y_train`, and `y_test`, which are used for training and testing the model.

Building a linear regression model

The next step is to build a linear regression model. The goal of linear regression is to find a linear relationship between the input features and the target variable. The formula for linear regression is:

$$Y = w1 * \mathrm{x1} + \mathrm{w2} * x2 + \ldots + wn * xn + b$$

In this formula:

- `y` is the predicted value (for example, house price)
- `x` represents the input features (such as rooms, size, and age)
- `w` are the weights or parameters learned by the model
- `b` is the bias term

To build a linear regression model using scikit-learn, the `LinearRegression` class from the `sklearn.linear_model` module is used. The following code imports the class:

```
from sklearn.linear_model import LinearRegression
```

The model is then initialized:

```
model = LinearRegression()
```

To train the model, the `fit()` method is called with the training features and target:

```
model.fit(X_train, y_train)
```

That builds and trains the model for us. The next step is to make predictions.

Making predictions

Once the model is trained, predictions can be made on the **test set**. The following code uses the trained model to predict house prices for the test data:

```
y_pred = model.predict(X_test)
print(y_pred)
```

The predicted values can then be printed and compared with the actual target values from the test set to assess how well the model performs.

Evaluating model performance

To evaluate the performance of a regression model, metrics such as **mean squared error (MSE)** and **R-squared (R^2)** are commonly used.

Mean squared error measures the average of the squares of the errors, which are the differences between the actual and predicted values. The formula for mean squared error is:

$$MSE = (1/n) * \Sigma(actual - predicted)^2$$

Where `n` is the number of samples.

R-squared, on the other hand, indicates how well the independent variables explain the variance in the target variable. An R-squared value close to 1 indicates that the model explains a significant portion of the variance.

To calculate these metrics using scikit-learn, the following code imports the necessary functions:

```
from sklearn.metrics import mean_squared_error, r2_score
```

The mean squared error is calculated by passing the actual and predicted values:

```
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error is {mse}")
```

The R-squared value is calculated as follows:

```
r2 = r2_score(y_test, y_pred)
print(f"R-squared: {r2}")
```

Lower MSE indicates a model that makes better predictions, while a higher R-squared value, closer to 1, indicates that the model explains a significant portion of the variance in the target variable.

Feature scaling and regularization

Some machine learning models benefit from **feature scaling**, which ensures that all features are on a similar scale. This can improve the convergence of gradient-based algorithms such as linear regression. Scikit-learn's `StandardScaler` can be used to standardize features by removing the mean and scaling to unit variance. The following code imports the scaler:

```
from sklearn.preprocessing import StandardScaler
```

The scaler is initialized with the following:

```
scaler = StandardScaler()
```

The training data is then fitted and transformed:

```
X_train_scaled = scaler.fit_transform(X_train)
```

The test data is transformed using the same scaler:

```
X_test_scaled = scaler.transform(X_test)
```

This process standardizes the features, which can improve model performance.

Regularization helps prevent overfitting by penalizing large coefficients. Common techniques include **ridge regression** and **lasso regression**. To use ridge regression, the `Ridge` class is imported from `sklearn.linear_model`:

```
from sklearn.linear_model import Ridge
```

The ridge regression model is initialized with a regularization parameter `alpha`:

```
ridge_model = Ridge(alpha=1.0)
```

The model is then trained using the scaled training data:

```
ridge_model.fit(X_train_scaled, y_train)
```

Lasso regression is similar but uses the `Lasso` class. The strength of regularization can be adjusted by changing the `alpha` value. For example, to use a weaker regularization:

```
from sklearn.linear_model import Lasso
lasso_model = Lasso(alpha=0.1)
lasso_model.fit(X_train_scaled, y_train)
```

Both ridge and lasso regression can be used to create models and fit them with the training data by calling the appropriate function.

Hands-on project: Predicting house prices using linear regression

For this hands-on project, the task is to build a linear regression model to predict house prices based on input features such as number of rooms, crime rate, and proximity to city centers. The steps followed are:

1. Load the data. The California housing dataset is loaded, which contains features such as population, median income, and house age. The target is the median house price.
2. Perform exploratory data analysis (EDA). The data is inspected for missing values and general statistics, and feature distributions are visualized using histograms.
3. Prepare features and targets. The target variable is `MedHouseValue`, which represents the median house value, and the remaining columns are treated as features.
4. Perform a train-test split. The dataset is divided into training and test sets in order to evaluate the model's performance on data that has not been seen during training.

5. Train the model by fitting a linear regression model using the training set.
6. Evaluate the model. This involves calculating the **mean squared error** (MSE) and **R-squared** to measure how well the model performs.
7. Visualize the results of the linear regression model. Actual house prices are plotted against the predicted house prices to provide a visual assessment of how well the model's predictions align with the true values.
8. Display the model coefficients of the features used in the model. By examining these coefficients, it becomes possible to understand the relative importance of each feature in predicting house prices. The magnitude and sign of each coefficient indicate how changes in the corresponding feature affect the predicted house price.

Let's perform these steps.

Importing libraries

Create a new Python 3 notebook for this project. The following code imports the necessary libraries for data handling, model building, evaluation, and visualization:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import fetch_california_housing
import matplotlib.pyplot as plt
```

These imports include the following libraries and their purposes:

- Pandas and NumPy for data manipulation
- `train_test_split` from scikit-learn for splitting the dataset
- `LinearRegression` for building the model
- `mean_squared_error` and `r2_score` for evaluating model performance
- `fetch_california_housing` for loading the dataset
- `matplotlib.pyplot` for plotting graphs

Next, let's load the data.

Loading and inspecting the California housing dataset

The next step is to load the California housing dataset. The following code fetches the dataset and loads it as a DataFrame:

```
california = fetch_california_housing(as_frame=True)
df = california.frame
```

To display the first few rows of the dataset and verify that the data has loaded correctly, the following command is used:

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

Figure 2.2 displays the first five rows of the California housing dataset, including the columns `MedInc`, `HouseAge`, `AveRooms`, `AveBedrms`, `Population`, `AveOccup`, `Latitude`, `Longitude`, and `MedHouseVal`:

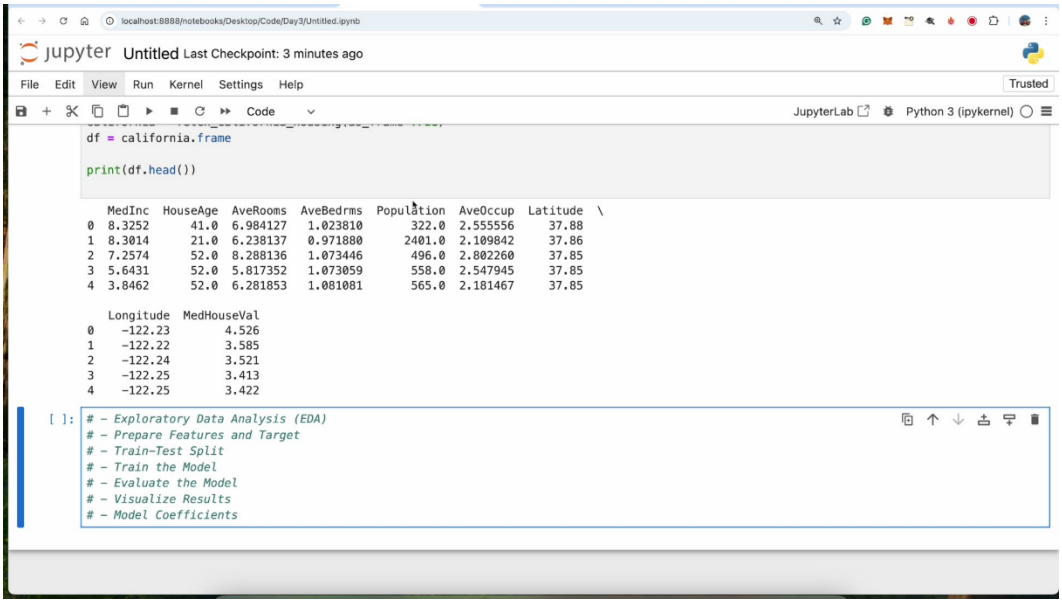


Figure 2.2: First five rows of the California housing dataset with columns

Next step is to do some basic exploratory data analysis.

Perform exploratory data analysis

To perform basic exploratory data analysis, the following steps are executed:

1. Check for missing values in the dataset
2. Display basic statistics for each column
3. Plot histograms to visualize the distribution of each feature

The code below checks for missing values, displays descriptive statistics, and plots histograms for the features:

```
print(df.isnull().sum())
print(df.describe())
df.hist(bins=20, figsize=(10, 8))
plt.show()
```

The output confirms that there are no missing values in any of the columns:

```
MedInc          0
HouseAge        0
AveRooms        0
AveBedrms       0
Population      0
AveOccup        0
Latitude        0
Longitude       0
MedHouseVal     0
dtype: int64
```

Figure 2.3: Output of the isnull().sum() command showing the count of missing values for each column

The next output displays descriptive statistics for each column, including count, mean, standard deviation, minimum, maximum, and quartiles:

```
              MedInc      HouseAge      AveRooms     AveBedrms    Population  \
count  20640.000000  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.870671     28.639486      5.429000      1.096675   1425.476744
std        1.899822     12.585558      2.474173      0.473911   1132.462122
min        0.499900      1.000000      0.846154      0.333333      3.000000
25%        2.563400     18.000000      4.440716      1.006079    787.000000
50%        3.534800     29.000000      5.229129      1.048780   1166.000000
75%        4.743250     37.000000      6.052381      1.099526   1725.000000
max       15.000100     52.000000    141.909091     34.066667  35682.000000

           AveOccup      Latitude     Longitude   MedHouseVal
count  20640.000000  20640.000000  20640.000000  20640.000000
mean       3.070655     35.631861   -119.569704      2.068558
std       10.386050      2.135952      2.003532      1.153956
min        0.692308     32.540000   -124.350000      0.149990
25%        2.429741     33.930000   -121.800000      1.196000
50%        2.818116     34.260000   -118.490000      1.797000
75%        3.282261     37.710000   -118.010000      2.647250
max     1243.333333     41.950000   -114.310000      5.000010
```

Figure 2.4: Descriptive statistics output from the describe() method

Histograms are plotted for each feature to visualize their distributions. The plots show the spread of values for features such as `MedInc, HouseAge, AveRooms, AveBedrms, Population, AveOccup, Latitude, Longitude, and MedHouseVal`. For example, the latitude and longitude distributions reflect the geographic spread across California, while features like average number of bedrooms and median income show their own characteristic distributions.

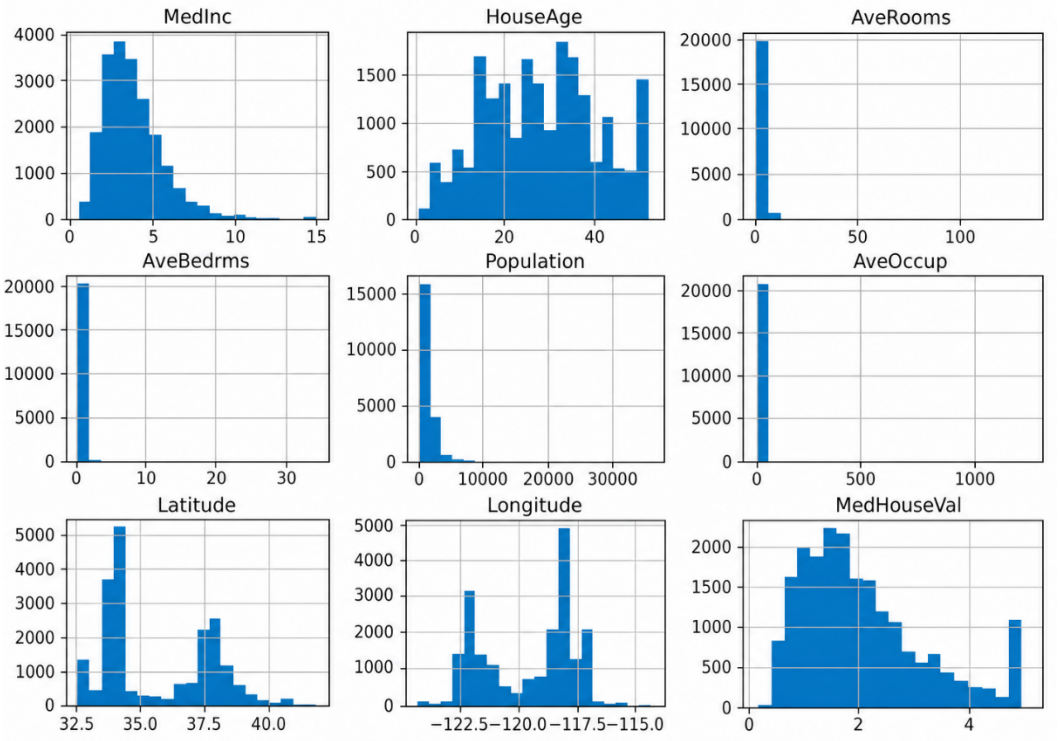


Figure 2.5: Histograms showing the distribution of features in the California housing dataset

Preparing features and target variables

The next step is to prepare the features and the target variable for model training. The features are selected by dropping the target column, `MedHouseVal`, from the DataFrame. The target variable is set as the `MedHouseVal` column. The following code prepares the features and target:

```
X = df.drop('MedHouseVal', axis=1)
y = df['MedHouseVal']
```

Creating the train-test split

To evaluate the model's performance, the data is split into training and testing sets. The split assigns 80 percent of the data to the training set and 20 percent to the testing set, with a fixed random state for reproducibility.

The following code performs the train-test split:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

To verify the split, the shapes of the training and testing sets are printed:

```
print(f"Training set shape: {X_train.shape}")
print(f"Testing set shape: {X_test.shape}")
```

The output is:

```
Training set shape: (16512, 8)
Testing set shape: (4128, 8)
```

Training the linear regression model

A linear regression model is initialized and trained on the training data. The following code creates the model and fits it to the training set:

```
model = LinearRegression()
model.fit(X_train, y_train)
```

Making predictions and evaluating the model

Predictions are made on the test set using the trained model:

```
y_pred = model.predict(X_test)
```

To evaluate the model, the MSE and R-squared score are calculated for the test data:

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R-Squared: {r2:.4f}")
```

The output is:

```
Mean Squared Error (MSE): 0.5559
R-Squared: 0.5758
```

Next, let's visualize the data.

Visualizing actual vs. predicted values

A scatter plot is created to visualize the relationship between the actual and predicted median house values. The following code generates the plot:

```
plt.figure(figsize=(10, 6))
plt.scatter(y_test, y_pred, alpha=0.6, color='blue')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=3)
plt.title('Actual vs Predicted House Prices')
plt.xlabel('Actual Prices (in $100,000s)')
plt.ylabel('Predicted Prices (in $100,000s)')
plt.show()
```

In this code, the plot is given a title, "Actual vs Predicted House Prices," to clearly indicate the comparison being visualized. The x-axis label is set to "Actual Prices (in $100,000s)" to reflect the units of the data, and the y-axis label is set to "Predicted Prices (in $100,000s)" for consistency. The plot is then displayed, showing the predicted prices versus the actual prices.

The plot in *Figure 2.6* shows predicted house prices on the y-axis and actual house prices on the x-axis, both in units of $100,000s. The blue points represent individual predictions, and the red dashed line indicates the ideal fit where predicted values equal actual values.

This plot allows for visual assessment of how closely the predicted values match the actual values and helps identify any systematic deviations or patterns in the predictions:

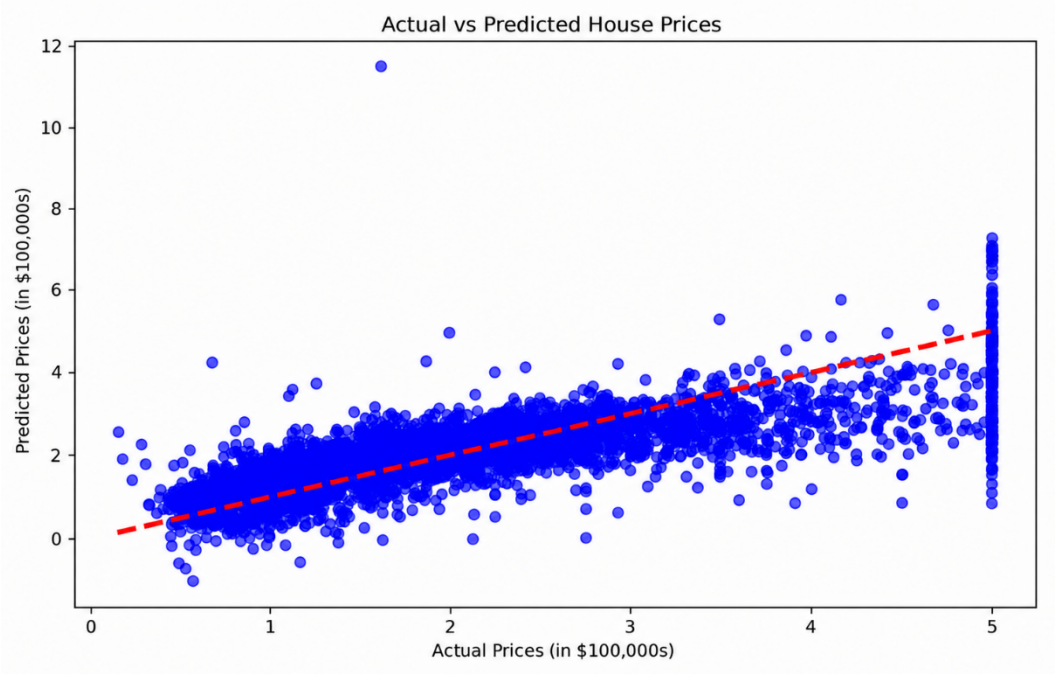


Figure 2.6: Scatter plot comparing actual and predicted house prices, with the red dashed line indicating perfect predictions

The data points are closely aligned, especially at lower price ranges, with some deviation at higher values, but overall the results are satisfactory.

Displaying model coefficients

The next step is to display the **coefficients** of the linear regression model. This is accomplished by creating a pandas DataFrame containing the model's coefficients and the corresponding feature names. The following code constructs the DataFrame and prints the coefficients:

```
coefficients = pd.DataFrame(model.coef_, X.columns, columns=['Coefficient'])
print(coefficients)
```

The output displays the coefficients for each feature in the model:

```
             Coefficient
MedInc          0.448675
HouseAge        0.009724
AveRooms       -0.123323
AveBedrms       0.783145
```

```
Population   -0.000002
AveOccup     -0.003526
Latitude     -0.419792
```

This table shows the contribution of each feature to the predicted house price according to the linear model.

This concludes the hands-on demonstration of a complete pipeline for predicting house prices using linear regression. The process covered data pre-processing, model training, evaluation, and visualization. Further experimentation can be conducted by applying regularization techniques such as ridge or lasso, or by trying different algorithms to compare their performance.

Understanding classification models in ML

Classification is a type of supervised learning where the goal is to categorize data into predefined labels. Our objective for this section is to build a logistic regression model to classify whether an email is spam or not, and to explore key concepts of classification, including performance metrics such as accuracy, precision, recall, and confusion matrix.

What is classification?

Classification is the task of predicting the class or category of an input based on its features. Unlike regression, where the output is continuous, in classification the output is categorical. For example, determining whether an email is spam or not is a binary classification problem. Other examples include classifying an image as either a cat, dog, or bird, which is referred to as multi-class classification, or predicting whether a customer will churn, which is another binary classification example.

Logistic regression for classification

Logistic regression is a supervised learning algorithm used for binary classification problems. It predicts the probability of an instance belonging to a particular class using the logistic or sigmoid function. The logistic function is defined as follows:

$$P(y = 1) = \frac{1}{1 + e^{-z}}$$

Where $z = w * x + b$. Here, b represents the bias term. This is the mathematical definition of logistic regression.

The dataset that we will use for demonstration is the spam detection dataset, which is used to classify emails as either scam (represented by 1) or not scam (represented by 0), based on features such as the frequency of certain words.

Building a logistic regression classifier

The process of building a logistic regression classifier involves three steps:

1. Loading and preparing the data set. In this step, the data set is loaded and split into training and testing sets.
2. Training the logistic regression model. The model is trained using scikit-learn.
3. Making predictions. The trained model is used to predict whether the emails in the test set are spam or not.

This process will be covered in a hands-on exercise. But, to do a warm-up exercise, let's practice it. The following code demonstrates how to load and prepare the dataset for logistic regression classification using scikit-learn:

```
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.read_csv("spam_data.csv")
X = data.drop("Spam", axis=1)
y = data["Spam"]
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

Next, the logistic regression model is initialized and trained using the training data:

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
```

After training, predictions are made on the test set as follows:

```
y_pred = model.predict(X_test)
print(y_pred)
```

This prints the predicted values for the test set.

Evaluating the classification model

Evaluation of the classification model involves several metrics, including confusion matrix, accuracy, precision, recall, and the F1 score. Let's begin with the confusion matrix.

Confusion matrix

The **confusion matrix** is a table that summarizes the performance of a classification algorithm by comparing actual and predicted values. The entries in the confusion matrix include:

- True positives (TP), which are correctly predicted as spam
- True negatives (TN), which are correctly predicted as not spam
- False positives (FP), which are incorrectly predicted as spam (false alarms)
- False negatives (FN), which are incorrectly predicted as not spam

The following code demonstrates how to compute the confusion matrix using scikit-learn:

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
```

Printing the confusion matrix in a program will display the confusion matrix for the dataset.

Accuracy

Accuracy is the ratio of correctly predicted observations to the total number of observations. The formula for accuracy is:

$$Accuracy = (TP + TN) / (TP + TN + FP + FN)$$

This formula calculates the proportion of correct predictions made by the classification model out of all predictions. In scikit-learn, accuracy can be computed using the `accuracy_score` function from the metrics module. The following code demonstrates how to calculate and print the accuracy:

```
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

Precision and recall

Precision is the ratio of correctly predicted positive observations to the total predicted positives. In the context of spam detection, it measures how many emails predicted as spam are actually spam. The formula for precision is:

$$Precision = TP / (TP + FP)$$

Recall, also known as sensitivity, is the ratio of correctly predicted positive observations to all observations in the actual positive class. For spam detection, it measures how many actual spam emails were correctly identified. The formula for recall is:

$$Recall = TP/(TP + FN)$$

To compute precision and recall using scikit-learn, the following code can be used:

```
from sklearn.metrics import precision_score, recall_score

precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
print(precision)
print(recall)
```

These metrics provide insight into the model's ability to correctly identify positive cases and avoid false alarms.

F1 score

The **F1 score** is the harmonic mean of precision and recall. It provides a balance between precision and recall, especially when they are not equally important. The formula for the F1 score is:

$$F1\ Score = 2 * (Precision * Recall) / (Precision + Recall)$$

The F1 score can be computed using scikit-learn as follows:

```
from sklearn.metrics import f1_score

f1 = f1_score(y_test, y_pred)
print(f1)
```

Visualizing the decision boundary

Visualizing the decision boundary helps in understanding how the model separates different classes, such as spam versus not spam. When working with a 2D feature set, the dataset can be reduced to two features for easier visualization. To visualize the decision boundary, the following steps are performed:

1. Import the necessary libraries for plotting and numerical operations:

```
import matplotlib.pyplot as plt
import numpy as np
```

2. Define a mesh grid for plotting the decision boundary. The step size `h` determines the granularity of the grid:

```
h = 0.02
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
```

Here, `X_test[:, 0].min()` and `X_test[:, 0].max()` extract the minimum and maximum values from the first feature of the test data. Additionally, the `X_test[:, 0]` selects all the rows of the first column (0), representing the one feature in feature space.

`x_min` and `x_max` define the range for the x-axis, and `y_min` and `y_max` define the range for the y-axis. The mesh grid divides the feature space into small squares, and the step size controls how fine or coarse the grid will be.

`np.arange(x_min, x_max, h)` generates a 1D array of equally spaced values between `x_min` and `x_max` h having spacing of `h`.

3. Predict the class for each point on the grid:

```
Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

The `ravel()` method flattens the 2D arrays into 1D arrays, and `np.c_` concatenates them column-wise to create a 2D array of input points for prediction. The predictions are then reshaped back to the original grid shape for visualization.

4. Plot the decision boundary and the test data points:

```
plt.contourf(xx, yy, Z, alpha=0.8)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolors='k',
marker='o', s=100)
plt.show()
```

The `contour()` function creates a filled contour plot to visualize the decision boundary, with the `alpha` parameter controlling transparency. The `scatter()` function plots the actual test data points, coloring them based on their class labels, adding black edges, using circular markers, and setting the marker size for visibility.

Hands-on project: Spam detection using logistic regression

The hands-on project involves building a logistic regression classifier to predict whether an email is spam or not, based on features such as word frequency and email length. The steps followed in this project are:

1. Load the dataset and split it into training and testing sets
2. Train the logistic regression model to classify emails as spam or not spam
3. Evaluate the model using accuracy, confusion matrix, precision, recall, and F1 score
4. Visualize the confusion matrix using Seaborn's heatmap

The dataset for this project can be found on Kaggle on this link `https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv`. After downloading and unzipping the dataset, place the CSV file in the same folder as the project code.

Importing libraries

The following code demonstrates the setup and initial steps:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
```

Loading and inspecting data

Load the spam detection dataset and inspect its structure to identify which columns are numeric and which are non-numeric:

```
data = pd.read_csv("emails.csv")
print(data.head())
print(data.dtypes)
```

This output will be the first few rows of the dataset and the data type of each column. You will notice that some columns contain non-numeric data such as email IDs (for example, "Email 3165"), which cannot be used directly for model training.

Preparing the features and target

To prepare the data, drop the target column and any non-numeric columns. Keep only the numeric features for training:

```
# Drop the target column
X = data.drop(columns=["Prediction"])

# Keep only the numeric features
X = X.select_dtypes(include="number")

# Set the target variable
y = data["Prediction"]
```

The `select_dtypes(include="number")` method filters the features to include only numeric columns, which is required for logistic regression.

Creating the train-test split

Divide the data into training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
```

Training the logistic regression model

Initialize and train the logistic regression model:

```
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
```

The `max_iter=1000` parameter ensures the model has sufficient iterations to converge during training.

Making predictions

To generate predictions on the test set, the following code assigns the predicted values to a variable:

```
y_pred = model.predict(X_test)
```

Evaluating the model

Next, the model is evaluated using several metrics. The following code calculates the accuracy, precision, recall, and F1 score by comparing the true labels with the predicted labels:

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

To display these evaluation metrics in a readable format, the following code prints each value with a descriptive label:

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1_score: {f1}")
```

The output of the code is as follows:

```
  Email No.  the  to  ect  and  for  of    a  you  hou  ...  connevey  jay  \
0   Email 1    0   0    1    0    0   0    2    0    0  ...         0    0
1   Email 2    8  13   24    6    6   2  102    1   27  ...         0    0
2   Email 3    0   0    1    0    0   0    8    0    0  ...         0    0
3   Email 4    0   5   22    0    5   1   51    2   10  ...         0    0
4   Email 5    7   6   17    1    5   2   57    0    9  ...         0    0

   valued  lay  infrastructure  military  allowing  ff  dry  Prediction
0       0    0               0         0         0   0    0           0
1       0    0               0         0         0   1    0           0
2       0    0               0         0         0   0    0           0
3       0    0               0         0         0   0    0           0
4       0    0               0         0         0   1    0           0

[5 rows x 3002 columns]
Email No.        str
the            int64
to             int64
ect            int64
and            int64
               ...
military       int64
allowing       int64
ff             int64
dry            int64
Prediction     int64
Length: 3002, dtype: object
Accuracy: 0.9719806763285024
Precision: 0.9435215946843853
Recall: 0.9594594594594594
F1: 0.9514237855946399
```

Figure 2.7: Evaluation metrics and dataset overview for the spam detection classification model

The output of the code displays the calculated accuracy, precision, recall, and F1 score values, all of which are in the 90 percent range.

Visualizing the confusion matrix

The next step involves visualizing the confusion matrix. The confusion matrix is computed and then displayed as a heatmap using Seaborn. The following code performs these actions:

```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d')
plt.title('Confusion Matrix')
plt.show()
```

A warning message appears indicating that the total number of iterations reached the limit during logistic regression training, but the process completed and produced results. The

confusion matrix is shown, illustrating the counts of true positives, true negatives, false positives, and false negatives:

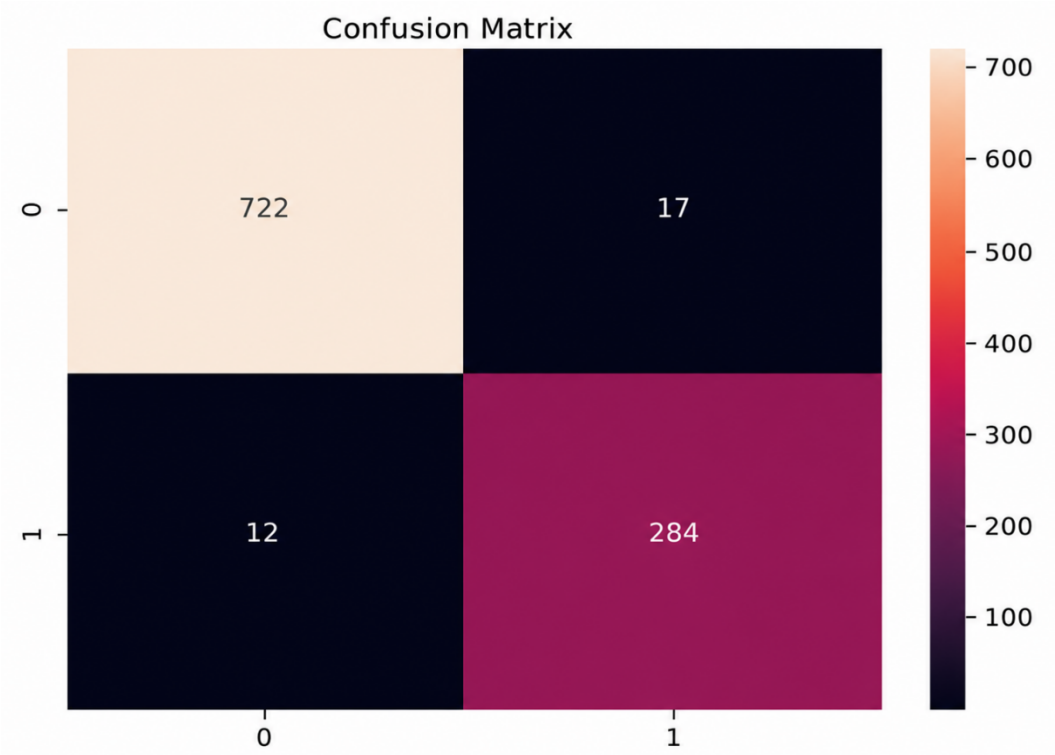


Figure 2.8: Confusion matrix heatmap showing the classification results for spam detection predictions

Summary

Machine learning and classification models encompass foundational concepts such as supervised learning, data preparation, and the process of splitting datasets for training and testing. The chapter details the implementation and evaluation of linear and logistic regression models, including techniques like feature scaling, regularization, and visualization of predictions and decision boundaries. Practical applications are demonstrated through hands-on projects, such as spam detection, while key performance metrics—accuracy, precision, recall, and F1 score—are used to assess model effectiveness.

In the next chapter, *Chapter 3, Neural Networks and Natural Language Processing*, you will move into neural networks and natural language processing, starting with the fundamentals of deep learning architectures.

3

Neural Networks and Natural Language Processing

Neural networks have become central to advances in artificial intelligence, enabling machines to interpret complex patterns in data such as images and language. Within the domain of natural language processing and computer vision, deep learning techniques allow for the automated extraction and classification of features from raw inputs, often surpassing traditional methods in accuracy and adaptability. By constructing and evaluating neural network models, it becomes possible to perform tasks like handwritten digit recognition, demonstrating the practical application of these concepts in real-world scenarios. The fundamental concepts of neural networks and deep learning are introduced through the construction of a simple feed-forward neural network for classifying handwritten digits using the MNIST dataset. This approach provides an initial hands-on experience with deep learning frameworks such as TensorFlow.

The following topics will be covered in this chapter:

- What is a neural network
- MNIST data overview
- Building a simple neural network for MNIST classification
- Evaluating the neural network
- Understanding activation functions: ReLU and Softmax
- Hands-on project: Hand-written digit classification using neural networks
- Introduction to natural language processing
- Sentiment analysis: Understanding text classification
- Using pre-trained models for NLP

- Building a sentiment analysis with TensorFlow
- Model evaluation and confusion matrix
- Hands-on project: Sentiment analysis using pre-trained NLP models
- Summary

By the end of this chapter, we will:

- Understand the structure and working of a basic neural network
- Learn about activation functions, layers, and backpropagation
- Build a simple neural network for image classification
- Train and evaluate the model on the MNIST dataset
- Become familiar with popular deep learning libraries such as TensorFlow
- Understand the basics of NLP and its importance in AI
- Learn text preprocessing techniques such as tokenization, stemming, and vectorization
- Build a sentiment analysis model using a pre-trained NLP model
- Evaluate the model using metrics such as accuracy and F1 score
- Understand the concept of transfer learning for NLP tasks

Technical requirements

To follow along with the examples and hands-on projects in this chapter, you will need the following:

- Python 3.8 or above
- Jupyter Notebook
- The following Python libraries installed via pip:
    - `tensorflow`
    - `numpy`
    - `matplotlib`
    - `scikit-learn`
    - `nltk`
    - `transformers`
    - `tensorflow-datasets`
    - `pillow`

- `torch`
- `accelerate`

You can install these libraries using the following command:

```
pip install tensorflow numpy matplotlib scikit-learn nltk transformers tensorflow-datasets pillow torch accelerate
```

Additionally, you will need to download NLTK data resources. After installing NLTK, run the following commands in a Python environment:

```
import nltk
http://nltk.download('punkt')
nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('omw-1.4')
```

All code examples in this chapter assume that you have Jupyter Notebook installed and running, with the required libraries available in your Python environment. Models and datasets will be downloaded automatically when first accessed.

> **Note**
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

What is a neural network?

A **neural network** is a series of connected layers of nodes, also called neurons, modeled to mimic the human brain. The structure of a neural network consists of three main components. The first component is the **input layer**, where the input data enters the network. The second component is the **hidden layers**, which are the layers situated between the input and output layers and are responsible for the computation within the network. The final component is the **output layer**, which provides the final result or prediction.

Several key concepts are central to the operation of neural networks:

- **Weights and biases** are parameters of the network that are learned during the training process

- **Activation functions** such as ReLU, sigmoid, and softmax determine the output of a neuron
- **Backpropagation** is a process used to update the weights by utilizing the gradient of the errors, which is also referred to as the loss

Introduction to deep learning frameworks

Two prominent frameworks widely used for deep learning are **TensorFlow** and **PyTorch**. TensorFlow is a high-level library that offers easy integration into production systems. In contrast, PyTorch is recognized for its dynamic computational graph and its Pythonic, flexible nature, making it popular for research applications. For this chapter, we use TensorFlow due to its beginner-friendly interface.

MNIST data overview

The **MNIST dataset** consists of 28 × 28 pixel grayscale images of handwritten digits ranging from 0 to 9. This dataset is a standard benchmark for image classification tasks in deep learning. The training set contains 60,000 images of digits, while the test set contains 10,000 images. The dataset provides a practical foundation for exploring image classification with neural networks.

To begin working with the MNIST dataset in TensorFlow, the following steps are required:

Install TensorFlow using the following command:

```
pip install tensorflow
```

Import the necessary libraries, including `matplotlib.pyplot` for visualization and the MNIST dataset from `tensorflow.keras.datasets`

Load the training and test data as tuples containing the images and their corresponding labels. This is how the code looks:

```
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

To visualize an example from the dataset, the following code displays the first image in the training set along with its label:

```
plt.imshow(x_train[0], cmap='gray')
plt.title(f"Label: {y_train[0]}")
plt.show()
```

This code selects the first image from the training set, displays it in grayscale, and sets the plot title to the corresponding digit label. The resulting plot shows a handwritten digit (such as digit 5) as labeled in the dataset.

The plot in *Figure 3.1* shows a grayscale image of a handwritten digit from the MNIST training set, with the title indicating its label:

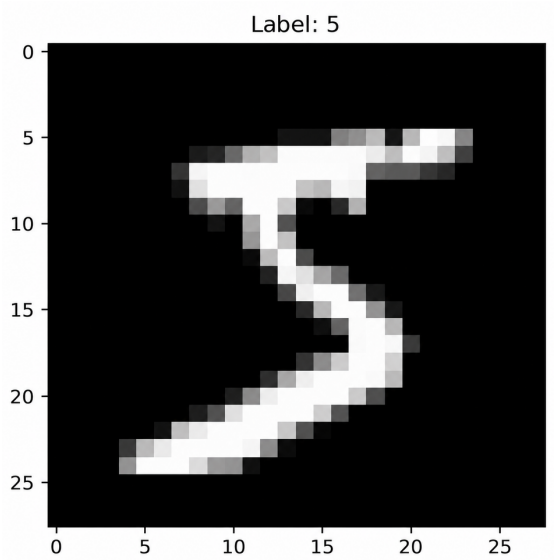


Figure 3.1: Grayscale image of a handwritten digit from the MNIST training dataset

Now that you have loaded and visualized the MNIST dataset, the next step is to preprocess this data and build a neural network to classify these handwritten digits.

Building a simple neural network for MNIST classification

The process of building a simple neural network for MNIST classification is divided into four steps. The first step is **data pre-processing**. Before feeding the data into the network, the pixel values must be reshaped and normalized.

Step 1: Data preprocessing

Before feeding the data into the network, the pixel values must be reshaped and normalized. First, the MNIST dataset must be loaded. The following code imports the necessary libraries and loads the data:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The data must be reshaped to samples of 28 x 28 x 1 for compatibility with TensorFlow. Additionally, the data type is converted to `float32`, as the default may be `float64`. The same reshaping and type conversion are applied to both the training and test datasets. The following code reshapes the MNIST data for use in TensorFlow:

```
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1).astype('float32')
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1).astype('float32')
```

This code ensures that each image has the correct shape and data type for input into a neural network model.

The next step in data pre-processing is to normalize the pixel values. The original pixel values in the MNIST dataset range from 0 to 255. To normalize these values to a range between 0 and 1, each pixel value is divided by 255.

> **Note**
>
> This normalization is necessary because the image color spectrum is defined up to 255, and scaling the values ensures that the neural network receives input data within a consistent range.

The following code normalizes the pixel values in both the training and test datasets:

```
x_train /= 255
x_test /= 255
```

This operation transforms all pixel values so that they fall within the interval from 0 to 1.

Step 2: Defining the neural network architecture

The second step is to define the neural network architecture. In this context, a simple **feedforward neural network** is constructed. The architecture consists of an input layer, one or more hidden layers with ReLU activation, and an output layer with Softmax activation for multi-class classification.

To build this model, the `Sequential` class is required from `tensorflow.keras.models`. The following code imports the necessary class for constructing the model:

```
from tensorflow.keras.models import Sequential
```

The `Sequential` model is used to stack layers in a linear fashion, which is suitable for building feedforward neural networks.

The next requirement is to import two specific layers from `tensorflow.keras.layers`: **Dense** and **Flatten**. These layers are essential for constructing the neural network model. The following code imports the `Dense` and `Flatten` layers:

```
from tensorflow.keras.layers import Dense, Flatten
```

To initialize the model, the `Sequential` model is used. The model is created by assigning it to a variable:

```
model = Sequential()
```

The first layer added to the model is the input layer, which uses the f**latten** layer. This layer transforms the 28 x 28 pixel images into a one-dimensional vector, making the data suitable for the subsequent dense layers. The input shape is specified as 28 x 28 x 1, corresponding to the dimensions of the MNIST images.

The following code adds the Flatten layer to the model and specifies the input shape:

```
model.add(Flatten(input_shape=(28, 28, 1)))
```

This step ensures that all input images are converted from their original two-dimensional format into a single vector, preparing them for further processing by the neural network.

The next step in constructing the neural network is to add the hidden and output layers. The hidden layer is configured with 128 neurons and uses the **ReLU activation function**. This is

accomplished by adding a Dense layer to the model, specifying the number of units and the activation function. The following code adds the hidden layer to the model:

```
model.add(Dense(128, activation='relu'))
```

This layer introduces nonlinearity into the model, allowing it to learn complex patterns in the data.

The output layer is then added, consisting of 10 neurons, one for each possible digit in the MNIST dataset. The **Softmax activation function** is used in the output layer to enable multi-class classification by converting the raw outputs into probabilities. The following code adds the output layer to the model:

```
model.add(Dense(10, activation='softmax'))
```

This completes the definition of the neural network architecture.

Step 3: Compiling the neural network model

The next step is to compile the model. Compilation specifies the optimizer, loss function, and evaluation metric to be used during training. For this task, `sparse categorical cross-entropy` is selected as the loss function, which is appropriate for multi-class classification problems. The Adam optimizer is chosen for its efficiency in achieving fast convergence. The evaluation metric selected is `accuracy`. The following code compiles the model with these specifications:

```
model.compile(
    optimizer='adam',
    loss = 'sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

This step finalizes the model configuration and prepares it for training.

Step 4: Training the neural network

The model is then trained on the preprocessed training data. Training is performed for 5 epochs with a batch size of 128. Additionally, 20 percent of the training data is set aside for validation, allowing monitoring of accuracy and loss on both the training and validation datasets. The following code trains the model:

```
model.fit(x_train, y_train, epochs=5, batch_size=128, validation_split=0.2)
```

This completes the four steps required to build a simple neural network for MNIST digit classification.

Evaluating the neural network

Evaluation of the neural network consists of two parts: model evaluation and making predictions.

Model evaluation

After training, the model is evaluated on the test dataset to assess its generalization performance. The evaluation returns the test loss and test accuracy, which indicate how well the model performs on unseen data.

The following code evaluates the model on the test data and prints the test accuracy:

```
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_accuracy}")
```

Making predictions

The trained model can be used to make predictions on new images. To demonstrate this, predictions are generated for the test dataset. The first test image and its predicted label are then visualized. The following code generates predictions and displays the first test image along with its predicted label:

```
predictions = model.predict(x_test)
plt.imshow(x_test[0].reshape(28, 28), cmap='gray')
plt.title(f"Predicted label is {predictions[0].argmax()}")
plt.show()
```

This process completes the evaluation of the neural network, including both model evaluation and prediction visualization.

Understanding activation functions: ReLU and softmax

Activation functions play a crucial role in neural networks. They introduce non-linearity to the network, enabling it to learn complex patterns and relationships in data. Without activation functions, even a deep neural network with many layers would simply perform a series of linear transformations, equivalent to a single linear operation. By introducing non-linearity through activation functions, the network can model intricate, non-linear relationships.

Different activation functions have different properties and are suited for different layers and tasks in a neural network.

We will look at two major activation functions in this section.

The **ReLU (Rectified Linear Unit)** is the most common activation function used and is defined as $f(x) = max(0, x)$. It introduces non-linearity into the model, enabling the network to learn complex relationships in the data.

The following code demonstrates how to implement and visualize the ReLU activation function using NumPy and Matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 100)
y = np.maximum(0, x)
plt.plot(x, y)
plt.title("ReLU Activation Function")
plt.show()
```

This code creates a plot showing the ReLU activation function, where negative values are zero and positive values pass through unchanged.

The **Softmax activation function** is used in the output layer for multi-class classification. It converts the raw outputs (logits) into probabilities, allowing the model to assign a probability to each class.

The following code demonstrates how to implement and visualize the Softmax activation function:

```
import numpy as np
import matplotlib.pyplot as plt

# Sample logits (raw neural network outputs)
logits = np.array([2.0, 1.0, 0.1])

# Softmax formula: e^x / sum(e^x)
exp_logits = np.exp(logits)
softmax_output = exp_logits / np.sum(exp_logits)

print(f"Logits: {logits}")
print(f"Softmax Output: {softmax_output}")
```

```
print(f"Sum of probabilities: {np.sum(softmax_output)}")

# Visualization of Softmax for multiple classes
classes = ['Class 1', 'Class 2', 'Class 3']
plt.bar(classes, softmax_output)
plt.ylabel('Probability')
plt.title('Softmax Activation Function Output')
plt.ylim(0, 1)
plt.show()
```

This code shows how Softmax converts raw logits (2.0, 1.0, 0.1) into probabilities that sum to 1.

```
Logits: [2.  1.  0.1]
Softmax Output: [0.65900114 0.24243297 0.09856589]
Sum of probabilities: 1.0
```

The bar chart visualizes the probability distribution across the three classes, making it clear how Softmax emphasizes the highest logit while still assigning non-zero probabilities to the others:

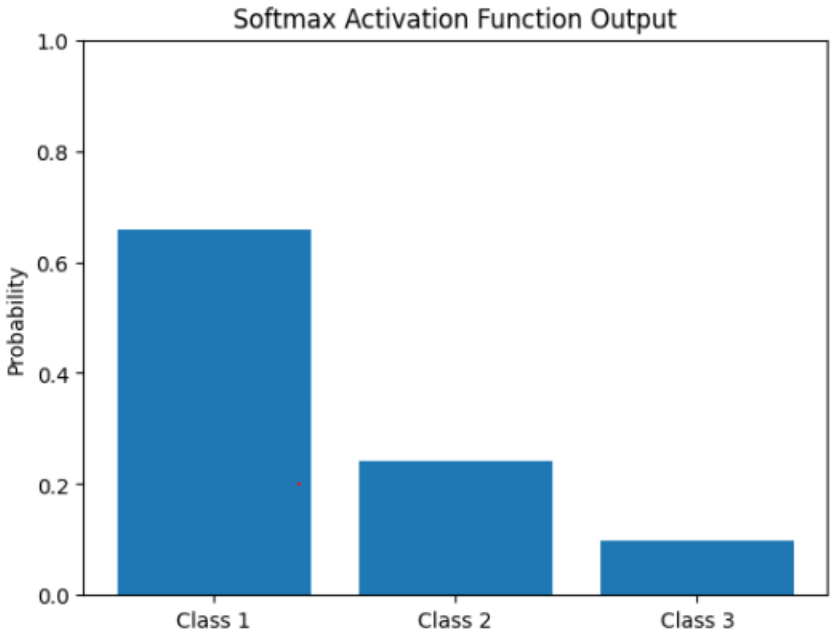


Figure 3.2: Probability distribution across three classes using softmax activation function

Hands-on project: Handwritten digit classification using neural networks

The hands-on project involves handwritten digit classification using neural networks. The dataset used is the MNIST dataset, and the objective is to build a neural network to classify handwritten digits. The network must include at least one hidden layer with ReLU activation and an output layer with Softmax activation for multi-class classification.

The steps to be followed are:

1. Preprocess the data by normalizing the pixel values and reshaping the input
2. Define the neural network architecture using at least one hidden layer and an output layer suitable for 10-class classification
3. Train the model and monitor the accuracy and loss during training
4. Evaluate the model on the test dataset, and visualize some predictions

To begin with, create a new Python 3 notebook. The first step is to import the necessary libraries. The following code imports the MNIST dataset from TensorFlow's Keras module:

```
from tensorflow.keras.datasets import mnist
```

The following code imports the necessary modules for building the neural network model, including the Sequential model, Dense and Flatten layers, and a utility for converting labels to categorical format:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
```

The next step is to load and preprocess the dataset. The following code loads the MNIST data and prepares it for training and testing:

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1).astype('float32') / 255
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1).astype('float32') / 255
```

The neural network is then constructed using the Sequential API. The model consists of a flatten layer to convert the input images into a one-dimensional array, followed by a dense

hidden layer with 128 units and an activation function, and a final Dense output layer with softmax activation for multi-class classification:

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28, 1)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

The model is compiled with the Adam optimizer, sparse categorical cross-entropy loss, and accuracy as the evaluation metric:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Training is performed using the following code, which fits the model to the training data for five epochs, with a batch size of 128 and a validation split of 20 percent:

```
model.fit(X_train, y_train, epochs=5, batch_size=128, validation_split=0.2)
```

After training, the model is evaluated on the test data to obtain the test loss and accuracy:

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Accuracy: {test_accuracy}")
```

Predictions are made on the test data, and the first test image is visualized along with its predicted label:

```
import numpy as np
import matplotlib.pyplot as plt

predictions = model.predict(X_test)
plt.imshow(X_test[0].reshape(28, 28), cmap='gray')
plt.title(f"Predicted Label: {np.argmax(predictions[0])}")
plt.show()
```

These steps include loading and preprocessing the data, building the neural network, compiling the model, training, evaluation, and making predictions on the test data. The output confirms that the model is able to correctly predict the label for a sample image from the test set:

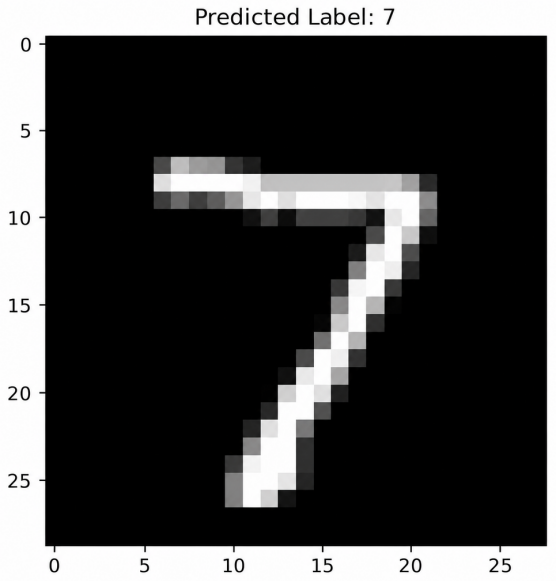


Figure 3.3: Handwritten digit prediction output showing the model correctly classifying the digit as 7

This hands-on project demonstrated how to build, train, and evaluate a neural network for image classification. Next, we will explore a different type of data: natural language, using neural networks for text-based tasks like sentiment analysis.

Introduction to natural language processing

This section introduces **natural language processing (NLP)** and focuses on text classification through sentiment analysis. The objective is to build a sentiment analysis model that classifies text as positive or negative using a pre-trained model from Hugging Face.

Natural language processing (NLP) is a branch of AI that enables machines to understand, interpret, and manipulate human language. NLP plays a crucial role in applications such as chatbots, sentiment analysis, and machine translation.

Some of the common NLP tasks include:

- **Text classification**, where a document or sentence is classified, for example, as spam or not spam, or as having positive versus negative sentiment
- **Named entity recognition (NER)**, where entities such as names, locations, and dates are identified
- **Machine translation**, which refers to translating text from one language to another
- **Speech recognition,** which involves converting spoken words into text

Sentiment analysis: understanding text classification

Sentiment analysis is the process of determining whether a piece of text conveys positive, negative, or neutral sentiment. This technique is widely used in applications such as customer feedback, social media monitoring, and product reviews. For example, the statement "I am having an amazing day" is classified as positive, while "I am having a horrible day" is negative. A sentence such as "It's an amazing day, but work is where I have to be, so I don't like it much" represents a neutral sentiment, expressing both positive and negative aspects.

The problem statement for this section involves building a sentiment analysis model that classifies movie reviews as positive or negative using a pre-trained model.

Text preprocessing

Three main steps are covered in text preprocessing: tokenization, stemming and lemmatization, and vectorization.

Tokenization

Tokenization is the process of breaking a sentence into words or subwords, known as **tokens**. For example, the following code demonstrates tokenization using NLTK:

The following code imports the `word_tokenize` function from NLTK, defines a sample sentence, tokenizes it, and prints the resulting tokens:

```
from nltk.tokenize import word_tokenize
sentence = "I love this movie!"
tokens = word_tokenize(sentence)
print(tokens)
```

The tokenized output is:

```
['I', 'love', 'this', 'movie', '!']
```

This process separates words and punctuation, ensuring that each token is recognized individually.

Stemming and lemmatization

Stemming reduces words to their root form. For example, "running" is reduced to "run". The following code demonstrates stemming using NLTK's `PorterStemmer`:

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
```

```
print(stemmer.stem("running"))
print(stemmer.stem("standing"))
```

The output is:

```
run
stand
```

This reduction helps decrease the number of unique words in the input data.

Lemmatization on the other hand reduces words to their base or dictionary form, known as the lemma. Unlike stemming, lemmatization uses linguistic knowledge to produce meaningful root words. For example, "running" is reduced to "run", and "better" is reduced to "good". The following code demonstrates lemmatization using NLTK's WordNetLemmatizer:

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("running", pos='v'))
print(lemmatizer.lemmatize("better", pos='a'))
```

The output is:

```
run
good
```

Lemmatization provides more accurate results compared to stemming because it understands the context and part of speech of the word. This approach is particularly useful when you need to preserve the semantic meaning of words in your text processing pipeline.

Vectorization

Text data must be converted into numerical format to be processed by machine learning algorithms, as machines only understand numbers. **Vectorization** is the process of transforming text into numerical vectors. One common method is TFIDF, or term frequency-inverse document frequency. The following code demonstrates vectorization using scikit-learn's `TfidfVectorizer`:

```
from sklearn.feature_extraction.text import TfidfVectorizer
corpus = ["I love this movie", "This movie is terrible"]
vectorizer = TfidfVectorizer()
```

```
X = vectorizer.fit_transform(corpus)
print(X.toarray())
```

The output is:

```
[[0.         0.6316672  0.44943642 0.6316672  0. ]
[0.6316672  0.         0.44943642 0.         0.6316672 ]]
```

It converted the array to its vectorized format. This process converts the text into arrays of numerical values, representing the importance of each word in the corpus.

Using pre-trained models for NLP

Pre-trained models are used in NLP because they apply **transfer learning**, where models trained on large datasets are fine-tuned on smaller datasets, making them highly effective for various NLP tasks. **Hugging Face** provides a repository of pre-trained NLP models for tasks such as sentiment analysis, named entity recognition, and text classification.

The following steps are required to use a pre-trained sentiment analysis model from Hugging Face:

1. Install the Hugging Face Transformers library by running:

   ```
   pip install transformers
   ```

   This command provides access to all the models available from Hugging Face

2. Use a pre-trained sentiment analysis model. For example, the following code imports the pipeline function from transformers, loads the sentiment analysis pipeline, and classifies a sample text:

   ```
   from transformers import pipeline
   sentiment_pipeline = pipeline("sentiment-analysis")
   result = sentiment_pipeline("I love this movie")
   print(result)
   ```

   This code returns a result indicating positive sentiment for the sample text.

3. Next, we need to fine-tune the model. Fine-tuning is the process of adapting a pre-trained model to a specific task by training it on a smaller dataset. The following code

provides a complete, runnable fine-tuning example using the Trainer class and `TrainingArguments` from transformers:

```
from transformers import (
    AutoTokenizer, AutoModelForSequenceClassification,
    Trainer, TrainingArguments
)
import torch
from torch.utils.data import Dataset
model_name = "distilbert-base-uncased"
hf_tokenizer = AutoTokenizer.from_pretrained(model_name)
hf_model = AutoModelForSequenceClassification.from_pretrained(
    model_name, num_labels=2)
class SentimentDataset(Dataset):
    def __init__(self, texts, labels):
        self.encodings = hf_tokenizer(
            texts, truncation=True, padding=True, max_length=64
        )
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        item = {k: torch.tensor(v[idx]) for k, v in self.encodings.items()}
        item["labels"] = torch.tensor(self.labels[idx])
        return item
train_dataset = SentimentDataset(
    ["I love this movie", "This film was terrible",
     "Great acting", "I disliked it"],
    [1, 0, 1, 0]
)
eval_dataset = SentimentDataset(
    ["An excellent movie", "A disappointing film"],
    [1, 0]
)

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=1,
    per_device_train_batch_size=2,
```

```
    per_device_eval_batch_size=2,
    report_to="none"
)
trainer = Trainer(
    model=hf_model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset
)
trainer.train()
```

This complete example defines the tokenizer, model, training dataset, evaluation dataset, training arguments, and Trainer before starting fine-tuning, preventing undefined variable errors

These three steps: installing the transformers library, using a pre-trained sentiment analysis model, and optionally fine-tuning the model, form the workflow for using pre-trained models in NLP.

Building a sentiment analysis model with TensorFlow

The process for building a sentiment analysis model with TensorFlow involves three main steps: loading the dataset, preprocessing the data, and defining and training the model.

1. For loading the dataset, the IMDB movie reviews dataset is available via TensorFlow Datasets. The following code loads the dataset and returns the train and test splits along with dataset information:

```
import numpy as np
import tensorflow_datasets as tfds
(train_data, test_data), info = tfds.load(
    "imdb_reviews", split=["train", "test"],
    with_info=True, as_supervised=True)
```

2. The next step is to pre-process the data. This step involves tokenizing and padding sequences to ensure all reviews have a similar length. The following code imports the necessary classes and prepares the tokenizer:

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words=10000, oov_token="<OOV>")
```

```
train_texts, train_labels = [], []
test_texts, test_labels = [], []

for text, label in tfds.as_numpy(train_data):
    train_texts.append(text.decode("utf-8"))
    train_labels.append(label)

for text, label in tfds.as_numpy(test_data):
    test_texts.append(text.decode("utf-8"))
    test_labels.append(label)

train_labels = np.asarray(train_labels)
test_labels = np.asarray(test_labels)
tokenizer.fit_on_texts(train_texts)
train_sequences = tokenizer.texts_to_sequences(train_texts)
test_sequences = tokenizer.texts_to_sequences(test_texts)
padded_sequences = pad_sequences(train_sequences, maxlen=120)
padded_test_sequences = pad_sequences(test_sequences, maxlen=120)
```

3. The final step is to define and train the model. We'll build a simple LSTM-based sentiment analysis model using TensorFlow Keras. The following code imports the required classes:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
```

Next, let's create the model:

```
model = Sequential([
       Embedding(input_dim=10000, output_dim=64, input_length=120),
       LSTM(64),
       Dense(1, activation='sigmoid')
])
```

Let's optimize this model:

```
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

To train the model, the model's fit method is called with the padded sequences and training labels, specifying five epochs and providing validation data using the padded test sequences and test labels:

```
model.fit(padded_sequences, train_labels, epochs=5,
validation_data=(padded_test_sequences, test_labels))
```

That is all we need to do to build a sentiment analysis model with TensorFlow.

Model evaluation and confusion matrix

Model evaluation is performed using four metrics: accuracy, precision, recall, and F1 score. These metrics can be imported from scikit-learn as follows:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score
```

To generate predictions, the model's predict method is used on the padded test sequences. The predictions are then thresholded at 0.5 and converted to integers:

```
predictions = model.predict(padded_test_sequences)
predictions = (predictions > 0.5).astype(int).ravel()
```

The following code calculates the accuracy, precision, recall, and F1 score using the test labels and predictions:

```
accuracy = accuracy_score(test_labels, predictions)
precision = precision_score(test_labels, predictions)
recall = recall_score(test_labels, predictions)
f1 = f1_score(test_labels, predictions)
```

This approach evaluates the model using the four metrics based on the predictions from the previous section.

Next, to visualize the confusion matrix for better understanding of classification results, additional libraries are imported as follows:

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
```

The confusion matrix is computed and visualized as a heatmap:

```
cm = confusion_matrix(test_labels, predictions)
sns.heatmap(cm, annot=True, fmt='d')
plt.title('Confusion Matrix')
plt.show()
```

This entire process evaluates the model first by calculating the four metrics and then by visualizing the confusion matrix to better understand the classification results.

Hands-on project: Sentiment analysis using pre-trained NLP models

The task for this project is to build a sentiment analysis model using pre-trained models such as Hugging Face's DistilBERT or TensorFlow's IMDb movie review dataset. The steps followed are:

1. Preprocess the text data by tokenizing and padding sequences
2. Use a pre-trained model for sentiment analysis or build a model from scratch using TensorFlow
3. Train the model and monitor performance using accuracy, precision, recall, and F1 score
4. Evaluate and visualize the results using the confusion matrix

To use a pre-trained model from Hugging Face, the pipeline is imported and initialized for sentiment analysis:

```
from transformers import pipeline
sentiment_pipeline = pipeline('sentiment-analysis')
```

Sample texts are defined for sentiment analysis:

```
texts = ["I love this movie", "This movie was terrible"]
```

The sentiment pipeline is used to analyze the sample texts:

```
results = sentiment_pipeline(texts)
for text, result in zip(texts, results):
    print(f"Text: {text }, Sentiment: {result['label']}, Score:
{result['score']}")
```

The output for our code so far is:

```
Text: I love this movie | Sentiment: POSITIVE, Score: 0.9998766183853149
Text: This movie was terrible | Sentiment: NEGATIVE, Score: 0.9995916485786438
```

The output indicates that "I love this movie" is classified as positive with a score of 0.99, and "This movie was terrible" is classified as negative with a score of 0.99. If no model is specified, the pipeline defaults to using DistilBERT for sequence classification.

Sentiment analysis with TensorFlow and the IMDB dataset

To perform sentiment analysis using TensorFlow, import the necessary libraries:

```
import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

The next step is to load the IMDb dataset and unpack the train/test splits together with the dataset information:

```
(train_data, test_data), info = tfds.load('imdb_reviews', split=['train',
'test'], with_info=True, as_supervised=True)
```

Next, convert the TensorFlow string tensors to Python strings, collect the labels, tokenize the reviews, and pad the sequences as follows:

```
tokenizer = Tokenizer(num_words=10000)
train_texts, train_labels = [], []
for text, label in tfds.as_numpy(train_data):
    train_texts.append(text.decode("utf-8"))
    train_labels.append(label)
train_labels = np.asarray(train_labels)
tokenizer.fit_on_texts(train_texts)

train_sequences = tokenizer.texts_to_sequences(train_texts)
train_padded = pad_sequences(train_sequences, maxlen=120)
```

The LSTM model is defined and compiled using the following:

```
model = tf.keras.Sequential([
            tf.keras.layers.Embedding(10000, 16, input_length=120),
            tf.keras.layers.LSTM(64),
            tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(
            optimizer='adam',
            loss='binary_crossentropy',
            metrics=['accuracy']
)
```

The model is trained using padded sequences and labels:

```
model.fit(train_padded, train_labels, epochs=5, validation_split=0.2)
```

This process demonstrates how to load the data, tokenize and pad the sequences, and define and compile the LSTM model for sentiment analysis using TensorFlow.

Summary

Neural networks and deep learning concepts were introduced alongside practical implementation using popular frameworks, with a focus on building and evaluating a simple neural network for handwritten digit classification using the MNIST dataset. The chapter also covered foundational techniques in natural language processing, including text preprocessing, tokenization, stemming, lemmatization, and vectorization, culminating in sentiment analysis using both custom and pre-trained models. Deployment strategies for both image and text models as web services were discussed, highlighting model evaluation, debugging, and real-world application through hands-on projects.

In the next chapter, you will learn how to deploy these models as a web service, making them accessible to real-world users.

4

Deploying an AI Model as a Web Service

Deploying an AI model as a web service bridges the gap between machine learning development and real-world application, allowing intelligent systems to interact dynamically with users. This process involves transforming a trained model into an accessible online tool by integrating it with a web framework, building user interfaces, and configuring cloud deployment. As a result, AI-driven predictions become available to anyone with internet access, enabling practical, scalable solutions across diverse domains.

The objective of this chapter is to deploy a model using Flask, enabling users to input data and receive predictions from the deployed AI model. The emphasis is on integrating AI models into web applications and deploying them on a cloud platform such as Heroku. In this chapter, we will cover the following topics:

- Installing Flask
- Creating a basic Flask application
- Creating a web interface for the model
- Deploying the Flask application to Heroku

By the end of this chapter, we will:

- Understand the basics of web development using Flask
- Learn how to integrate a machine learning model into a web application
- Deploy a machine learning model as a RESTful API
- Deploy the application on a cloud platform like Heroku for public access

Technical requirements

To follow along with the examples and hands-on projects in this chapter, you will need the following:

- Python 3.8 or above
- A code editor such as Sublime Text or Visual Studio Code
- The following Python libraries installed via pip:
    - `flask`
    - `scikit-learn`
    - `joblib`
    - `pandas`
    - `numpy`
    - `tensorflow` (for Keras IMDB dataset)

You can install these libraries using the following command:

```
pip install flask scikit-learn joblib pandas numpy tensorflow
```

For cloud deployment (optional):

- A free Heroku account from `https://www.heroku.com`
- Heroku Command Line Interface (CLI) installed on your system

To install Heroku CLI, visit `https://devcenter.heroku.com/articles/heroku-cli` and follow the installation instructions for your operating system.

This chapter assumes you have a project directory (for example, `flask_app`) where you will create the following files:

- `myserver.py` (Flask application file)
- `index.html` (HTML template for the web interface)
- `imdb_sentiment_model.pkl` (saved trained model, generated during the chapter)
- `imdb_vectorizer.pkl` (saved TF-IDF vectorizer, generated during the chapter)
- `requirements.txt` (list of project dependencies, generated using pip freeze)

All code examples in this chapter assume that you have Jupyter Notebook or a Python IDE installed, with the required libraries available in your Python environment. The trained model files will be generated when you run the code to build and save the sentiment analysis model.

Installing Flask

Flask is a lightweight web framework for Python, commonly used to create web applications quickly. It is particularly useful for deploying machine learning models as a web service. The following steps are required to install Flask and prepare the project directory:

1. Open a terminal window
2. Navigate to the desired project directory using the `cd` command, for example, `cd Desktop` and then `cd flask_app` (or whatever is the name of your folder)
3. Install Flask using `pip` by running the following command:

```
pip install Flask
```

This command installs Flask, which typically completes within a few seconds.

Creating a basic Flask application

Let's begin by creating a basic Flask application. Perform the following steps:

1. Open a code editor, such as Sublime Text
2. Save a new file in the directory with the name `myserver.py`
3. Import the Flask class from the Flask library:

```
from flask import Flask
```

4. Initialize the Flask app by assigning `Flask(__name__)` to a variable:

```
app = Flask(__name__)
```

5. Define a route for the homepage using the `@app.route('/')` decorator. This is the route for the base path. We'll define a function such that the route returns a simple text response, such as "hello world":

```
@app.route('/')
def home():
    return 'hello world'
```

6. Initialize the app to run in debug mode by checking if `__name__ == '__main__'` and calling `app.run(debug=True)`:

```
if __name__ == '__main__':
    app.run(debug=True)
```

The entire code looks like this:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'hello world'

if __name__ == '__main__':
    app.run(debug=True)
```

To run the Flask application, the following command is executed in the terminal:

```
python myserver.py
```

The server starts and indicates the address where it is running, such as localhost. Opening this address in a web browser displays the text "hello world."

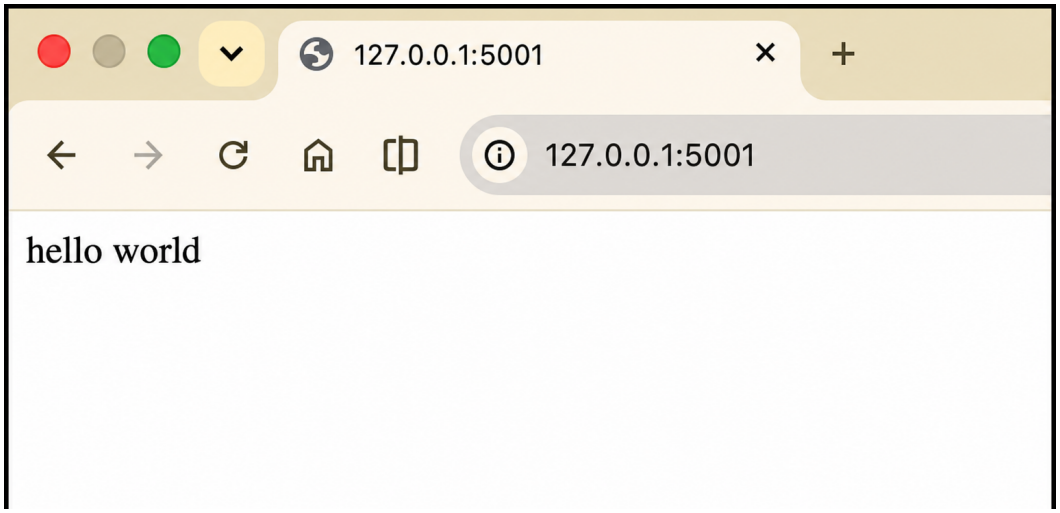


Figure 4.1: Browser displaying the output of a basic Flask application showing "hello world"

Modifying the return value in the route function, for example to "hello my first Flask app", refreshing the browser updates the displayed text accordingly. The route can return plain text or HTML:

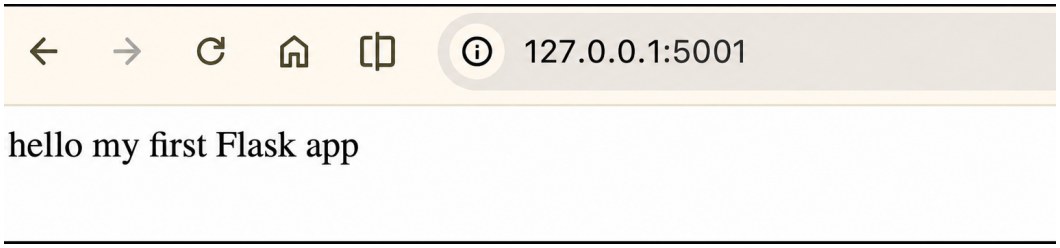


Figure 4.2: Browser displaying the output of a basic Flask application showing "hello my first Flask app"

The `@app.route` decorator defines a route for the webpage, and the `app.run` function starts the Flask development server.

Creating a web interface for the model

The process of creating a web interface for the model is divided into three steps:

1. Build a prediction function
2. Create a Flask web form for input
3. Create the prediction route

The first step is to build a prediction function. Let's get started.

Building a prediction function

Logistic regression is used to create a prediction model from the IMDB database. The trained model can be loaded using Pickle or Joblib, and a function is created to make predictions. The following code imports the necessary libraries for building the prediction function:

```
import pandas as pd
from sklearn.model_selection import train_test_split
```

The following libraries are imported to support data handling, feature extraction, model training, evaluation, serialization, and numerical operations: pandas, scikit-learn's `train_test_split`, `TfidfVectorizer`, `LogisticRegression`, `accuracy_score`, `Joblib`, and NumPy:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import joblib
import numpy as np
from keras.datasets import imdb
from keras.preprocessing import sequence
```

The IMDB dataset is used as the data source, and Keras's preprocessing sequence module is also imported.

The next step is to load the IMDB dataset:

```
num_words = 5000
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=num_words)
```

The IMDB dataset is large, so only the top 5000 most frequent words are retained. The data is split into training and test sets, with variables `X_train`, `y_train`, `X_test`, and `y_test` assigned accordingly. The IMDB data is loaded by specifying the number of words to keep, which controls the vocabulary size and limits the amount of data loaded from the IMDB database. This approach ensures that only the most relevant words are included for model training and evaluation.

The next step is to convert the integer sequences back to words using Keras' word index. This is accomplished by assigning the word index from the IMDB dataset to a variable:

```
word_index = imdb.get_word_index()
```

This mapping allows conversion of integer sequences back to words. To enable this, a reverse mapping is defined for converting the word index back to words:

```
reverse_word_index = {v: k for k, v in word_index.items()}
```

This reverse mapping enables the transformation of sequences into readable text, which is necessary for use with TF-IDF. The following function is defined to decode a review from a sequence of integers to a string of words:

```
def decode_review(text_sequence):
    return ' '.join([reverse_word_index.get(i - 3, '?') for i in text_sequence])
```

This function converts sequences to readable text. The next step is to decode the text sequences for both the training and test datasets:

```
X_train_text = [decode_review(sequence) for sequence in X_train]
X_test_text = [decode_review(sequence) for sequence in X_test]
```

The next step is data preprocessing. The text data is converted into numerical features using **TF-IDF**. The TF-IDF vectorizer is initialized with English stop words and the maximum number of features set to the number of words:

```
vectorizer = TfidfVectorizer(stop_words='english', max_features=num_words)
xtrain_tfidf = vectorizer.fit_transform(X_train_text)
xtest_tfidf = vectorizer.transform(X_test_text)
```

The next step is training the **logistic regression** model. The model is initialized with a maximum of 200 iterations and trained using the TF-IDF features and the training labels:

```
model = LogisticRegression(max_iter=200)
model.fit(xtrain_tfidf, y_train)
```

After training, the model is evaluated. Predictions are made on the test set, and the accuracy is calculated and printed:

```
y_pred = model.predict(xtest_tfidf)
accuracy = accuracy_score(y_test, y_pred)
print(f'Model Accuracy: {accuracy:.4f}')
```

Finally, after creating the model, both the model and the vectorizer are saved using `joblib`. This allows the trained model and vectorizer to be loaded later by the Flask application:

```
joblib.dump(model, 'imdb_sentiment_model.pkl')
joblib.dump(vectorizer, 'imdb_vectorizer.pkl')
```

Running this code produces an accuracy value, for example, 0.8778, and confirms that the files `imdb_sentiment_model.pkl` and `imdb_vectorizer.pkl` have been created. These files are available in the working directory and can be used by the Flask application.

Creating a Flask web form for input

The second step is to create a Flask web form for input. An HTML file is created in the project folder. The file is named `index.html` and contains the following structure:

```
<html>
    <body>
        <h1>Movie Review</h1>
        <form action="/predict" method="post">
            <label for="review">Enter review text</label><br>
            <textarea name="review" rows="10" cols="50"></textarea><br><br>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
```

Opening this file in a browser displays a form with a header, a label, a text area for entering the review, and a submit button. Submitting the form sends the data to the `/predict` route of the web application.

To render this HTML file from Flask, create a new Flask application file, name it `myserver.py`. The following code is used to import Flask and render the HTML template:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

The `render_template` function retrieves the HTML file from the `templates` folder and returns it as the response for the root route.

Additionally, for the `render_template` to be supplied, place the `index.html` file in a folder—let's call it `templates`.

To verify that the setup works, open a terminal and change the working directory to the project folder. The Flask application can then be started from the terminal using the command:

```
python myserver.py
```

The terminal output confirms the application is running and ready to accept requests. The following figure shows the terminal output after running the application. It shows the submit form with input and the button:

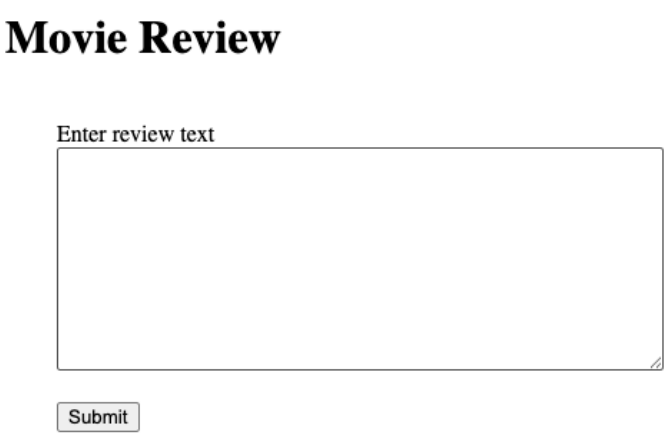


Figure 4.3: HTML web form for movie review input with a text area for entering review text and a submit button

Creating the prediction route in Flask

When the form is submitted, it sends the data to the `/predict` route. If this route is not defined in the server code, Flask returns a "not found" error for the `/predict` URL. To resolve this, a new route must be added to the Flask application.

The following code defines the `/predict` route and specifies that it accepts POST requests, matching the method used in the HTML form:

```
@app.route('/predict', methods=['POST'])
def predict():
    return "Negative"
```

After adding this route, submitting the form returns the response "negative" from the server, confirming that the route is functioning. However, no prediction logic has been implemented yet.

Loading the model and vectorizer

To enable predictions, the necessary libraries must be imported and the trained model and vectorizer loaded. The following code imports the required modules and loads the model and vectorizer using `joblib`:

```
import joblib

model = joblib.load('imdb_sentiment_model.pkl')
vectorizer = joblib.load('imdb_vectorizer.pkl')
```

Implementing the prediction logic

Within the `predict` function, the review text is retrieved from the submitted form data. The review is accessed using the key corresponding to the text area's name attribute in the HTML form. The function checks if the review text is provided; if not, it returns a message indicating that no review is available.

If the review text is present, it is transformed using the loaded vectorizer, and the model is used to predict the sentiment. The prediction result is interpreted as either positive or negative, and the response is returned accordingly. The following code implements this logic:

```
from flask import request

@app.route('/predict', methods=['POST'])
def predict():
    review = request.form.get('review')
    if not review:
        return "No review available"
    review_tfidf = vectorizer.transform([review])
    prediction = model.predict(review_tfidf)
```

```
    sentiment = "positive" if prediction[0] == 1 else "negative"
    return f"The movie review is <b>{sentiment}</b>"
```

If the `request` module is not imported, a "request is not defined" error occurs. Importing `request` from Flask resolves this issue.

Testing the prediction route

After implementing the prediction logic, the server is restarted. Submitting different reviews through the web form produces the expected results. For example, submitting "the movie was horrible" returns "movie review is negative.". This is shown in the following figure:

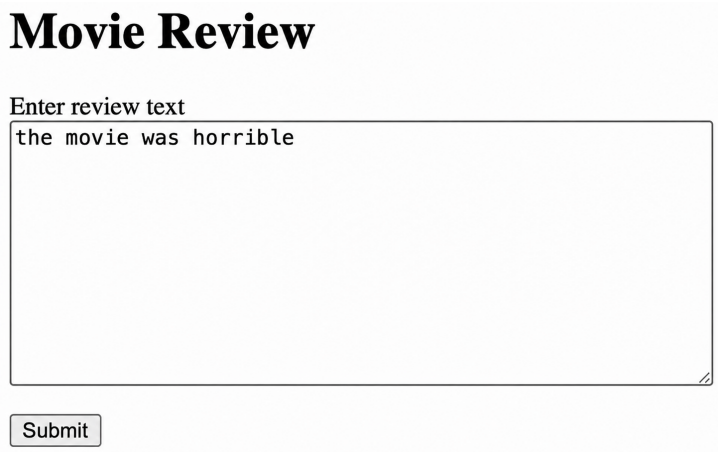


Figure 4.4: Flask web application displaying the submitted movie review form with input field and submit button

The following image shows the sentiment generated:



Figure 4.5: Web form submission showing negative sentiment prediction result for the review

While submitting "love the movie" returns "the movie review is positive".

This completes the steps of building a prediction function, creating a Flask web form for input, and defining the prediction route.

Deploying the Flask application to Heroku

Deploying the Flask application to Heroku involves three main steps: setting up Heroku, preparing the application for deployment, and deploying to Heroku and testing the deployment.

Setting up Heroku

Heroku is a cloud platform that simplifies the deployment of web applications. An application can be created directly from the Heroku website `https://www.dashboard.heroku.com/apps` or using the Heroku CLI. You will need to sign up on the Heroku website here `https://www.dashboard.heroku.com`.

The CLI is recommended for its ability to create pipelines and manage deployments efficiently. To begin, the Heroku CLI must be installed. The documentation and installation instructions are available at this website: `https://devcenter.heroku.com/articles/heroku-cli`

Heroku provides several installation methods, including tarballs, installation via Ubuntu or Arch Linux, and installation with `npm`. It is recommended to use one of the standard installation methods. The following command installs the Heroku CLI globally using `npm`:

```
npm install -g heroku
```

After installation, the version can be checked with the following command:

```
heroku version
```

The terminal output confirms that the latest version of the Heroku CLI is installed.

Getting started with the Heroku CLI

Start by logging into CLI using:

```
heroku login
```

Press any key to continue. Click on the login button, and it will log you in inside the CLI.

Preparing the application for deployment

The process of preparing the application for deployment begins by organizing the project files. The application is located inside a folder. Within this folder, create a new directory named `webApp`. The `templates` folder is moved into this new `webApp` folder. Move the main server file,

named `myserver.py`, also into `webApp`, along with the pickle files (`.pkl`) required for the application. This collection of files constitutes the web application.

It is common practice to use the name `app.py` for the main application file, as this is a standard convention. Rename the `myserver.py` file to `app.py`.

To prepare the application for deployment, a `Procfile` must be created in the root directory of the `webApp` folder. The `Procfile` specifies how the application should be run. The following line is added to the `Procfile` to indicate that the app will be run using Gunicorn, a production-grade WSGI HTTP server:

```
web: gunicorn app:app
```

This configuration ensures that Heroku knows how to launch the application.

The next requirement is a `requirements.txt` file in the `templates` folder, which lists all the Python dependencies needed by the application. This file can be generated using a command in the terminal. The following steps are performed:

1. Navigate to the `webApp` directory using the terminal
2. Run the command to generate the requirements file:

```
pip freeze > requirements.txt
```

This creates a `requirements.txt` file containing all the necessary libraries for the application.

Deploying to Heroku

With the application prepared, the next step is to deploy it to Heroku. The following sequence of commands is executed in the terminal:

1. Initialize a new Git repository in the `webApp` directory:

```
git init
```

2. Add all files to the repository:

```
git add .
```

3. Commit the files with a message:

```
git commit -m "Initial commit"
```

4. Create a new Heroku application:

```
heroku create
```

   This command creates the app and provides the location where it is hosted.

5. Push the code to Heroku:

```
git push heroku master
```

This uploads the code to the Heroku server, where it processes the requirements file and installs all necessary Python packages.

After deployment, Heroku confirms that everything is set up and the application is live. To open the deployed application, the following command is used:

```
heroku open
```

This command opens the deployed web application in a browser. Occasionally, it may take a short time for the application to become accessible. If the application does not load immediately, retrying after a brief wait may resolve the issue:

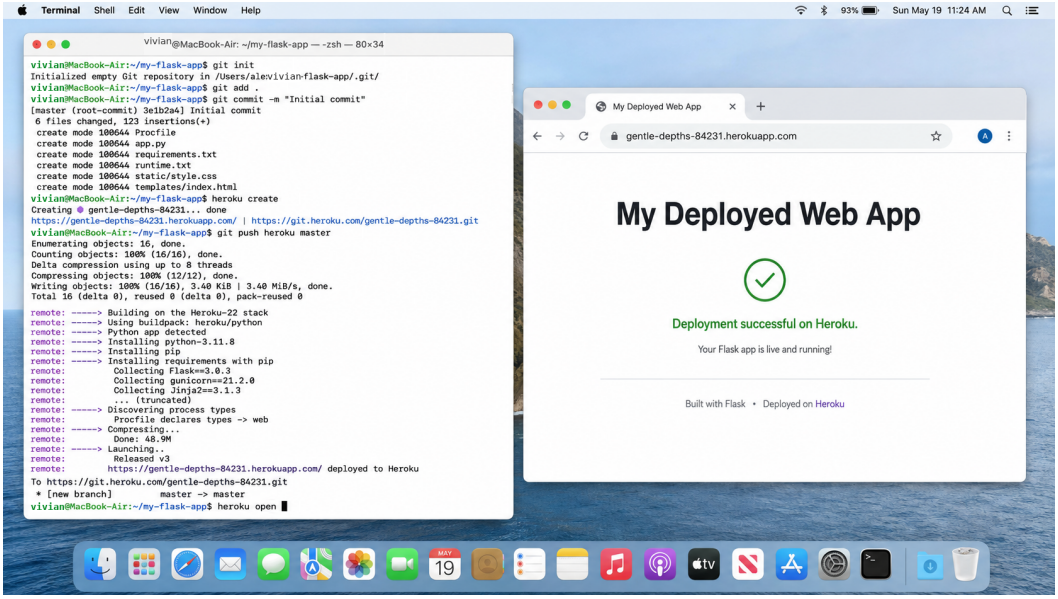


Figure 4.6: Deployment confirmation screen showing the success message

Once the application is accessible, its functionality can be tested by submitting sample inputs through the web interface. For example, submitting "The movie is amazing" returns a positive result, while "The movie is bad" returns a negative result. This demonstrates that the deployed Flask application is functioning as intended.

Testing the web service and API

After deployment, it is important to test both the web service and the API, including handling edge cases. The web service can be tested by entering text into the form on the deployed application. Alternatively, requests can be sent using tools such as Postman or `curl` to simulate client interactions with the API.

To test the API endpoint, a request can be made to the application's URL. For example, submitting "hello world" through the form returns a positive result. Submitting "not so good" may produce less certain results, as the model attempts to interpret negation and sentiment. The following figure illustrates this output:

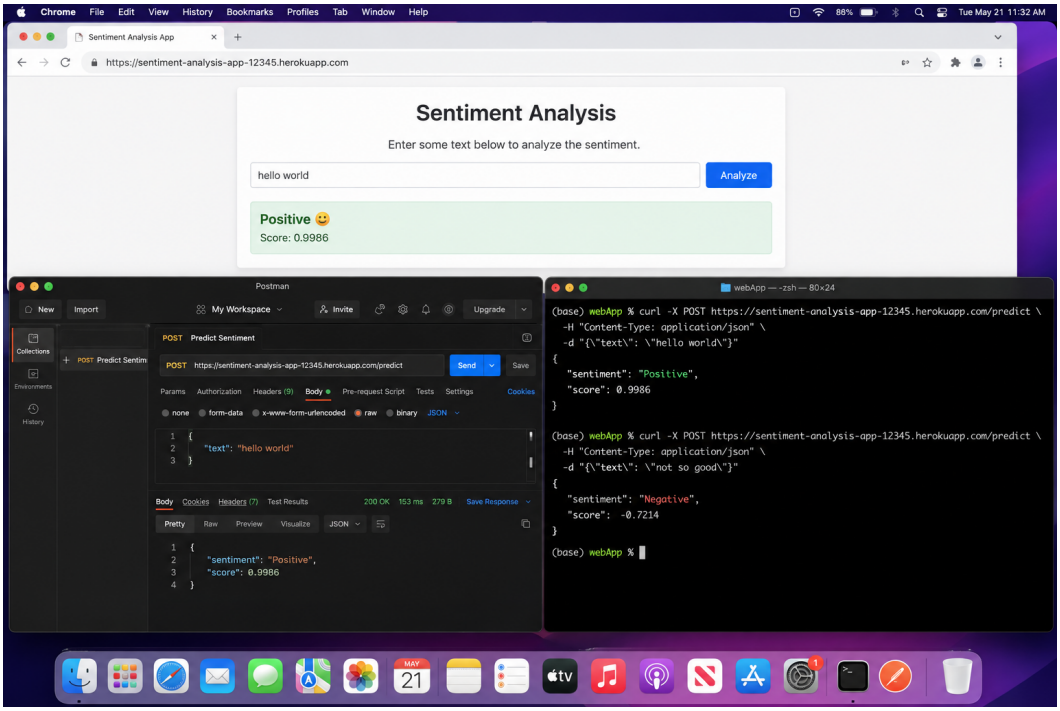


Figure 4.7: Sentiment analysis API testing showing Postman request results with "hello world" classified as positive and "not so good" classified as negative

For more advanced testing, Postman can be used:

1. Open Postman and create a new request
2. Set the request type to POST
3. Enter the application URL with the `/predict` endpoint
4. In the body, select form data and add a key named review with the desired review text
5. Send the request and observe the response

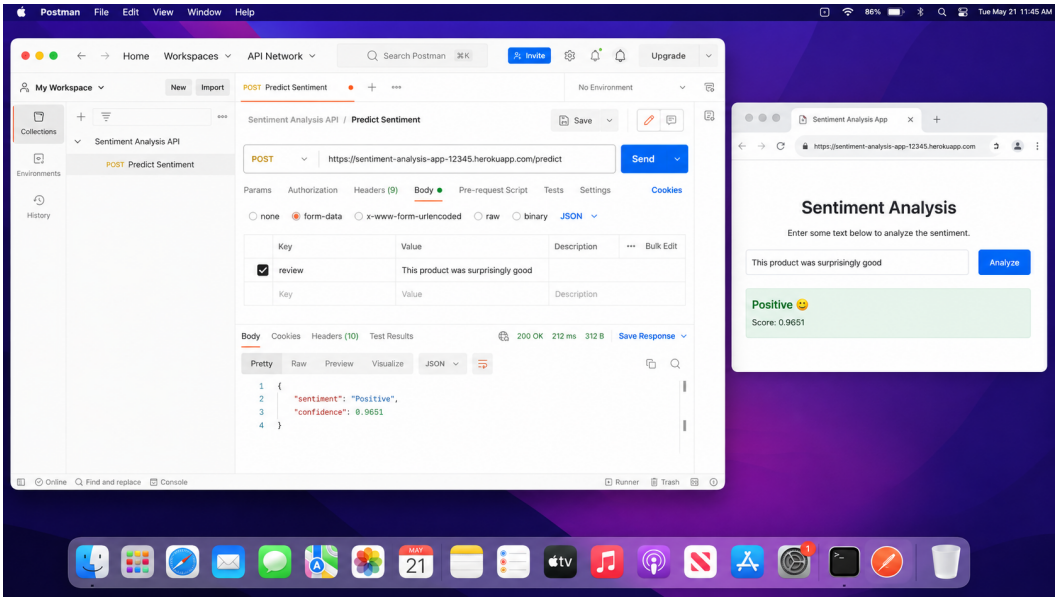


Figure 4.8: Postman interface displaying an API request to the deployed sentiment analysis service with the response showing a positive sentiment classification

This approach allows for testing various edge cases and verifying that the API responds as expected.

The API can also be tested using `curl` from the terminal. By copying the appropriate `curl` command and executing it in the terminal, a request is sent to the deployed application, and the response is displayed in the terminal. For example, submitting "Movie was bad" returns a negative result.

These methods provide multiple options for testing the deployed web service and ensuring that it operates correctly in different scenarios.

Summary

Deploying an AI model as a web service involves building a user-friendly interface and making predictions accessible over the internet. The process begins with installing Flask and developing a basic application, followed by designing a web form to collect user input and implementing the necessary routes to handle predictions using a pre-trained model and vectorizer. The workflow concludes with preparing the application for deployment, setting up a Heroku account, deploying the Flask app to Heroku, and verifying the functionality of the web service and API through testing.

In the next chapter, you will apply everything covered so far through a series of hands-on projects that bring together Python, machine learning, neural networks, and deployment.

5

Supervised Learning and Forecasting Projects

Putting concepts into practice is the most effective way to build confidence with AI and Python. This chapter brings together the programming and machine learning foundations covered in earlier chapters and applies them through a structured series of hands-on projects, progressing from beginner-level exercises to more complex machine learning and forecasting tasks. Each project is self-contained, building on a specific skill or technique and reinforcing it through real code and real data.

The following topics are covered in this chapter:

- Beginner projects
- Machine learning projects
- Forecasting and prediction projects

Technical requirements

To follow along with the projects in this chapter, you will need the following:

- Python 3.8 or above installed on your machine
- Jupyter Notebook or any Python IDE (such as VS Code or PyCharm)
- The following Python libraries installed via pip:
    - `pandas`
    - `numpy`
    - `scikit-learn`
    - `matplotlib`

- `seaborn`
- `xgboost`

You can install all required libraries at once using:

```
pip install pandas numpy scikit-learn matplotlib seaborn xgboost
```

Datasets used in this chapter can be downloaded from Kaggle (`https://www.kaggle.com`). A free Kaggle account is required to download datasets. Specific dataset instructions are provided within each project.

Beginner projects

This section introduces three entry-level projects that cover core Python and text-based machine learning skills. Starting with a command-line calculator to reinforce control flow and function definitions, the projects then move into building a rule-based chatbot using pattern matching and a spam email detector using logistic regression. These projects are designed to be accessible with no prior machine learning experience, while laying the groundwork for the more complex models that follow.

Implementing a basic calculator in Python

This project demonstrates the implementation of a simple command-line calculator capable of performing basic arithmetic operations: addition, subtraction, multiplication, and division.

The implementation begins by creating a new Python file.

Defining arithmetic functions

The first step is to define functions in Python for each arithmetic operation. Each function takes two arguments, `x` and `y`, and returns the result of the operation.

The following code defines the function for addition:

```
def add(x, y):
    return x + y
```

Similarly, functions for subtraction, multiplication, and division are defined. For division, an additional check is included to handle division by zero:

```
def subtract(x, y):
    return x - y

def multiply(x, y):
```

```
    return x * y

def divide(x, y):
    if y == 0:
        return "Error: division by zero not allowed"
    else:
        return x / y
```

This approach ensures that all basic arithmetic operations required by the calculator are supported.

Creating the calculator function

A calculator function is defined to manage user interaction and perform calculations based on user input. This function does not take any parameters.

The calculator function displays a menu of operations and prompts the user to select an operation. The following steps are performed:

1. Print the operation selection menu
2. Enter a loop that continues to prompt the user for input until the user chooses to exit
3. Prompt the user to select an operation by entering a number corresponding to addition, subtraction, multiplication, or division
4. Check if the user's choice is valid
5. If valid, prompt the user to enter the first and second numbers, supporting both integers and floats
6. Perform the selected operation by calling the corresponding function and display the result
7. Ask the user if another calculation should be performed
8. If the user enters anything other than "yes", exit the loop and print a goodbye message

The following code implements the calculator function:

```
def calculator():
    print("Select operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")

    while True:
```

```
        choice = input("Enter choice (1/2/3/4): ")
        if choice in ['1', '2', '3', '4']:
            num1 = float(input("Enter first number: "))
            num2 = float(input("Enter second number: "))

            if choice == '1':
                print(f"{num1} + {num2} = {add(num1, num2)}")
            elif choice == '2':
                print(f"{num1} - {num2} = {subtract(num1, num2)}")
            elif choice == '3':
                print(f"{num1} * {num2} = {multiply(num1, num2)}")
            elif choice == '4':
                print(f"{num1} / {num2} = {divide(num1, num2)}")
            else:
                print("Invalid input")

        next_calculation = input("Do you want to perform another calculation?
(yes/no): ")
        if next_calculation.lower() != "yes":
            break
    print("Exiting Calculator. Goodbye!")
```

Running the calculator

To run the calculator, the following steps are performed:

1. Open a terminal window
2. Navigate to the folder where this file is created
3. Ensure that the file is present in the directory
4. Run the calculator by executing the command:

```
python <your file name.py>
```

Tip

Replace the `<your file name.py>` with your actual `.py` file.

The calculator prompts the user to select an operation, enter two numbers, and displays the result. The input is taken as a float to support both whole numbers and decimal values.

Example interaction is as follows:

```
Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
Enter choice (1/2/3/4): 1
Enter first number: 56
Enter second number: 28
56.0 + 28.0 = 84.0
Do you want to perform another calculation? (yes/no): yes
Enter choice (1/2/3/4): 2
Enter first number: 20
Enter second number: 10
20.0 - 10.0 = 10.0
Do you want to perform another calculation? (yes/no): yes
Enter choice (1/2/3/4): 3
Enter first number: 10
Enter second number: 5
10.0 * 5.0 = 50.0
Do you want to perform another calculation? (yes/no): yes
Enter choice (1/2/3/4): 4
Enter first number: 10
Enter second number: 0
10.0 / 0.0 = Error: division by zero not allowed
Do you want to perform another calculation? (yes/no): yes
Enter choice (1/2/3/4): 4
Enter first number: 55
Enter second number: 5
55.0 / 5.0 = 11.0
Do you want to perform another calculation? (yes/no): no
Exiting calculator. Goodbye.
```

The calculator continues to prompt for operations until the user chooses to exit by entering "no."

Building a simple chatbot using predefined responses

This section introduces the creation of a simple chatbot that uses predefined responses. To construct a simple chatbot in Python that responds with predefined answers based on

keywords in user input, a rule-based approach is used. This method involves defining a set of responses for specific patterns or keywords.

Defining the response dictionary

The process begins by creating a new Python file. Import the regular `expression` module, which is built into Python, to handle pattern matching. This is accomplished with the following code, which enables the use of regular expressions for matching user input:

```
import re
```

Create a dictionary that maps keywords to predefined responses. The following code defines a responses dictionary, where each key is a keyword or phrase, and each value is the chatbot's reply. For example, if the input contains "hello," the chatbot responds with "Hi there. How can I assist you today?" Similarly, "hi" maps to "Hello. How can I help you?" and "how are you" maps to "I am just bored, but I am doing great. How about you?" This structure allows the chatbot to provide a specific reply for each recognized input. Here is how the dictionary looks:

```
responses = {
"hello": "Hi there. How can I assist you today?",
"hi": "Hello. How can I help you?",
"how are you": "I am just bored, but I am doing great. How about you?",
"what is your name": "I'm a chatbot created to assist you.",
"help": "Sure, I'm here to help. What do you need assistance with?",
"bye": "Goodbye! Have a great day!",
"thank you": "You're welcome! I'm happy to help.",
"default": "I'm not sure I understand. Could you please rephrase?"
}
```

Matching user input to responses

Next, define a function to find the appropriate response based on the user's input. The function takes the user's input as an argument and processes it to determine the correct reply. The first step inside the function is to convert the user input to lowercase, ensuring that the matching process is case-insensitive. This is achieved with the following code:

```
def chatbot_response(user_input):
    user_input = user_input.lower()
```

To determine which predefined response to use, the function must iterate through the keywords defined in the responses dictionary and check if any of these keywords are present in

the user's input. This is accomplished by looping over each keyword and using the regular expression search function to look for the keyword within the user's input.

The following code demonstrates how to perform this check and return the corresponding response if a keyword match is found:

```
for keyword in responses:
    if re.search(keyword, user_input):
        return responses[keyword]
```

If none of the keywords match the user's input, the function should return a default response. The responses dictionary includes a key called "default" with the value "I am not sure I understand. Could you please rephrase?" This ensures that the chatbot always provides a reply, even when the input does not match any predefined keyword. The code for handling the default response is as follows:

```
return responses["default"]
```

The complete `chatbot_response` function, incorporating both the keyword search and the default response, is as shown:

```
def chatbot_response(user_input):
    user_input = user_input.lower()
    for keyword in responses:
        if re.search(keyword, user_input):
            return responses[keyword]

    return responses["default"]
```

Running the chatbot

To interact with the chatbot, a main function is defined to run the conversation loop. This function prints an initial greeting from the chatbot and instructs the user to type "bye" to exit the conversation. The function then enters a loop that continuously prompts the user for input, processes the input, and prints the chatbot's response. The following code sets up the main loop for the chatbot:

```
def chatbot():
    print("Chatbot: Hello. I'm here to assist you.")
    print("Type 'bye' to exit.")
    while True:
```

```
        user_input = input("You: ")
        if user_input.lower() == "bye":
            print("Chatbot: Goodbye! Have a great day.")
            break
        response = chatbot_response(user_input)
        print(f"Chatbot: {response}")

chatbot()
```

With this structure, the chatbot will continue to interact until the user types "bye," at which point it will print a farewell message and exit the loop.

Now the chatbot can be executed from outside the loop by simply calling the chatbot function. This approach allows the main conversation loop to start immediately. The file should be saved before running the chatbot.

To run the chatbot, the following steps are performed:

1. Open a terminal window
2. Check the current working directory to ensure it is the correct folder containing the chatbot script
3. Execute the script

Upon running the script, the chatbot prints its greeting:

```
Chatbot: Hello. I'm here to assist you.
Type 'bye' to exit.
You: hello
Chatbot: Hi there. How can I assist you today?
```

Typing "hi" results in:

```
Chatbot: Hello. How can I help you?
```

Similarly, entering "help" produces:

```
Chatbot: Sure, I'm here to help. What do you need assistance with?
```

This demonstrates that the chatbot responds with predefined answers based on the detected keyword in the user's input. Additional responses can be added to the chatbot by expanding the responses dictionary.

The chatbot constructed in this section is a manual chatbot that waits for the user to type a message, searches for a keyword in the input, and replies with a predefined response. The conversation continues until the user types "bye," at which point the chatbot exits.

Building a spam email detector using scikit-learn

This project focuses on constructing a machine learning model that classifies emails as spam or not spam, based on their textual content. This process involves techniques such as text preprocessing and logistic regression, transforming raw email data into features that the model can interpret. The project demonstrates the fundamental steps in text classification and provides the knowledge required to build models that process and interpret text data.

Downloading and preparing the dataset

The first step is to obtain the dataset. The required dataset is the *Spambase* dataset from *UC Irvine*, which is available on Kaggle. The following steps outline the process for acquiring and preparing the dataset:

1. Navigate to `https://www.kaggle.com`
2. If an account does not exist, create a new account
3. Search for "*Spambase UC Irvine*" on Kaggle
4. Download the first dataset that appears. For this project, we are using .
5. Extract the downloaded zip file from the downloads folder
6. Locate the file named `Spambase.csv`
7. Move `Spambase.csv` to the working folder
8. Rename the file to `spam.csv` for convenience
9. Create a new Python file and save it

Importing libraries for spam detection

The next step is to import the necessary libraries for the project. The following code imports the required libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
```

The libraries in this code are as follows:

- `import pandas as pd`, imports the **pandas** library using the alias `pd`. Pandas is a data manipulation library that enables loading and manipulating data in a structured format such as a DataFrame.
- `from sklearn.model_selection import train_test_split`, imports the `train_test_split` function from the **scikit-learn** library. This function is used to split data into training and testing sets, which is a common step in preparing data for machine learning models.
- `from sklearn.linear_model import LogisticRegression` imports the **logistic regression** model from scikit-learn: Logistic regression is used for binary classification problems. In this context, the task is to classify emails as either spam or not spam.
- The next line imports several performance metrics from scikit-learn. These metrics are used to evaluate the performance of the classification model.
    - The `confusion_matrix` generates a matrix that displays the performance of the classification by showing the counts of true positives, true negatives, false positives, and false negatives
    - The `accuracy_score` calculates how often the model's predictions are correct
    - The `precision_score` measures the proportion of positive predictions that are actually correct
    - The `recall_score` measures the proportion of actual positives that are correctly identified by the model.
    - The `f1_score` is the harmonic mean of precision and recall, providing a single metric that balances both concerns
- `import seaborn as sns` imports the **seaborn** library for data visualization. Seaborn is built on top of matplotlib and provides functions to create attractive and informative statistical graphics.
- `import matplotlib.pyplot as plt` brings in the **pyplot** module from matplotlib. This module enables the creation of various graphs and charts, such as the confusion matrix that will be plotted later.

Loading and splitting the dataset

The next step involves loading the dataset and splitting it into training and testing sets.

1. Load the dataset using pandas by reading the CSV file named `spambase.csv` and assign it to the variable `data`.

2. Create the feature matrix `x` by dropping the column named `spam` from the dataset. This removes the target column, leaving only the features used for prediction.
3. Assign the target variable `y` as the `spam` column from the dataset. This column contains binary labels: 0 for not spam and 1 for spam.

The following code performs these steps:

```
data = pd.read_csv('spambase.csv')
x = data.drop('spam', axis=1)
y = data['spam']
```

At this point, `x` contains all columns except the target column `spam`, and `y` contains the binary labels indicating whether an email is spam.

The next step is to split the dataset into training and testing sets. This is done using the `train_test_split` function. The function requires the feature set `x`, the target `y`, the proportion of the data to use for testing, and a random seed for reproducibility. The steps are as follows:

1. Call `train_test_split` with `x` and `y` as arguments
2. Set `test_size=0.2` to allocate 20 percent of the data for testing and 80 percent for training
3. Set `random_state=42` to ensure the split is reproducible

The following code splits the data and assigns the resulting arrays to variables for training and testing features and labels:

```
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
random_state=42)
```

This produces four variables: `x_train`, `x_test`, `y_train`, and `y_test`, which represent the training features, testing features, training labels, and testing labels, respectively.

Training the logistic regression model

The next step is to train the logistic regression model to classify emails as spam or not spam. The following code performs these actions:

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
model = LogisticRegression(max_iter=1000)
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)
```

The first three lines standardize the training and test features using StandardScaler. The scaler is fitted only on the training data and then applied to the test data. The logistic regression model is initialized with `max_iter=1000` to give the lbfgs solver enough iterations to converge. The model is trained using the scaled training data, and the final line predicts whether the records in the test set are spam or not spam.

Printing the original, unscaled test feature data can be helpful for inspection before proceeding with evaluation. The following code prints the contents of `X_test`:

```
print(X_test)
```

The output is as follows:

```
      word_freq_make  word_freq_address  word_freq_all  word_freq_3d  \
3683            0.00                0.0           0.00           0.0
4412            0.71                0.0           0.71           0.0
2584            0.00                0.0           0.91           0.0
69              0.00                0.0           0.00           0.0
1844            0.00                0.0           0.54           0.0
...              ...                ...            ...           ...
1068            0.16                0.0           0.67           0.0
4351            0.00                0.0           0.00           0.0
3027            0.00                0.0           1.60           0.0
3455            0.75                0.0           0.00           0.0
1898            0.00                2.4           0.00           0.0

      word_freq_our  word_freq_over  word_freq_remove  word_freq_internet  \
3683           0.00            0.00              0.00                0.00
4412           0.00            0.00              0.00                0.00
2584           0.00            0.00              0.00                0.45
69             0.00            0.00              0.00                0.00
1844           0.00            0.00              0.00                0.00
...             ...             ...               ...                 ...
1068           0.33            0.16              0.33                0.84
4351           0.00            0.00              0.00                0.00
3027           0.00            0.00              0.00                0.00
3455           0.00            0.00              0.37                0.00
```

```
1898          0.80           0.00             0.00               0.00

      word_freq_order  word_freq_mail  ...  word_freq_conference  char_freq_;  \
3683             0.00            0.00  ...                   0.0         0.00
4412             0.00            0.71  ...                   0.0         0.00
2584             0.00            0.00  ...                   0.0         0.00
69               0.00            0.00  ...                   0.0         0.00
1844             0.00            0.00  ...                   0.0         0.00
...               ...             ...  ...                   ...          ...
1068             0.16            0.50  ...                   0.0         0.00
4351             0.00            0.00  ...                   0.0         0.00
3027             0.00            0.00  ...                   0.0         0.00
3455             0.00            0.37  ...                   0.0         0.00
1898             0.00            0.00  ...                   0.0         0.12

      char_freq_(  char_freq_[  char_freq_!  char_freq_$  char_freq_#  \
3683        0.000        0.000        0.000        0.000          0.0
4412        0.000        0.000        0.000        0.000          0.0
2584        0.000        0.000        0.000        0.000          0.0
69          0.201        0.000        0.000        0.100          0.0
1844        0.188        0.047        0.000        0.000          0.0
...           ...          ...          ...          ...          ...
1068        0.224        0.000        1.151        0.056          0.0
4351        0.000        0.000        0.000        0.000          0.0
3027        0.250        0.000        0.000        0.000          0.0
3455        0.099        0.597        0.000        0.000          0.0
1898        0.000        0.000        0.000        0.000          0.0

      capital_run_length_average  capital_run_length_longest  \
3683                       1.000                           1
4412                       1.032                           2
2584                       1.320                           7
69                         4.548                          59
1844                       1.745                          12
...                          ...                         ...
1068                       4.928                          63
4351                       1.000                           1
3027                       2.065                          12
3455                       2.125                          13
1898                       1.696                          15
```

```
      capital_run_length_total
3683                         3
4412                        32
2584                       103
69                         141
1844                        89
...                        ...
1068                       621
4351                         3
3027                        95
3455                        85
1898                        56

[921 rows x 57 columns]
```

Evaluating the model and visualizing results

The next step is to evaluate the trained model using several metrics: accuracy, confusion matrix, precision, recall, and F1 score. These metrics provide a comprehensive view of the model's performance in classifying emails as spam or not spam.

1. Calculate the **accuracy** of the model using `accuracy_score`, passing `y_test` and `y_pred` as arguments. Accuracy represents the percentage of correct predictions, including both spam and not spam emails.
2. Calculate the **precision** using `precision_score` with the same arguments. Precision is the proportion of emails predicted as spam that are actually spam, computed as true positives divided by the sum of true positives and false positives.
3. Calculate the **recall** using `recall_score`, again with `y_test` and `y_pred`. Recall is the proportion of actual spam emails that are correctly predicted, calculated as true positives divided by the sum of true positives and false negatives.
4. Calculate the **F1 score** using `f1_score` with the same arguments. The F1 score is the harmonic mean of precision and recall, providing a single metric that balances both.

The following code performs these calculations:

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
```

To display the results, the following code prints the accuracy, precision, recall, and F1 score:

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

The confusion matrix provides a more detailed breakdown of the model's predictions. It shows the counts of true positives, true negatives, false positives, and false negatives. In our case, a true positive is an email that was correctly predicted as spam, a true negative is an email that was correctly predicted as not spam, a false positive is an email that was incorrectly predicted as spam but was not spam, and a false negative is an email that was incorrectly predicted as not spam but was actually spam. The steps are as follows:

1. Generate the confusion matrix using `confusion_matrix`, passing `y_test` and `y_pred` as arguments. Assign the result to the variable `cm`.
2. Visualize the confusion matrix using Seaborn's `heatmap` function. Pass the confusion matrix `cm`, set `annot=True` to annotate the heatmap with the actual values, and set `fmt='d'` to display the values as integers.
3. Add a title to the plot, labeling it as "Confusion Matrix"
4. Display the plot using `plt.show()`

The following code generates and visualizes the confusion matrix:

```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d')
plt.title('Confusion Matrix')
plt.show()
```

Run the file. When the script is executed, the data is loaded, the model is trained, and the evaluation metrics are printed. For example, the output may show:

```
Accuracy: 0.91Precision: 0.92Recall: 0.87F1 Score: 0.89
```

The following confusion matrix plot displays the counts of true positives, true negatives, false positives, and false negatives in its four quadrants:

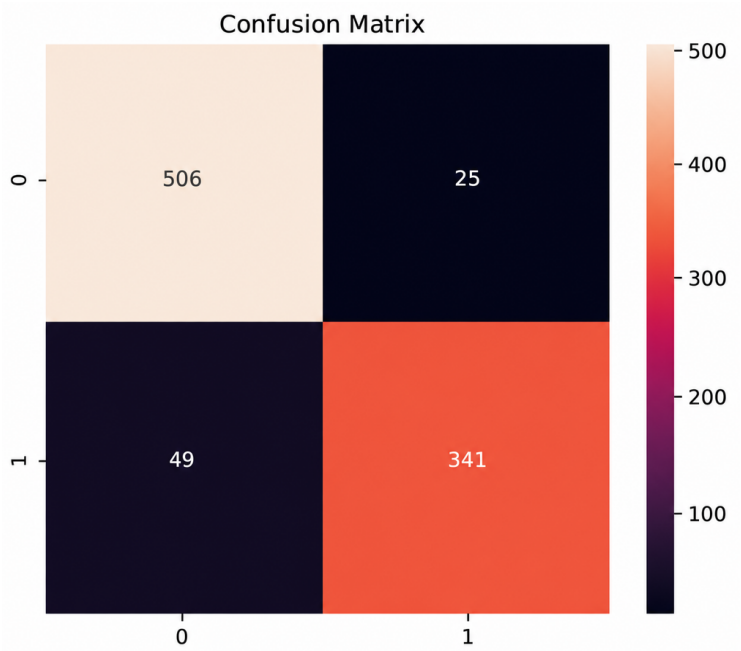


Figure 5.1: Confusion matrix heatmap showing the classification performance of the spam email detector model

The high accuracy and strong precision-recall metrics demonstrate that logistic regression is highly effective for email spam classification.

Machine learning projects

This section applies supervised learning techniques to a range of real-world prediction and classification problems. The projects cover linear regression for house price prediction, logistic regression for diabetes and customer churn prediction, decision trees for flower classification, Random Forest for car price and human activity recognition, XGBoost for employee attrition, and ensemble methods for disease prediction and credit card fraud detection. Each project introduces a distinct dataset and use case, progressively building familiarity with the scikit-learn workflow.

Predicting house prices with linear regression

This project focuses on predicting house prices using **linear regression** with the *California housing dataset*. This dataset contains information about various districts in California, including features such as population, median income, house age, and more.

Loading and preparing the data

Create a Python file and import the necessary libraries to handle data manipulation, model selection, regression, evaluation metrics, and numerical operations. The following code imports the required libraries:

```
import pandas as pd    # Used for data manipulation
from sklearn.datasets import fetch_california_housing  # Imports the California
housing dataset
from sklearn.model_selection import train_test_split  # Splits the dataset into
training and testing sets
from sklearn.linear_model import LinearRegression
# Imports the linear regression model
from sklearn.metrics import mean_squared_error, r2_score  # Metrics to evaluate
model performance
import numpy as np  # For numerical operations
```

Next, the California housing dataset is loaded as a pandas DataFrame. This dataset includes features such as median income, house age, average number of rooms, and others, along with the target variable, which is the median house value (in hundreds of thousands of dollars). The following code loads the dataset and creates a DataFrame:

```
housing = fetch_california_housing(as_frame=True)
df = housing.frame
```

To get an overview of the data, the first five rows of the DataFrame can be displayed:

```
print("California Housing Data:")
print(df.head())
```

The output will be:

Output

```
California Housing Data:

   MedInc  HouseAge  AveRooms  AveBedrms
0   8.3252      41.0  6.984127   1.023810
1   8.3014      21.0  6.238137   0.971880
2   7.2574      52.0  8.288136   1.073446
3   5.6431      52.0  5.817352   1.073059
4   3.8462      52.0  6.281853   1.081081

   Population  AveOccup  Latitude  Longitude  MedHouseVal
0       322.0  2.555556     37.88    -122.23        4.526
1      2401.0  2.109842     37.86    -122.22        3.585
2       496.0  2.802260     37.85    -122.24        3.521
3       558.0  2.547945     37.85    -122.25        3.413
4       565.0  2.181467     37.85    -122.25        3.422

[5 rows x 9 columns; columns shown in two blocks]
```

Figure 5.2: First five rows of the California housing dataset displaying various features

The target variable, `MedHouseVal`, represents the median house value in hundreds of thousands of dollars. For example, a value of 1 means $100,000, and 1.5 means $150,000.

Selecting features and target

The next step is to separate the features (independent variables) and the target (dependent variable). All columns except the last column, `MedHouseVal`, are used as features. The following code separates the features and target:

```
X = df.drop('MedHouseVal', axis=1)  # Features (independent variables)
y = df['MedHouseVal']               # Target variable (dependent variable)
```

Splitting the data into training and test sets

The dataset is then split into training and testing sets. Twenty percent of the data is used for testing, and a random state is set to ensure reproducibility. The following code performs the split:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Training the linear regression model

The linear regression model is initialized and trained using the training data. The following code trains the model:

```
model = LinearRegression()
model.fit(X_train, y_train)
```

The `fit` function trains the model using the training features and target values.

Making predictions about house prices

After training, predictions are made on the test set using the trained model. The following code makes predictions:

```
y_pred = model.predict(X_test)
```

This uses the trained model to predict the median house value for the test set.

Evaluating model performance

The model is evaluated using **mean squared error** and **R-squared (R2) score**. Mean squared error measures the average squared difference between the actual and predicted house values, while R2 score indicates how well the model fits the data. An R2 score closer to 1 indicates better performance. The following code evaluates the model:

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error = {mse}")
print(f"R2 Score = {r2}")
```

The output will be:

```
Mean Squared Error = 0.55
R2 Score = 0.57
```

Displaying model coefficients

The intercept and coefficients of the linear regression model can be displayed. The intercept represents the expected mean value of the target when all features are zero. The coefficients

indicate the effect of each feature on the predicted house value. The following code displays the intercept and coefficients:

```
print(f"Intercept: {model.intercept_}")
print(f"Coefficients: {model.coef_}")
```

Output:

```
Intercept: -37.02327770606416
Coefficients: [ 4.48674910e-01  9.72425752e-03 -1.23323343e-01  7.83144907e-01
 -2.02962058e-06 -3.52631849e-03 -4.19792487e-01 -4.33708065e-01]
```

To make the coefficients more interpretable, a DataFrame can be created that pairs each coefficient with its corresponding feature name. The following code creates and prints the coefficients DataFrame:

```
coef_df = pd.DataFrame(model.coef_, index=X.columns, columns=['Coefficient'])
print("Coefficients for each feature:")
print(coef_df)
```

The output will be:

```
Coefficients for each feature:
              Coefficient
MedInc          0.448675
HouseAge        0.009724
AveRooms       -0.123323
AveBedrms       0.783145
Population     -0.000002
AveOccup       -0.003526
Latitude       -0.419792
Longitude      -0.433708
```

Testing the model with new input

The model can be tested with new data by creating a new DataFrame containing values for each feature. For example, a new house with a median income of 5, house age of 30 years, average number of rooms as 6, average bedrooms as 1, population of 500, average occupancy of 3, latitude of 34.05, and longitude of -118.25.

The following code creates new data and predicts the house price:

```
new_data = pd.DataFrame({
    'MedInc': [5],
    'HouseAge': [30],
    'AveRooms': [6],
    'AveBedrms': [1],
    'Population': [500],
    'AveOccup': [3],
    'Latitude': [34.05],
    'Longitude': [-118.25]
})
predicted_price = model.predict(new_data)
print(f"Predicted house price: ${predicted_price[0]*100000:.2f}")
```

The output came out to be:

```
Predicted house price: $254000.00
```

Tip

Please note that the predicted value may change slightly.

Given the specified features, the predicted house price is approximately $254,000, since the model predicts in units of hundreds of thousands of dollars.

This process demonstrates how to predict house prices using linear regression with the California housing dataset, including data preparation, model training, evaluation, and making predictions with new data.

Predicting diabetes with logistic regression

This project demonstrates how to implement a diabetes prediction model using **logistic regression**. The *Pima Indians Diabetic Dataset* is used for this purpose. This dataset is widely recognized for predicting whether a patient has diabetes based on several medical features such as glucose level, blood pressure, and BMI.

Let's start by creating a Python file and importing the necessary libraries. The following code imports the required libraries for data manipulation, model training, and evaluation:

```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import seaborn as sns
import matplotlib.pyplot as plt
```

To proceed with the project, the *Pima Indians Diabetes Dataset* must be loaded. This dataset is available at a specific URL. The URL for the dataset is:

`https://raw.githubusercontent.com/jbrownlee/datasets/master/pima-indians-diabetes.data.csv`

The relevant column names in the dataset are: pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, diabetes pedigree function, age, and outcome. These columns represent the features and the target variable in the dataset. The following code imports the visualization libraries and loads the dataset from the URL, specifying the column names:

```
# Load the Pima Indians diabetes dataset
url = 'https://raw.githubusercontent.com/jbrownlee/datasets/master/pima-indians-
diabetes.data.csv'

column_names = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']

df = pd.read_csv(url, names=column_names)
```

This code defines the URL for the dataset, specifies the column names, and loads the dataset into a pandas DataFrame named `df`.

Inspecting the diabetes dataset

To display the first few rows of the dataset, the following code is used. This allows inspection of the data structure and the values present in each column:

```
print("Diabetes Dataset:")
print(df.head())
```

The output is as follows:

```
Diabetes Dataset:
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6      148             72             35        0  33.6
1            1       85             66             29        0  26.6
2            8      183             64              0        0  23.3
3            1       89             66             23       94  28.1
4            0      137             40             35      168  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

Figure 5.3: Output of data.head() showing the first few rows of the diabetes dataset with all feature columns and target values

The `Outcome` column contains either 1 or 0, indicating diabetic or non-diabetic status, respectively.

Selecting features and target variables

The next step is to separate the features from the target variable. All columns except for `Outcome` are used as features. The `Outcome` column is used as the target. The following code drops the `Outcome` column from the DataFrame to create the features, and assigns the `Outcome` column to the target variable:

```
X = df.drop('Outcome', axis=1)
y = df['Outcome']
```

Splitting the dataset into training and test sets

To prepare for model training and evaluation, the dataset is split into training and testing sets. The split is performed so that 20 percent of the data is used for testing, and a fixed random state is set for reproducibility.

The following code performs the split and assigns the resulting arrays to variables for training and testing:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Training the logistic regression model

A **logistic regression** model is then created and trained using the training data. The maximum number of iterations for the solver is set to 1000 to ensure convergence.

The following code initializes the model and fits it to the training data:

```
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
```

Making predictions with patient data

After training, predictions are made on the test set using the trained model. The following code generates predictions for the test data:

```
y_pred = model.predict(X_test)
```

Evaluating model performance

The model's performance is evaluated using three metrics: accuracy score, confusion matrix, and classification report. Each metric is calculated and printed. The following code computes and prints the accuracy score:

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

The confusion matrix is calculated and printed as follows:

```
cm = confusion_matrix(y_test, y_pred)
print(f"Confusion matrix: {cm}")
```

The classification report is generated and printed with the following code:

```
class_report = classification_report(y_test, y_pred)
print(f"Classification report: {class_report}")
```

The output of these metrics is as follows:

```
Accuracy: 0.7467532467532467
Confusion matrix: [[78 21]
                   [18 37]]
Classification report:
              precision    recall  f1-score   support

           0       0.81      0.79      0.80        99
           1       0.64      0.67      0.65        55

    accuracy                           0.75       154
   macro avg       0.73      0.73      0.73       154
weighted avg       0.75      0.75      0.75       154
```

Visualizing the confusion matrix

To visualize the confusion matrix, a heatmap is created using seaborn. The heatmap displays the confusion matrix values with annotations, formatted as integers, and uses the `'Blues'` color map. The plot is given a title and labeled axes. The following code generates and displays the confusion matrix heatmap:

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

The plot in *Figure 5.4* shows the confusion matrix for the logistic regression model's predictions on the test set:

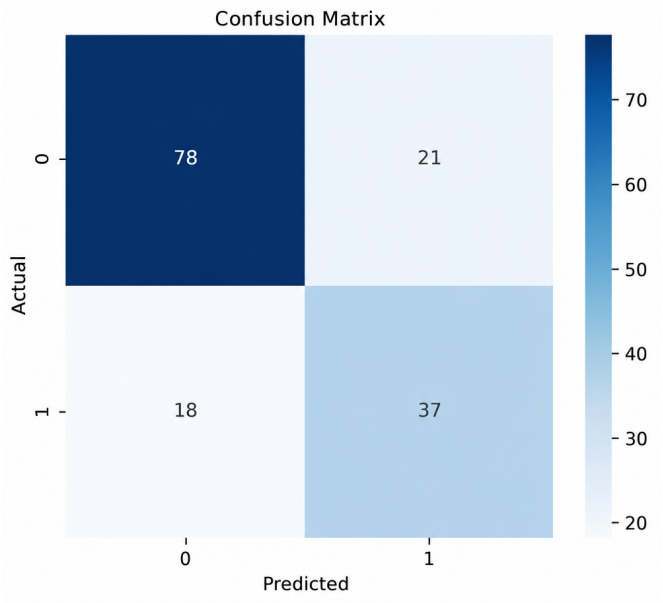


Figure 5.4: Confusion matrix heatmap showing the classification performance of the diabetes prediction model

The confusion matrix reveals that the logistic regression model correctly identifies most non diabetic cases but has room for improvement in detecting true diabetic cases, suggesting the need for model optimization or threshold adjustment.

Testing the model with a new patient record

To test the trained model with new data, a new data point is created in the form of a DataFrame. The required features are set to specific values, such as pregnancies, glucose, blood pressure, skin thickness, insulin, BMI, diabetes pedigree function, and age. The following code creates a new data point with example values:

```
my_data = pd.DataFrame({
    'Pregnancies': [5],
    'Glucose': [120],
    'BloodPressure': [72],
    'SkinThickness': [35],
    'Insulin': [80],
    'BMI': [32.0],
    'DiabetesPedigreeFunction': [0.5],
    'Age': [42]
})
```

This DataFrame can then be used for making predictions with the trained model.

To obtain the predicted outcome for the new data point, the model is asked to predict whether the individual is diabetic or not based on the provided features. The prediction is made using the trained model, and the result is printed. The output is either 0 or 1, where 1 indicates diabetic and 0 indicates non-diabetic.

The following code predicts the outcome for the new data point and prints a message indicating whether the individual is diabetic or non-diabetic:

```
predicted_outcome = model.predict(my_data)
if predicted_outcome[0] == 1:
    print("Diabetic")
else:
    print("Non-diabetic")
```

The output for these features is:

```
Non-diabetic
```

The output displays "Non-diabetic" for the example data provided. If the values of certain features, such as glucose level, are increased, the model may predict "Diabetic" instead. The confusion matrix is also displayed, showing the counts of true negatives, true positives, false negatives, and false positives, along with the predicted outcome for the new data point.

Classifying iris flower dataset using decision trees

This project focuses on classifying the Iris flower dataset using decision trees. The *Iris dataset* is a well-known dataset in machine learning, containing data about the sepal and petal lengths and widths of three types of Iris flowers: Setosa, Versicolor, and Virginica. The goal is to train a decision tree classifier to predict the species of the Iris flower based on these measurements.

Importing libraries for decision tree classification

Start by creating a new Python file and importing the libraries required. We'll use the following libraries:

```
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

Next, the decision tree classifier is imported from scikit-learn. This classifier implements the decision tree algorithm for classification tasks. Additionally, the metrics module from scikit-

learn is imported to provide tools for evaluating the performance of the classifier. The function `plot_tree` is also imported, which is used to visualize the trained decision tree. Finally, `matplotlib.pyplot` is imported as `plt` to enable visualization of the decision tree structure. The following code imports these additional libraries and functions:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn import metrics
import matplotlib.pyplot as plt
```

At this point, all necessary components are available for building and evaluating the decision tree classifier.

The next step is to load and display the Iris dataset.

Loading and preparing the Iris dataset

The process begins by loading the Iris dataset using the `load_iris()` function. This dataset contains features such as sepal length, sepal width, petal length, and petal width. To enable easier manipulation, the dataset is converted into a pandas DataFrame. The target variable, which represents the species of the Iris flower, is then added as a new column named `species`.

The following code loads the Iris dataset, converts it into a DataFrame, and adds the species column:

```
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target
```

To gain an initial understanding of the data, the first few rows of the DataFrame are displayed. The following code prints the first five rows:

```
print("Iris dataset:")
print(df.head())
```

The output will be:

```
Iris dataset:
sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  species
0                5.1               3.5                1.4               0.2
0
1                4.9               3.0                1.4               0.2
0
```

```
2              4.7            3.2            1.3            0.2
0
3              4.6            3.1            1.5            0.2
0
4              5.0            3.6            1.4            0.2
0
```

Splitting features and target species

The next step involves separating the features and the target variable. The features, which include sepal length, sepal width, petal length, and petal width, are stored in the variable `X`. The target variable, `species`, is stored in the variable `y`. The following code separates the features and the target:

```
X = df.drop('species', axis=1)
y = df['species']
```

The `species` column contains the target values, which are encoded as integers representing the three Iris species: Setosa, Versicolor, and Virginica.

Splitting the dataset into training and test sets

To evaluate the performance of the classifier, the dataset is split into training and test sets. The `train_test_split` function is used for this purpose, with 70 percent of the data allocated for training and 30 percent for testing. The `random_state` parameter is set to 42 to ensure reproducibility, so the same split occurs each time the code is run. The following code performs the split:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
```

Training the decision tree classifier

With the data prepared, the next step is to train a decision tree classifier. A variable named `classifier` is created and initialized with the decision tree classifier. The following code trains the decision tree classifier on the training set by calling the `fit` method and passing the training features and target:

```
classifier = DecisionTreeClassifier(random_state=42)
classifier.fit(X_train, y_train)
```

After training the classifier, predictions are made on the test set. The following code predicts the species of the iris flowers in the test set and stores the results in `y_pred`:

```
y_pred = classifier.predict(X_test)
```

Evaluating the model

The model is evaluated using three metrics: accuracy, confusion matrix, and classification report. The following code calculates and prints these metrics:

```
accuracy = metrics.accuracy_score(y_test, y_pred)
conf_matrix = metrics.confusion_matrix(y_test, y_pred)
class_report = metrics.classification_report(y_test, y_pred)
print(f"\nAccuracy: {accuracy*100:.2f}%")
print("\nConfusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(class_report)
```

Output:

```
Accuracy: 100.00%

Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Visualizing the decision tree

The decision tree is visualized using Matplotlib. The figure size is set to 12 x 8. The `plot_tree` function is called with the classifier, feature names, class names, and the `filled` parameter set to `True`. The feature names label the features (sepal length, sepal width, petal length, petal width), and the class names label the target classes (Setosa, Versicolor, Virginica). Setting `filled=True` colors the tree nodes based on the class they predict, providing a visual understanding of the decision process. A title is added to the plot, and the plot is displayed.

The following code visualizes the trained decision tree:

```
plt.figure(figsize=(12, 8))
plot_tree(classifier, feature_names=iris.feature_names,
class_names=iris.target_names, filled=True)
plt.title("Decision Tree for Iris Flower Classification")
plt.show()
```

After completing the code, the next step is to execute it. Running the script displays the decision tree for iris classification.

The resulting visualization is color-coded, with nodes labeled according to the feature being split, such as petal width. The tree indicates true or false branches based on the feature values. This visual output provides a clear representation of how the decision tree classifier makes decisions for the iris dataset.

The plot in *Figure 5.5* shows the color-coded decision tree generated for iris flower classification:

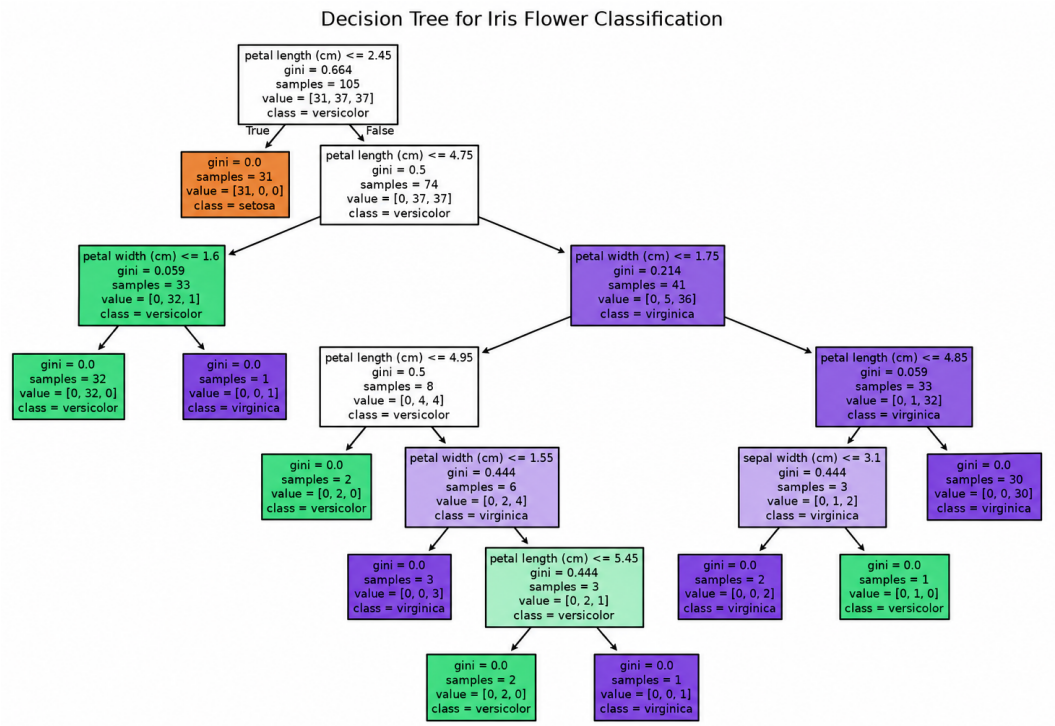


Figure 5.5: Decision tree visualization for Iris flower classification with color-coded nodes representing the three species classes

This demonstrates how the `plot_tree` function can be used to create an informative and visually appealing classifier tree, completing the iris flower classification project using decision trees.

Predicting car prices using random forest

In this project, a car price prediction model is created using **random forest**. The objective is to predict the price of a car based on various features such as make, model, year, and mileage.

Random forest is an **ensemble learning method** that is suitable for regression tasks. It combines multiple decision trees to improve prediction accuracy.

To implement car price prediction using a random forest regressor, the process begins by preparing the working environment. The following steps outline the procedure as demonstrated in the transcript.

Creating a sample car dataset

The first step in the code is to import the necessary libraries. The following code imports these libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

The `RandomForestRegressor` from `sklearn.ensemble` implements the random forest regression model for predicting prices. The `mean_squared_error` and `r2_score` from `sklearn.metrics` are used to evaluate model performance.

Next, a sample dataset is created as a Python dictionary. Each key in the dictionary represents a feature: make, model, year, mileage, and price. Each list contains the data for that particular feature. For example, the first car is a 2015 Toyota Corolla with 50,000 miles and a price of 15,000. The second car is a 2017 Honda Civic with 30,000 miles and a price of 17,000. The following code creates the sample dataset:

```
data = {
    'make': ['Toyota', 'Honda', 'Ford', 'Toyota', 'Honda', 'Ford', 'Toyota',
'Honda', 'Ford', 'Toyota'],
    'model': ['Corolla', 'Civic', 'Focus', 'Camry', 'Accord', 'Fusion', 'Prius',
'Fit', 'Mustang', 'Yaris'],
    'year': [2015, 2017, 2016, 2018, 2015, 2017, 2016, 2018, 2015, 2017],
    'mileage': [50000, 30000, 40000, 20000, 60000, 35000, 45000, 25000, 70000,
15000],
    'price': [15000, 17000, 14000, 20000, 13000, 16000, 15500, 18000, 12000,
21000]
}
```

To convert this dictionary into a DataFrame, the Pandas `DataFrame` function is used. This makes it easier to analyze and manipulate the data. The DataFrame is then printed to display its contents. The following code converts the dictionary to a DataFrame and prints it:

```
df = pd.DataFrame(data)
print("Dataset:\n", df)
```

The output for this will be:

```
Dataset:
      make    model  year  mileage  price
0   Toyota  Corolla  2015    50000  15000
1    Honda    Civic  2017    30000  17000
2     Ford    Focus  2016    40000  14000
3   Toyota    Camry  2018    20000  20000
4    Honda   Accord  2015    60000  13000
5     Ford   Fusion  2017    35000  16000
6   Toyota    Prius  2016    45000  15500
7    Honda      Fit  2018    25000  18000
8     Ford  Mustang  2015    70000  12000
9   Toyota    Yaris  2017    15000  21000
```

Encoding categorical features

The next step is to encode the categorical features using one-hot encoding. Since the 'make' and 'model' columns contain text, they need to be converted into numerical values. This is accomplished using the Pandas `get_dummies()` function, specifying the columns to encode. The resulting DataFrame, `df_encoded`, contains a separate column for each unique value in 'make' and 'model', with a binary indicator: 1 if the value is present, 0 otherwise. The following code performs one-hot encoding and prints the encoded DataFrame:

```
df_encoded = pd.get_dummies(df, columns=['make', 'model'])
print("\nEncoded Dataset:\n", df_encoded)
```

The output is as follows:

```
Encoded Dataset:
    year   mileage   price   make_Ford   make_Honda   make_Toyota   model_Accord  \
0   2015     50000   15000       False        False          True          False
1   2017     30000   17000       False         True         False          False
2   2016     40000   14000        True        False         False          False
3   2018     20000   20000       False        False          True          False
4   2015     60000   13000       False         True         False           True
5   2017     35000   16000        True        False         False          False
6   2016     45000   15500       False        False          True          False
7   2018     25000   18000       False         True         False          False
8   2015     70000   12000        True        False         False          False
9   2017     15000   21000       False        False          True          False

   model_Camry   model_Civic   model_Corolla   model_Fit   model_Focus  \
0        False         False            True       False         False
1        False          True           False       False         False
2        False         False           False       False          True
3         True         False           False       False         False
4        False         False           False       False         False
5        False         False           False       False         False
6        False         False           False       False         False
7        False         False           False        True         False
8        False         False           False       False         False
9        False         False           False       False         False

   model_Fusion   model_Mustang   model_Prius   model_Yaris
0         False           False         False         False
1         False           False         False         False
2         False           False         False         False
3         False           False         False         False
4         False           False         False         False
5          True           False         False         False
6         False           False          True         False
7         False           False         False         False
8         False            True         False         False
9         False           False         False          True
```

Figure 5.6: Output of the encoded car dataset showing feature columns with numerical values representing categorical features after encoding

The features and target variables are then defined. The features, `X`, are all columns except the target variable `'price'`, which is dropped using the `drop` method. The target variable, `y`, is the `'price'` column.

The following code defines the features and target:

```
X = df_encoded.drop('price', axis=1)
y = df_encoded['price']
```

Splitting the dataset into training and test sets

The dataset is then split into training and testing sets using `train_test_split`. The test size is set to 0.2, meaning 20 percent of the data is used for testing. The random state is set to 42 to ensure reproducibility.

The following code splits the data:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Training and evaluating the random forest model

A random forest regressor is then initialized. The number of estimators is set to 100, specifying 100 trees in the forest, which is a typical starting point. The random state is set to 42 for reproducibility. The following code initializes the random forest regressor:

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
```

The model is trained on the training set using the `fit` method, which allows the model to learn the relationship between the features and the target variable. The following code trains the model:

```
model.fit(X_train, y_train)
```

Predictions are made on the test set using the trained model. The predicted values are stored in the `y_pred` variable. The following code makes predictions:

```
y_pred = model.predict(X_test)
```

To evaluate the model, the mean squared error (MSE) and R-squared (R2) score are calculated. The following code evaluates the model and prints the results:

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R2 Score: {r2}")
```

The output produced is:

```
Mean Squared Error: 23447450.0
R2 Score: 0.22487768595041324
```

In this example, the mean squared error is approximately 23 million, which is a high value and indicates that the prediction accuracy is not optimal. The R2 score is approximately 0.22, which is not close to one, indicating that the model does not explain most of the variance in the data. The reason for this performance is that the dataset contains only 10 items. Providing more data would improve the model's performance. This example demonstrates how to use a random forest regressor on a dataset to train a model and make predictions.

Human activity recognition using smartphones dataset with random forest

This project focuses on classifying human activities such as walking, sitting, and standing based on sensor data collected from smartphones. The dataset used is the *Human Activity Recognition (HAR) dataset* from UCI, which contains data from accelerometers and gyroscopes embedded in smartphones. The classification task is performed using a **Random Forest** classifier.

Downloading and preparing the HAR dataset

Download the Human Activity Recognition Using Smartphones dataset from `https://www.kaggle.com/datasets/uciml/human-activity-recognition-with-smartphones`. Download `train.csv` and place it in the working directory. The official UCI source is `https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones`.

Importing libraries for activity recognition

Create a new Python file and begin the code by importing the required libraries. The following code imports the necessary modules for data loading, preprocessing, model building, evaluation, and visualization:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score
import seaborn as sns
import matplotlib.pyplot as plt
```

Tip

The newly introduced libraries are: `RandomForestClassifier` from `sklearn.ensemble` is the classifier used for human activity recognition.

Other libraries are already explained in earlier projects.

The next step is to load the dataset. The following code loads the HAR dataset from the CSV file:

```
df = pd.read_csv('train.csv')
```

This loads the training portion of the Human Activity Recognition dataset. The file contains 561 sensor-derived features, an activity label, and a subject identifier.

Preprocessing the activity dataset

The dataset requires preprocessing before training the model. The first step is to remove any columns that are not needed for the classification task. In this case, the column named `Activity` is dropped from the feature set, leaving only the sensor readings as features.

The following code removes the `Activity` column from the dataset, resulting in a DataFrame containing only the features:

```
X = df.drop(['Activity', 'subject'], axis=1)
```

This operation removes the activity label and subject identifier, ensuring that only sensor-derived measurements are used as input features.

Next, the target variable is defined. The target variable consists of the activity labels, such as walking, sitting, and standing. The following code extracts the `Activity` column as the target variable:

```
y = df['Activity']
```

At this stage, the features are stored in `X`, and the target labels are stored in `y`. The last column in the dataset, `Activity`, serves as the target variable for classification.

Splitting the dataset into training and test sets

The dataset is then split into training and testing sets. This is done to evaluate the model's performance on unseen data. The following code splits the features and target labels into training and test sets, using 80 percent of the data for training and 20 percent for testing:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

The `random_state` parameter is set to 42 to ensure reproducibility of the split. This split ensures that the model is trained on a majority of data and evaluated on a separate subset.

Training the random forest classifier

The next step is to train a **Random Forest classifier** on the training data. The Random Forest is an ensemble method that uses multiple decision trees for classification. The classifier is initialized with 100 trees (`n_estimators=100`) and a fixed random state for reproducibility. The following code creates and trains the Random Forest classifier:

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

This trains the Random Forest model on the training features and labels.

Making predictions on the test set

Once the model is trained, predictions are made on the test set. The following code generates predictions for the test features:

```
y_pred = model.predict(X_test)
```

This produces the predicted activity labels for the test data.

Evaluating the model

The model's performance is evaluated using several metrics: accuracy, precision, recall, and F1 score. The following code computes these metrics using the true and predicted labels:

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="weighted", zero_division=0)
recall = recall_score(y_test, y_pred, average="weighted", zero_division=0)
f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)
```

The accuracy is printed as a percentage for better readability:

```
print(f"Accuracy: {accuracy * 100:.2f}%")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
```

Output:

```
Accuracy: 97.69%
Precision: 0.9769602078562384
Recall: 0.9768864717878993
F1 Score: 0.9768878824556908
```

Visualizing the activity confusion matrix

To further analyze the model's performance, the confusion matrix is visualized using Seaborn's heatmap. The confusion matrix shows the number of correct and incorrect predictions for each activity class.

The following code computes the confusion matrix and displays it as a heatmap:

```
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.show()
```

Run the code using the following command in the terminal:

```
python <your_file_name.py>
```

Replace `<your_file_name.py>` with your file name. The plot in *Figure 5.7* shows the confusion matrix for the activity classification task, with each cell representing the number of predictions for each true and predicted class:

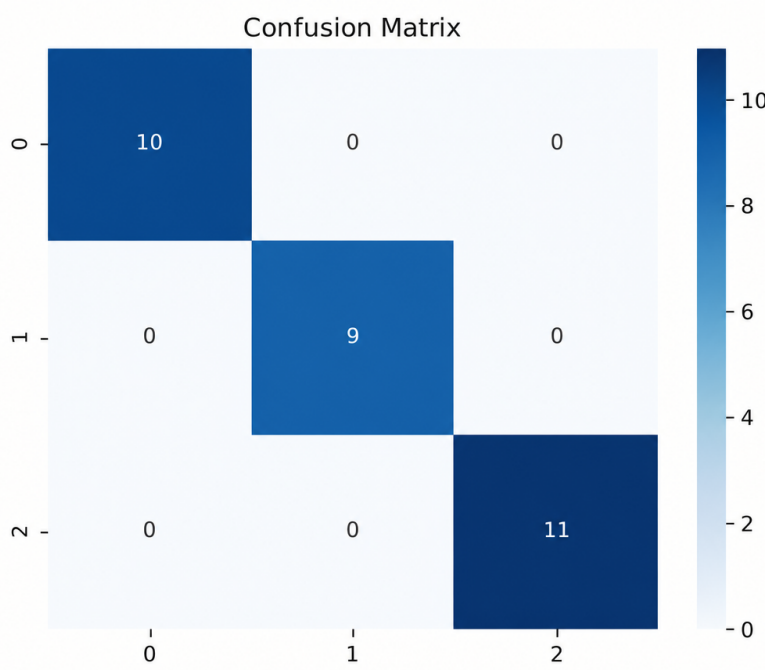


Figure 5.7: Confusion matrix heatmap showing the classification performance of the human activity recognition model across different activity categories

The confusion matrix shows excellent diagonal values, indicating that the random forest classifier accurately recognizes most activities with minimal classifications across all six activity categories.

The high accuracy demonstrates that sensor data from smartphones combined with a random forest classifier can effectively recognize human activities in real-world applications.

Predicting employee attrition using XGBoost

This project focuses on building a predictive model using **XGBoost** to determine whether an employee is likely to leave a company. The model utilizes various job-related features such as job satisfaction, years at the company, salary, and performance ratings.

The dataset used for this project is the *IBM HR Analytics Employee Attrition* dataset, which can be downloaded from Kaggle, available at `https://www.kaggle.com/datasets/pavansubhasht/ibm-hr-analytics-attrition-dataset`. Note that any employee attrition dataset from Kaggle can be used, provided the columns match or are adjusted accordingly.

Start by creating a Python file. The following code imports the necessary libraries for data manipulation, numerical operations, model building, evaluation, and visualization:

```
import pandas as pd
import numpy as np
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt
import seaborn as sns
```

Next, the dataset is loaded. The file path should be replaced with the actual location of the dataset. In this case, the file is renamed to `attrition.csv` and is located in the same directory. The following code loads the dataset and displays the first few rows to preview the data:

```
df = pd.read_csv('attrition.csv')
print("Dataset preview:")
print(df.head())
```

Output:

```
Dataset preview:
   Age Attrition     BusinessTravel  DailyRate              Department  \
0   41       Yes      Travel_Rarely       1102                   Sales
1   49        No  Travel_Frequently        279  Research & Development
2   37       Yes      Travel_Rarely       1373  Research & Development
3   33        No  Travel_Frequently       1392  Research & Development
4   27        No      Travel_Rarely        591  Research & Development

   DistanceFromHome  Education EducationField  EmployeeCount  EmployeeNumber  \
0                 1          2  Life Sciences              1               1
1                 8          1  Life Sciences              1               2
2                 2          2          Other              1               4
3                 3          4  Life Sciences              1               5
4                 2          1        Medical              1               7

   ...  RelationshipSatisfaction StandardHours  StockOptionLevel  \
0  ...                         1           80                 0
1  ...                         4           80                 1
```

```
2  ...                        2            80                 0
3  ...                        3            80                 0
4  ...                        4            80                 1

   TotalWorkingYears  TrainingTimesLastYear WorkLifeBalance  YearsAtCompany  \
0                  8                      0               1               6
1                 10                      3               3              10
2                  7                      3               3               0
3                  8                      3               3               8
4                  6                      3               3               2

  YearsInCurrentRole  YearsSinceLastPromotion  YearsWithCurrManager
0                  4                        0                     5
1                  7                        1                     7
2                  0                        0                     0
3                  7                        3                     0
4                  2                        2                     2

[5 rows x 35 columns]
```

This preview helps to understand the structure and content of the data, which contains many columns—35 in this case.

Preprocessing the attrition dataset

The next step is to preprocess the data. Irrelevant columns that do not affect attrition prediction are dropped. The following code removes the columns `EmployeeNumber`, `Over18`, and `StandardHours`:

```
df.drop(['EmployeeNumber', 'Over18', 'StandardHours'], axis=1, inplace=True)
```

Categorical variables are then encoded into numerical values using `LabelEncoder`. This is necessary because classifiers require numerical input. The following code encodes all categorical features:

```
label_encoder = LabelEncoder()
for column in df.select_dtypes(include=['object']).columns:
    df[column] = label_encoder.fit_transform(df[column])
```

Splitting features and target variable

The features and target variable are separated. All columns except `Attrition` are used as features (`X`), while `Attrition` is the target (`y`). In this dataset, `1` indicates the employee left, and `0` indicates the employee stayed.

The following code performs this separation:

```
X = df.drop('Attrition', axis=1)
y = df['Attrition']
```

Splitting the dataset into training and testing sets

The data is split into training and testing sets using an 80-20 split. The `stratify` parameter ensures class balance is preserved. The following code performs the split:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=45, stratify=y)
```

Training the XGBoost classifier

The **XGBoost classifier** is initialized with `use_label_encoder=False` to disable the deprecated label encoder and `eval_metric='logloss'`. The model is then trained on the training data. The following code initializes and trains the classifier:

```
model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')
model.fit(X_train, y_train)
```

Making predictions and evaluating the model

Predictions are made on the test set, and the model is evaluated using accuracy, confusion matrix, and classification report. The following code makes predictions and evaluates the model:

```
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)
print(f"\nModel accuracy: {accuracy * 100:.2f}%")
print("\nClassification Report:\n", class_report)
print("\nConfusion Matrix:\n", conf_matrix)
```

The output displays the model accuracy, which in this case is 84.01 percent, along with the classification report showing precision, recall, F1 score, and support for each class. The confusion matrix is also displayed, showing true positives, true negatives, false positives, and false negatives:

```
Model accuracy: 84.01%

Classification Report:
              precision    recall    f1-score    support

           0       0.87      0.95        0.91        247
           1       0.50      0.26        0.34         47

    accuracy                           0.84        294
   macro avg       0.69      0.60        0.62        294
weighted avg       0.81      0.84        0.82        294


Confusion Matrix:
 [[235  12]
 [ 35  12]]
```

Figure 5.8: XGBoost model performance metrics including accuracy, classification report, and confusion matrix for employee attrition prediction

The XGBoost model achieves 84.01% accuracy in predicting employee attrition, demonstrating the effectiveness of gradient boosting for HR analytics.

Visualizing the attrition confusion matrix

The confusion matrix is visualized using Seaborn for better readability. The following code creates a heatmap of the confusion matrix:

```
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues',
xticklabels=['Stayed', 'Left'], yticklabels=['Stayed', 'Left'])
plt.title('Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

This visualization provides a clear representation of the model's performance in distinguishing between employees who stayed and those who left.

The script is saved and executed from the command prompt. Upon running the script, the dataset preview is displayed, followed by the model's accuracy, classification report, and the

confusion matrix both as text and as a heatmap. A warning may appear regarding the label encoder, even though `use_label_encoder=False` is specified.

The confusion matrix plot shows the distribution of true positives, true negatives, false positives, and false negatives, providing insight into the model's predictive capabilities:

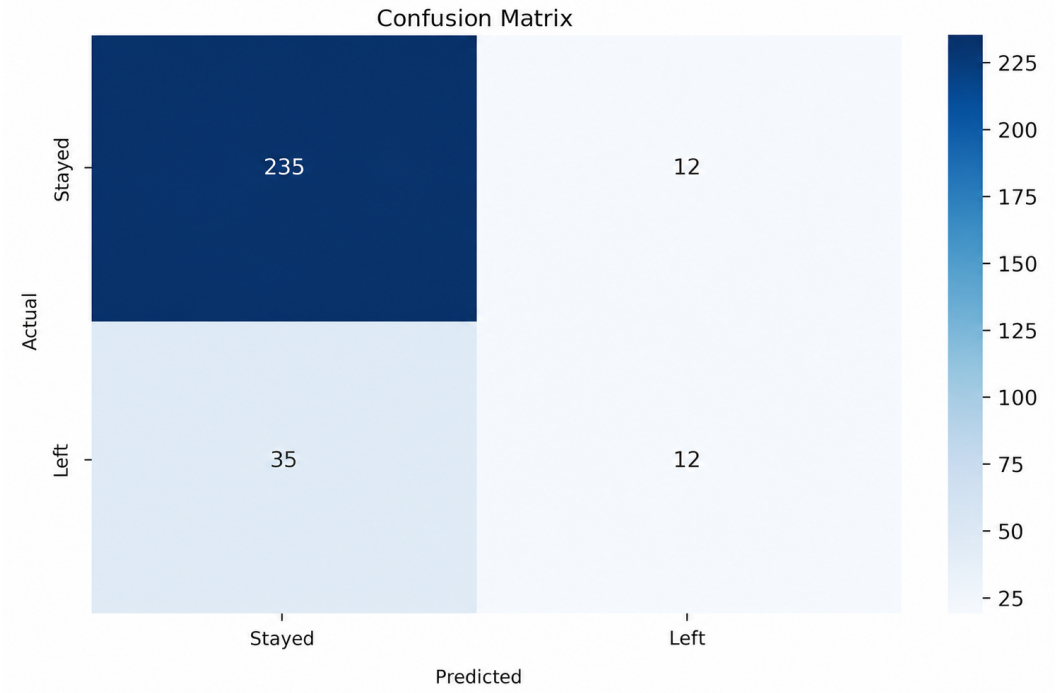


Figure 5.9: Confusion matrix heatmap showing the classification results for employee attrition prediction with true positives, true negatives, false positives, and false negatives

The heatmap visualization provides a clear breakdown of how well the model distinguishes between employees who stayed and those who left.

Predicting customer churn using logistic regression

In this project, a customer churn prediction model is built using a classification algorithm. The example uses **logistic regression** as the classification method.

The objective is to predict whether a customer will churn, meaning leaving the company, based on various features. The model is trained using a dataset containing customer data such as tenure, usage, and demographic information.

The dataset includes the following columns: `customer_id`, `gender`, `age`, `tenure`, `monthly_charges`, `total_charges`, `contract_type`, `payment_method`, and `churn`. The churn

column is either one or zero, indicating whether the customer has left the company. The dataset contains approximately 500 rows.

Importing libraries for churn prediction

The first step is to import the necessary libraries required for building the customer churn prediction model.

The following code imports all the required libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
```

The `StandardScaler` is imported from `sklearn.preprocessing` to scale features so that they have a mean of zero and a standard deviation of one, which helps improve model stability.

Loading and inspecting the churn dataset

Next, the dataset is loaded. The dataset file is named `customer_churn.csv` and is loaded into a data frame named `df`. If a different dataset is used, the file name should be replaced accordingly. The dataset should contain various features related to customer information and a `churn` column indicating churn status.

Dataset source: IBM Telco Customer Churn, available at `https://www.kaggle.com/datasets/abdallahwagih/telco-customer-churn`. Download the CSV and rename it `customer_churn.csv`. The simplified column names used in this example represent a teaching subset, so adapt the column names when using the full IBM dataset. The following code loads the dataset:

```
df = pd.read_csv('customer_churn.csv')
```

To confirm that the dataset has been loaded correctly, the first few rows are displayed using the `head` method:

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

The output is as follows:

```
   CustomerID  Count        Country       State         City  Zip Code  \
0  3668-QPYBK      1  United States  California  Los Angeles     90003
1  9237-HQITU      1  United States  California  Los Angeles     90005
2  9305-CDSKC      1  United States  California  Los Angeles     90006
3  7892-POOKP      1  United States  California  Los Angeles     90010
4  0280-XJGEX      1  United States  California  Los Angeles     90015

                Lat Long   Latitude   Longitude  Gender  ...        Contract  \
0  33.964131, -118.272783  33.964131 -118.272783    Male  ...  Month-to-month
1   34.059281, -118.30742  34.059281 -118.307420  Female  ...  Month-to-month
2  34.048013, -118.293953  34.048013 -118.293953  Female  ...  Month-to-month
3  34.062125, -118.315709  34.062125 -118.315709  Female  ...  Month-to-month
4  34.039224, -118.266293  34.039224 -118.266293    Male  ...  Month-to-month

  Paperless Billing             Payment Method  Monthly Charges Total Charges  \
0               Yes               Mailed check            53.85        108.15
1               Yes           Electronic check            70.70        151.65
2               Yes           Electronic check            99.65         820.5
3               Yes           Electronic check           104.80       3046.05
4               Yes  Bank transfer (automatic)           103.70        5036.3

  Churn Label Churn Value Churn Score  CLTV                   Churn Reason
0         Yes           1        86.0  3239   Competitor made better offer
1         Yes           1        67.0  2701                          Moved
2         Yes           1        86.0  5372                          Moved
3         Yes           1        84.0  5003                          Moved
4         Yes           1        89.0  5340  Competitor had better devices

[5 rows x 33 columns]
```

Basic data preprocessing is performed to handle missing data. Any rows with missing values are removed to ensure data integrity. This is done using the `dropna` method:

```
df['Total Charges'] = pd.to_numeric(df['Total Charges'], errors='coerce')

df = df.dropna(subset=['Total Charges']).copy()

print('Dataset shape after cleaning:', df.shape)
```

Further preprocessing may be necessary depending on the characteristics of the dataset, but this step ensures that there are no missing values.

Preprocessing and encoding categorical features

Next, categorical columns are converted to numeric using one-hot encoding. This is achieved using the `pd.get_dummies()` function, which is applied to the columns `gender`, `contract`, and `payment_method`. This step converts non-numeric categorical columns into numeric columns:

```
df_encoded = pd.get_dummies(df, columns=[ 'Gender', 'Contract', 'Payment Method'])
```

The features and target variable are then defined. The features are all columns except for `churn`, which is dropped from the data frame. The target variable is the `churn` column.

```
columns_to_drop = [
    'CustomerID',
    'Churn Label',
    'Churn Value',
    'Churn Score',
    'Churn Reason'
]

X = df.drop(columns=columns_to_drop)
y = df['Churn Value']
```

Splitting the dataset into training and test sets

The next step is to split the dataset into training and testing sets. The `train_test_split` function is used, with 20 percent of the data allocated for testing by setting `test_size=0.2`. The `stratify=y` parameter ensures that both classes, churned, and non-churned, are proportionally represented in the training and testing sets. The `random_state=42` parameter sets a random seed for reproducibility:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)
```

Scaling features and training the model

Feature scaling is performed for numerical stability. The `StandardScaler` is used to standardize features so that they have a mean of zero and a standard deviation of one. The scaler is fit on the training data and then applied to both the training and test sets.

The logistic regression classifier is then initialized. Logistic regression is commonly used for binary classification tasks such as churn prediction. The code is as follows:

```
numerical_columns = X.select_dtypes(include=['number']).columns.tolist()
categorical_columns = X.select_dtypes(include=['object', 'category',
'bool']).columns.tolist()

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_columns),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_columns)
    ]
)

model = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('classifier', LogisticRegression(max_iter=1000))
    ]
```

The model is trained using the `fit` method, which learns the relationship between the features and the churn status from the training data:

```
model.fit(X_train, y_train)
```

Predictions are made on the test set using the trained model. The `predict` method is used to classify whether each customer in the test set has churned or not:

```
y_pred = model.predict(X_test)
```

Evaluating the model

Model evaluation is performed using the accuracy score and classification report. The accuracy score calculates the percentage of correctly classified customers in the test set. The classification report includes precision, recall, and F1 score for each class, with `zero_division=1` to handle any divisions by zero:

```
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, zero_division=1)
print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

The output displays the accuracy score and the detailed classification report:

```
Accuracy: 0.69
Classification Report:
              precision    recall  f1-score   support

           0       0.75      0.80      0.77        60
           1       0.60      0.53      0.56        30

    accuracy                           0.69        90
   macro avg       0.67      0.66      0.67        90
weighted avg       0.69      0.69      0.69        90
```

The accuracy value shows the overall percentage of correctly classified samples, while the classification report provides metrics such as precision, recall, and F1 score for each class.

This output confirms that the data has been loaded and processed as expected. This approach demonstrates how to predict customer churn using logistic regression.

Detecting credit card fraud using scikit-learn

This project focuses on building a **credit card fraud detection model** using scikit-Learn. The objective is to classify credit card transactions as fraudulent or non-fraudulent based on features such as transaction amount, time, and other numerical data. The model is trained using a widely used dataset for fraud detection and is evaluated based on accuracy and precision.

For this project, use the Credit Card Fraud Detection dataset from `https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud`. Download `creditcard.csv` and place it in the working directory. The dataset contains 284,807 transactions, including 492 fraud cases, and the `Class` column identifies fraudulent transactions.

Importing libraries for fraud detection

Create a new Python file and import the following libraries introduced in earlier projects; these are used for data handling, preprocessing, model training, and evaluation:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score,
confusion_matrix
```

Loading and preparing the fraud dataset

The following code loads the dataset from a CSV file into a DataFrame. The file path can be replaced with the path to any preferred dataset:

```
df = pd.read_csv('creditcard.csv')
```

This line reads the dataset and stores it in the DataFrame `df`. The dataset contains multiple columns, with the `Class` column indicating whether each transaction is fraudulent.

The following code displays the first five rows of the dataset to provide a sample of the records:

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

Next, the features and target variable are separated. The features are all columns except the `Class` column, while the target variable is the `Class` column itself, which indicates whether a transaction is fraudulent (1) or not fraudulent (0):

```
X = df.drop('Class', axis=1)
y = df['Class']
```

Splitting and scaling the dataset

The dataset is then split into training and testing sets. The following code uses the `train_test_split` function to allocate 80% of the data for training and 20% for testing. The `stratify=y` argument ensures that both classes are proportionally represented in both the training and test sets. The `random_state=42` argument sets a random seed for reproducibility:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

To improve numerical stability, the features are scaled using the `StandardScaler`. This standardizes the features so that they have a mean of 0 and a standard deviation of 1:

```
scaler = StandardScaler()
```

The following code fits the scaler to the training data and transforms it, assigning the result to `X_train`:

```
X_train = scaler.fit_transform(X_train)
```

This operation fits the scaler on `X_train` and then transforms `X_train` using the calculated scaling parameters.

Next, the test data is transformed using the scaling parameters obtained from the training data. The following code performs this transformation:

```
X_test = scaler.transform(X_test)
```

This ensures that the test data is scaled in the same way as the training data, using the mean and standard deviation calculated from the training set.

Initializing and training the random forest classifier

The next step is to initialize the `RandomForestClassifier`. The following code creates a random forest classifier with 100 trees and a random state of 42:

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
```

This initializes the random forest classifier with the specified number of trees. The model is then trained on the training data by calling the `fit` function and passing in `X_train` and `y_train`:

```
model.fit(X_train, y_train)
```

This trains the model using the scaled training data and the corresponding target values.

The next step is to make predictions on the test set using the trained model. The following code uses the predict function of the model to generate predicted labels for the test data:

```
y_pred = model.predict(X_test)
```

This function applies the trained model to `X_test` and returns the predicted class labels, which are stored in `y_pred`.

Evaluating the model

To evaluate the model's performance, several metrics are calculated. First, the overall accuracy of the model is computed using the `accuracy_score` function. This metric indicates the proportion of correct predictions made by the model:

```
accuracy = accuracy_score(y_test, y_pred)
```

Next, a classification report is generated. This report provides detailed metrics such as precision, recall, and F1 score for each class (fraud and non-fraud):

```
report = classification_report(y_test, y_pred)
```

Finally, a confusion matrix is created to show the counts of true positives, false positives, true negatives, and false negatives:

```
matrix = confusion_matrix(y_test, y_pred)
```

The confusion matrix is not plotted in this example. If a plot is desired, libraries such as matplotlib or seaborn can be used to visualize the matrix.

The following code prints the accuracy, classification report, and confusion matrix:

```
print(f"\nAccuracy: {accuracy}")
print(f"\nClassification Report:\n {report}")
print(f"\nConfusion Matrix:\n {matrix}")
```

A representative run with `random_state=42` produces output similar to the following. Exact values may vary slightly by scikit-learn version.

Output:

```
Accuracy: 0.9995962220427653

Classification Report:
               precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.94      0.82      0.87        98

    accuracy                           1.00     56962
   macro avg       0.97      0.91      0.94     56962
weighted avg       1.00      1.00      1.00     56962


Confusion Matrix:
 [[56859     5]
 [   18    80]]
```

When this code is executed on the large dataset, the output first displays the head of the data (the first five rows). Since the dataset contains approximately 300,000 rows, the process may take some time to complete.

The accuracy is above 99.9%, but accuracy alone is not sufficient because fraud cases are rare. Review the fraud-class recall, F1 score, and confusion matrix to understand how many fraudulent transactions are missed and how many legitimate transactions are flagged.

This completes the project for credit card fraud detection using Scikit-Learn. The model is able to make highly accurate predictions on the provided credit card transaction data.

Disease prediction using machine learning algorithms

This project is focused on building a machine learning model to predict the likelihood of heart disease based on patient health data. The model uses supervised learning algorithms to analyze key health indicators and classify whether a patient is at risk of heart disease.

The steps followed in this project are:

1. Load and explore the dataset
2. Pre-process the data
3. Split the data into training and testing sets
4. Train the machine learning models
5. Evaluate model performance and make predictions on new data
6. Visualize the results

This project uses the processed Cleveland Heart Disease dataset. The official source is `https://archive.ics.uci.edu/dataset/45/heart+disease`. The prepared file used in this example is saved as `heartdisease.csv` and contains 13 input features plus a binary target column.

Loading and exploring the heart disease dataset

As usual, start by creating a new Python file. The following code imports the required libraries for data manipulation, numerical computations, model building, evaluation, and visualization:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt
```

```
import seaborn as sns
import warnings

warnings.filterwarnings('ignore')
```

The `warnings` library will help ignore the unnecessary warnings that we may get for this project. Next, the dataset is loaded using the following code:

```
df = pd.read_csv('heartdisease.csv')
```

This loads the CSV file into a DataFrame. If the file is located elsewhere or accessed via a URL, the file path should be adjusted accordingly.

To display the first few rows of the dataset, the following code is used:

```
print("Dataset sample:")
print(df.head())
```

Output:

```
Dataset sample:
   age  sex  cp  trtbps  chol  fbs  restecg  thalachh  exng  oldpeak  slp  \
0   63    1   3     145   233    1        0       150     0      2.3    0
1   37    1   2     130   250    0        1       187     0      3.5    0
2   41    0   1     130   204    0        0       172     0      1.4    2
3   56    1   1     120   236    0        1       178     0      0.8    2
4   57    0   0     120   354    0        1       163     1      0.6    2

   caa  thall  output
0    0      1       1
1    0      2       1
2    0      2       1
3    0      2       1
4    0      2       1
```

Preprocessing the heart disease dataset

Missing values are checked using the following code:

```
print("Missing values:")
print(df.isnull().sum())
```

This identifies and counts any missing or null values in each column.

Feature scaling is performed using `StandardScaler` to standardize features with mean 0 and standard deviation 1. The target column is dropped before scaling. The following code initializes the scaler and scales the features:

```
scaler = StandardScaler()
scaled_features = scaler.fit_transform(df.drop('target', axis=1))
```

The scaled features are stored in a new DataFrame, and the target variable is separated:

```
X = pd.DataFrame(scaled_features, columns=df.columns[:-1])
y = df['target']
```

Splitting the dataset into training and test sets

The data is split into training and testing sets using an 80-20 split. The `random_state` parameter ensures reproducibility and `stratify=y` maintains class proportions. The following code performs the split:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

Training multiple machine learning models

Two models are trained: **Logistic Regression** and **Random Forest Classifier**. For logistic regression, the following code initializes, trains, and predicts outcomes:

```
log_model = LogisticRegression(max_iter=1000, random_state=42)
log_model.fit(X_train, y_train)
log_preds = log_model.predict(X_test)
log_accuracy = accuracy_score(y_test, log_preds)
print(f"Logistic Regression Accuracy: {log_accuracy:.2f}")
```

For the random forest classifier, the following code is used:

```
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_preds = rf_model.predict(X_test)
rf_accuracy = accuracy_score(y_test, rf_preds)
print(f"Random Forest Accuracy: {rf_accuracy:.2f}")
```

Model selection and evaluation

The model with higher accuracy is selected as the best model. The following code selects the best model and predictions:

```
best_model = rf_model if rf_accuracy > log_accuracy else log_model
best_preds = rf_preds if rf_accuracy > log_accuracy else log_preds
```

The best model's metrics are displayed using the following code:

```
print("Best Model Metrics:")
print(f"Accuracy Score: {accuracy_score(y_test, best_preds)}")
print("Classification Report:")
print(classification_report(y_test, best_preds))
print("Confusion Matrix:")
print(confusion_matrix(y_test, best_preds))
```

Output:

```
Best Model Metrics:
Accuracy Score: 0.8852459016393442
Classification Report:
              precision    recall  f1-score   support

           0       0.89      0.86      0.88        29
           1       0.88      0.91      0.89        32

    accuracy                           0.89        61
   macro avg       0.89      0.88      0.88        61
weighted avg       0.89      0.89      0.89        61

Confusion Matrix:
[[25  4]
 [ 3 29]]
```

The classification report provides precision, recall, F1 score, and support, while the confusion matrix displays true negatives, true positives, false negatives, and false positives.

Visualizing the heart disease confusion matrix

The confusion matrix is visualized as a heatmap using Seaborn. The following code creates the plot:

```
plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, best_preds), annot=True, cmap='Blues',
fmt='d')
plt.title('Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

The output is:

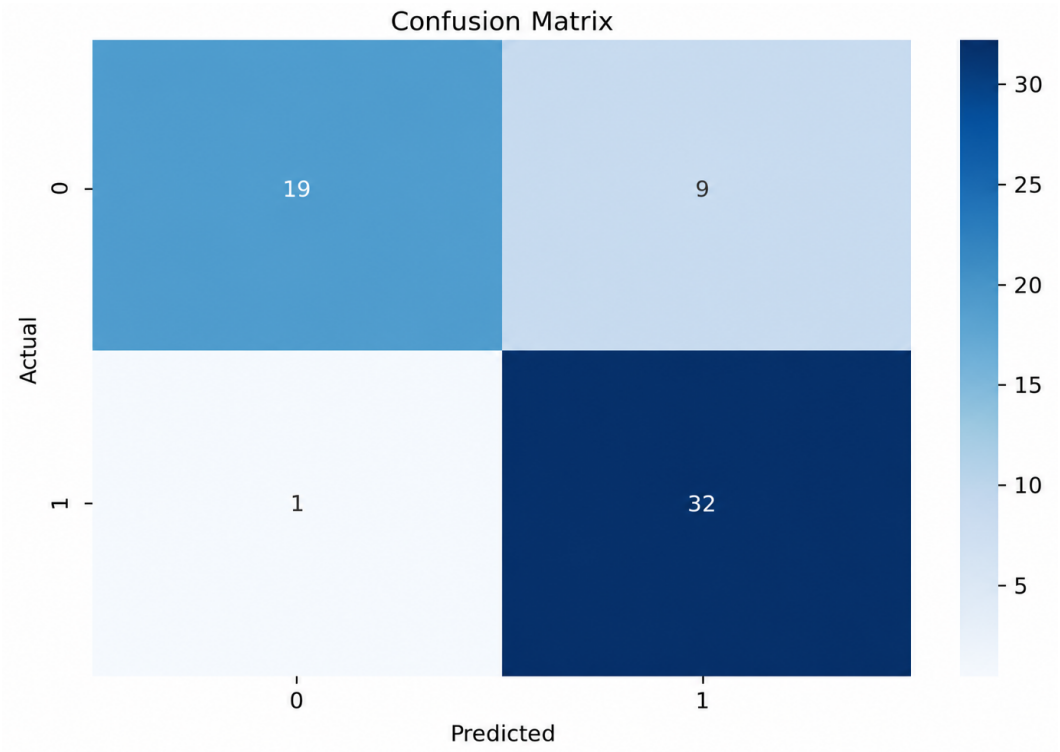


Figure 5.10: Confusion matrix heatmap showing the classification performance of the heart disease risk prediction model

This visualization annotates the matrix with values and uses a blue color map for clarity.

Predicting heart disease risk in new patient data

Predictions are made on a new data sample. A new DataFrame is created with random values for the features. The new data is scaled using the previously fitted scaler, and the best model is used to predict the outcome.

The following code creates a new data sample, scales it, and makes a prediction:

```
new_data = pd.DataFrame({
    'age': [54],
    'sex': [1],
    'cp': [0],
    'trestbps': [130],
    'chol': [230],
    'fbs': [0],
    'restecg': [1],
    'thalach': [150],
    'exang': [0],
    'oldpeak': [0.5],
    'slope': [2],
    'ca': [0],
    'thal': [2]
})
new_data_scaled = scaler.transform(new_data)
prediction = best_model.predict(new_data_scaled)
print("Prediction for new data:", "At risk of heart disease" if prediction[0] ==
1 else "No Heart Disease")
```

Output:

```
Prediction for new data: At risk of heart disease
```

The output indicates whether the new data sample is at risk of heart disease or not, based on the model's prediction.

Upon running the script, the confusion matrix is displayed, showing the distribution of true positives and true negatives. The prediction for the new data sample is also printed, indicating the likelihood of heart disease based on the values provided.

The dataset sample and missing-value counts are displayed first. In this run, logistic regression achieves an accuracy of 0.80 and random forest achieves 0.84, so the random forest model is

selected. The classification report, confusion matrix, and prediction for the new patient record are then displayed.

Forecasting and prediction projects

This section focuses on time-series and sequential prediction tasks using linear regression. The two projects demonstrate how to predict next-day temperature from historical weather data and how to forecast future stock prices from historical closing prices. Both projects follow the same core workflow: preparing features, splitting data, training, evaluating, and visualizing, which reinforces the pattern established in the machine learning section while introducing date-based feature engineering.

Weather forecasting using historical data

This project demonstrates how to build a weather forecasting model using historical data. The objective is to predict the temperature or weather conditions for the next day based on previous weather data. Linear regression is used for this prediction task.

Importing libraries for weather forecasting

Start by creating a new Python file. The following code imports the necessary libraries for handling the data, splitting it into training and test sets, building the linear regression model, evaluating its performance, and visualizing the results:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
```

Preparing the historical weather data

A sample historical weather dataset is used, which includes the following columns: day (from 1 to 10), temperature (ranging from 30 to 31 degrees), humidity, wind speed, precipitation, and next day temperature. The data is structured as a dictionary and then converted into a pandas DataFrame. The following code creates the DataFrame from the sample data:

```
data = {
    'Day': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'Temperature': [30, 31, 29, 30, 31, 30, 29, 30, 31, 30],
    'Humidity': [70, 65, 72, 68, 66, 70, 71, 69, 67, 68],
    'WindSpeed': [12, 14, 13, 15, 12, 13, 14, 12, 13, 15],
    'Precipitation': [0, 1, 0, 2, 1, 0, 1, 0, 2, 1],
```

```
    'NextDayTemperature': [31, 29, 30, 31, 30, 29, 30, 31, 30, 29]
}
df = pd.DataFrame(data)
```

Selecting features and target variables

The features selected for the model are temperature, humidity, wind speed, and precipitation for a particular day. The target variable is the next day's temperature. The following code selects the features and target variable from the DataFrame:

```
X = df[['Temperature', 'Humidity', 'WindSpeed', 'Precipitation']]
y = df['NextDayTemperature']
```

In this setup, `X` represents the independent variables (features), which include temperature, humidity, wind speed, and precipitation for a given day. `y` is the dependent variable (target), which is the temperature for the next day.

Splitting the data into training and test sets

The next step is to split the dataset into training and test sets. This is done so that the model can be trained on one portion of the data and evaluated on another, unseen portion to assess its performance. In this case, 80 percent of the data is used for training and 20 percent for testing. A random state of 42 is specified to ensure reproducibility of the split. The following code splits the features and target variable into training and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Training the linear regression model

After splitting the data, the linear regression model is initialized and trained using the training data. The model is first created by initializing an instance of the LinearRegression class. Then, the model is trained by fitting it to the training features and target variable. The following code initializes and trains the linear regression model:

```
model = LinearRegression()
model.fit(X_train, y_train)
```

This trains the model using the training data, `X_train` and `y_train`.

Making predictions with the trained model

The next step in the process is to use the trained model to make predictions. The model's `predict` method is called with the test feature set to generate predicted values for the next day's temperature. The following code uses the trained model to predict the next day's temperature for the test set and saves the predictions in `y_pred`:

```
y_pred = model.predict(X_test)
```

Evaluating model performance

To evaluate the performance of the model, two metrics are used: **mean squared error** and **R2 score**. These metrics are imported earlier in the workflow. The following code calculates the mean squared error and R2 score by comparing the actual and predicted values for the test set:

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

As discussed earlier, the **mean squared error** measures the average squared difference between the actual and predicted temperatures. A lower mean squared error indicates better model performance. The **R2 score** indicates how well the model fits the data, with a value closer to one representing better performance. These two metrics provide different perspectives on the model's accuracy and fit.

Visualizing actual and predicted temperatures

To visualize the results, the data is plotted using matplotlib. A figure is created with a width of 10 and a height of 6 to provide a clear view of the comparison between actual and predicted temperatures.

The following code creates a plot that displays both the actual temperatures from the test set and the predicted temperatures from the model. The actual temperatures are marked with circles, while the predicted temperatures are marked with x's for easy differentiation. The plot is given a title, and both axes are labeled appropriately. A legend is added to distinguish between the two lines, and the plot is displayed:

```
plt.figure(figsize=(10, 6))
plt.plot(y_test.values, label="Actual Temperatures", marker="o")
plt.plot(y_pred, label="Predicted Temperatures", marker="x")
plt.title("Actual vs Predicted Temperatures")
plt.xlabel("Test Sample Index")
plt.ylabel("Temperature")
```

```
plt.legend()
plt.show()
```

The plot in *Figure 5.11* shows the actual temperatures and the predicted temperatures for the test samples, allowing for a visual assessment of the model's performance:

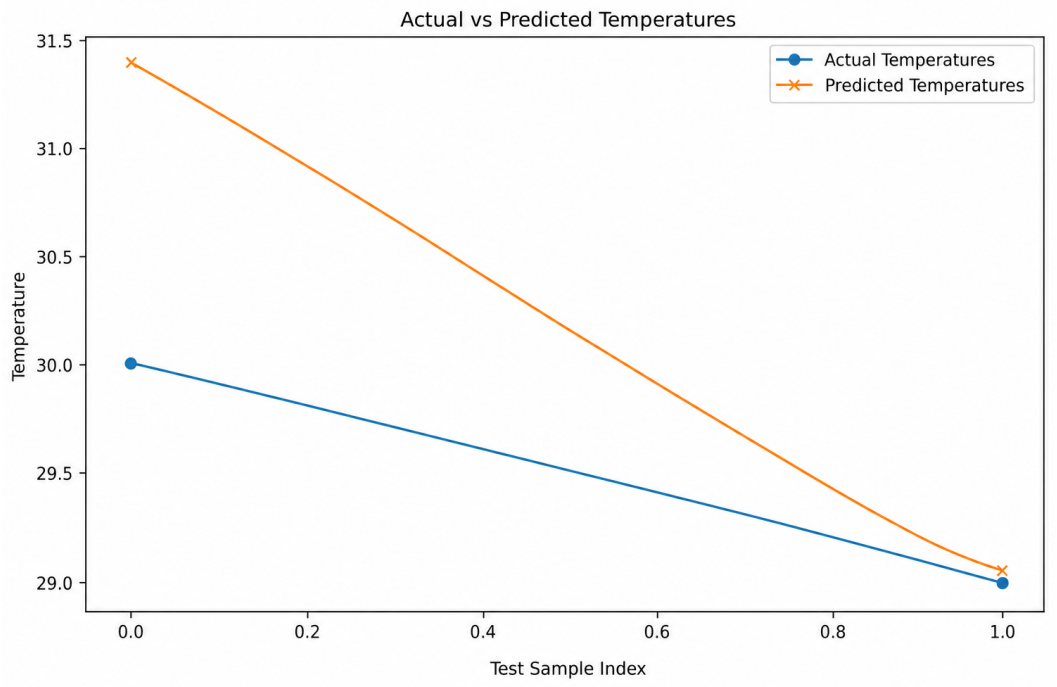


Figure 5.11: Line plot comparing actual temperatures and predicted temperatures from the weather forecasting model, with actual values marked as circles and predicted values marked as x's

The predicted temperatures closely track the actual values, demonstrating that linear regression effectively captures temperature trends from historical weather data.

Predicting temperature for new data

To predict the temperature for a new set of weather conditions, a new data object is created as a pandas DataFrame. This new data includes values for temperature, humidity, wind speed, and precipitation. The model's `predict` method is then used to estimate the next day's temperature based on these inputs.

The following code creates a new data sample and uses the trained model to predict the temperature for the next day. The predicted value is printed in Celsius, formatted to two decimal places:

```
new_data = pd.DataFrame({
    "Temperature": [30],
    "Humidity": [60],
    "WindSpeed": [10],
    "Precipitation": [0]
})
predicted_temperature = model.predict(new_data)
print(f"\n\nPredicted temperature: {predicted_temperature[0]:.2f} degrees
Celsius")
```

The output is:

```
Predicted temperature for the next day is 31.70 degrees Celsius
```

When this code is executed, the plot of actual versus predicted temperatures is displayed first. After closing the plot, the predicted temperature for the new data is printed, showing the model's estimate for the next day's temperature based on the provided weather conditions.

Predicting stock prices with linear regression

This project focuses on implementing stock price prediction using historical data and linear regression. The approach uses past stock prices to predict the future price of a stock. The dataset for this project contains dates and closing prices, and linear regression is applied to forecast future stock prices.

Start by creating a new Python file and importing the necessary libraries for the project:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt  # Used for plotting the actual and predicted
stock prices
```

Creating a sample historical stock price dataset

A sample historical stock price dataset is created for demonstration purposes. This dataset can be replaced with actual stock prices obtained from sources such as the Yahoo Finance API or Alpha Vantage. In this example, random data is used to illustrate the process.

The following code generates a date range for 10 consecutive days starting from January 1, 2024, and assigns a set of closing prices to these dates:

```
# Generate a date range for 10 consecutive days starting from 2024-01-01
data = {
    'Date': pd.date_range(start='2024-01-01', periods=10, freq='D'),
    'Close': [150, 152, 153, 155, 154, 156, 157, 158, 159, 160]
}
df = pd.DataFrame(data)
```

The next step is to convert the date column to a numerical format suitable for regression analysis. The `pd.Timestamp.toordinal` function is used to transform the dates into ordinal format, which represents the number of days since a fixed point in history. This transformation is necessary because the linear regression model requires numerical input. The following code converts the date column to ordinal format:

```
# Convert the date column to ordinal format for regression analysis
df["Date"] = df["Date"].map(pd.Timestamp.toordinal)
```

To verify that the dataset has been created and transformed correctly, the first few rows are displayed:

```
# Display the first few rows of the dataset to check correctness
print("Stock Price Data:")
print(df.head())
```

The output will be:

```
Stock Price Data:
     Date  Close
0  738886    150
1  738887    152
2  738888    153
3  738889    155
4  738890    154
```

Preparing features and target variables

The next step is to define the features and target variables for the regression model. The `Date` column is used as the feature, and the closing price (`Close`) is used as the target. Additional

variables can be added as features if desired, but in this example, only the date is used. The following code assigns the features and target variables:

```
# Define features (date) and target (closing price)
X = df[["Date"]]
y = df["Close"]
```

Splitting the dataset into training and test sets

To evaluate the performance of the linear regression model, the dataset is split into training and testing sets. The `train_test_split` function is used for this purpose, with 80 percent of the data used for training and 20 percent for testing. A random state of 42 is specified for reproducibility. The following code performs the split:

```
# Split the dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Training the linear regression model

The next step is to train the **linear regression model**. The model is initialized and then `fit` to the training data using the fit method. This trains the model on the features and target variables defined earlier.

The following code initializes the linear regression model and fits it to the training data:

```
# Initialize the linear regression model
model = LinearRegression()

# Train the model using the training data
model.fit(X_train, y_train)
```

Predicting closing prices on test data

After the linear regression model is trained, predictions can be made on the test dataset. The `predict` method is used for this purpose, taking the test features as input and returning the predicted values. The following code generates predictions using the trained model:

```
# Make predictions on the test dataset
y_pred = model.predict(X_test)
```

Evaluating model performance

To assess the performance of the trained linear regression model, two metrics are calculated: **mean squared error (MSE)** and **R2 score**. The following code calculates the mean squared error and R2 score using the test data and the model's predictions:

```
# Calculate mean squared error
mse = mean_squared_error(y_test, y_pred)

# Calculate R2 score
r2 = r2_score(y_test, y_pred)
```

The next step is to print the calculated metrics to evaluate how well the model performed:

```
# Print the evaluation metrics
print(f"Mean Squared Error: {mse}")
print(f"R2 Score: {r2}")
```

Output:

```
Mean Squared Error: 0.06023335316288063
R2 Score: 0.9950829915785404
```

Visualizing actual vs. predicted stock prices

To better understand the model's predictions, a plot is created to compare the actual stock prices with the predicted values. The plot uses the test data for both the actual and predicted stock prices, with different markers for each series.

The following code generates the plot using Matplotlib, setting the figure size, plotting both the actual and predicted values, and labeling the axes and legend:

```
# Create a figure with a fixed size
plt.figure(figsize=(10, 6))

# Plot actual stock prices
plt.plot(X_test, y_test, label="Actual stock prices", marker="o")

# Plot predicted stock prices
plt.plot(X_test, y_pred, label="Predicted stock prices", marker="x")

# Set the title of the plot
```

```
plt.title("Actual vs Predicted Stock Prices")

# Label the x-axis (date in ordinal format)
plt.xlabel("Date (Ordinal)")

# Label the y-axis (stock price in dollars)
plt.ylabel("Stock Price ($)")

# Display the legend
plt.legend()

# Show the plot
plt.show()
```

The resulting plot displays the actual stock prices and the predicted stock prices for the test data, allowing for a visual comparison of the model's performance:

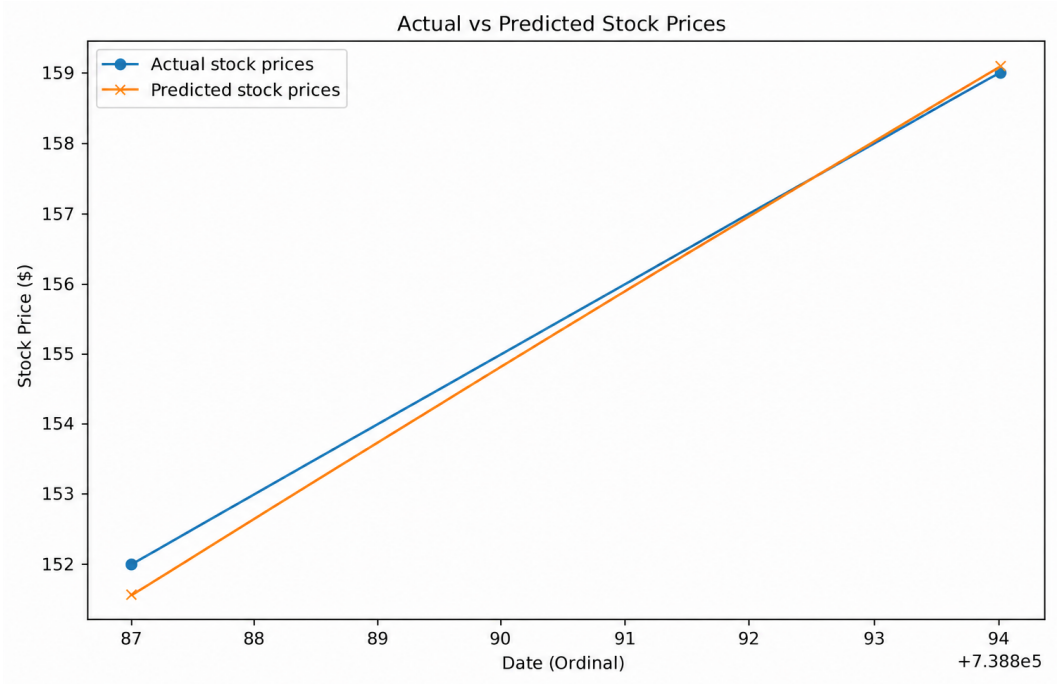


Figure 5.12: Visualization of actual versus predicted stock prices from the linear regression forecasting model

The actual and predicted stock prices align closely throughout the test period, confirming the linear regression model's strong forecasting capability.

Predicting stock price for a future date

To check the model's prediction for a new data point or a future date, a specific timestamp can be selected. For example, to predict the stock price on the 11th day after the existing data, the following steps are performed:

1. Assign a value to represent the future date. In this case, the date is set to January 11, 2024.
2. Create a DataFrame with the feature name 'date' and assign the future date value to it
3. Use the trained model to predict the stock price for this future date
4. Print the predicted stock price, formatting the output to two decimal places

The following code demonstrates these steps:

```
# Step 1: Assign the future date (e.g., January 11, 2024)
future_date = pd.Timestamp('2024-01-11').toordinal()

# Step 2: Create a DataFrame with the feature name 'date'
future_date = pd.DataFrame({'Date': [future_date]})

# Step 3: Predict the stock price for the future date
predicted_price = model.predict(future_date)

# Step 4: Print the predicted stock price formatted to two decimal places
print(f"Predicted stock price: ${predicted_price[0]:.2f}")
```

So, the script is executed to verify that the code functions as intended. Finally, execute the Python script.

Upon execution, the output displays the date and the predicted stock price. The plot generated shows that the actual and predicted stock prices are very close to each other, indicating that the model's predictions align well with the real data.

The **mean squared error** value obtained is 0.06, which is very close to zero. This low error value suggests that the model's predictions are highly accurate. Additionally, the **R2 score** is very close to one, further confirming the model's strong performance.

The last actual stock price was 160, and the predicted price for the next day is 161.21. This result suggests a slight increase in the stock price.

The output from the script is as follows:

```
Predicted stock price: $161.21
```

Based on this prediction, purchasing the stock could be considered, although no investment advice is being given. The demonstration concludes with encouragement for those who successfully completed the steps.

Summary

This chapter applied Python and machine learning skills across a range of practical projects, beginning with beginner-level exercises in calculator construction and rule-based chatbot design, then progressing to text classification with a spam email detector. The machine learning projects demonstrated supervised learning techniques including linear regression, logistic regression, decision trees, random forest, and XGBoost across domains such as housing, healthcare, finance, and human activity recognition. The chapter concluded with forecasting projects that used linear regression to predict weather conditions and stock prices from historical data.

In the next chapter, you will continue with hands-on projects, moving into deep learning, computer vision, natural language processing, recommendation systems, and AI game development.

6

Vision, Language, Recommendations and AI Projects

The previous chapter introduced hands-on projects spanning Python fundamentals, machine learning, and forecasting. This chapter continues that progression with projects in deep learning and computer vision, natural language processing, recommendation systems, and AI-driven game development. Each project builds on the libraries and workflows established earlier, introducing new techniques and frameworks along the way.

The following topics are covered in this chapter:

- Deep learning and computer vision projects
- Natural language processing (NLP) projects
- Recommendation system projects
- AI and game projects

Technical requirements

To follow along with the projects in this chapter, you will need the following:

- Python 3.8 or above
- Jupyter Notebook or any Python IDE (such as VS Code or PyCharm)
- The following Python libraries are installed via `pip`:
    - `tensorflow` and `keras`
    - `numpy`

- `nltk`
- `spacy`
- `torch` (PyTorch)
- `transformers` (Hugging Face)
- `scikit-learn`
- `pandas`
- `matplotlib` and `seaborn`
- `surprise`
- `speechrecognition`
- `pyttsx3`

You can install these libraries using:

```
pip install tensorflow keras numpy nltk spacy torch transformers scikit-learn
pandas matplotlib seaborn scikit-surprise speechrecognition pyttsx3
```

Additionally, download the spaCy English language model after installation using the following command:

```
python -m spacy download en_core_web_sm
```

Datasets used in this chapter can be downloaded from Kaggle (`https://www.kaggle.com`) and MovieLens (`https://grouplens.org/datasets/movielens/`). A free Kaggle account is required for Kaggle datasets.

Deep learning and computer vision projects

This section covers three projects that demonstrate the power of deep learning for visual and structural tasks. The first project builds a convolutional neural network using Keras and TensorFlow to classify handwritten digits from the MNIST dataset. The second extends this to binary image classification, training a CNN to distinguish between images of dogs and cats. The third takes a foundational approach, building a neural network entirely from scratch using only NumPy, providing insight into the mechanics of forward and backward propagation without relying on a deep learning framework.

Building an image classifier using Keras and TensorFlow

The next project involves constructing a simple image classifier to recognize handwritten digits using the **MNIST** dataset. The MNIST dataset consists of 28 × 28 grayscale images of handwritten digits from 0 to 9.

The implementation steps for the image classifier are as follows:

1. Install the required libraries. Keras and TensorFlow are installed using `pip`.
2. Load the MNIST dataset. Keras provides built-in datasets, including MNIST, which simplifies data loading.
3. Pre-process the data. Data normalization is performed to improve model performance.
4. Build the model. A convolutional neural network with convolution, pooling, flattening, and fully connected layers is used for classification.
5. Train the model. The model is trained on the training set and validated on the test set.
6. Evaluate and make predictions. After training, the model is evaluated, and predictions are made.

Create a Python file for this project.

Importing libraries for image classification

The first step is to import the required libraries. **NumPy** is imported for numerical operations, and **TensorFlow** is imported for building and training the neural network.

The following code imports the necessary libraries:

```
import numpy as np
import tensorflow as tf
```

NumPy is used for numerical operations in Python. TensorFlow is a deep learning library used to build and train the convolutional neural network (CNN).

The following code imports additional modules from TensorFlow's Keras API that are required for building and training the neural network:

```
from tensorflow.keras import datasets, layers, models
from tensorflow.keras.utils import to_categorical
```

The `datasets` module provides access to popular datasets such as MNIST. The `layers` module is used to add layers to the neural network, and `models` is used to define and compile the neural

network model. The `to_categorical` utility function converts class labels to one-hot encoded format, which is necessary for classification tasks.

Loading and preprocessing the MNIST dataset

The next step is to load the MNIST dataset. The MNIST dataset is loaded using the `datasets.mnist.load_data()` function. This function returns two tuples:

- One contains the training images and labels
- The second contains the test images and labels

The training images and labels are used for training the model, while the test images and labels are used for evaluating model accuracy.

The following code loads the MNIST dataset and assigns the data to the appropriate variables:

```
(train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data()
```

After loading the data, the pixel values of the images must be normalized. Normalization is performed by dividing each pixel value by 255, which scales the values to be between 0 and 1. This scaling makes the training process faster and more stable.

The following code normalizes the pixel values for both the training and test images:

```
train_images = train_images / 255.0
test_images = test_images / 255.0
```

The next step is to reshape the images. The convolutional neural network expects each image to have a 3D shape: height, width, and number of channels. Since the MNIST images are grayscale, each image has only one channel. Therefore, each image is reshaped to the dimensions 28 (height), 28 (width), and 1 (channel).

The following code reshapes the training and test images to the required dimensions:

```
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1)
test_images = test_images.reshape(test_images.shape[0], 28, 28, 1)
```

Finally, the labels are converted to one-hot encoded format. The `to_categorical` function is used for this purpose. One-hot encoding transforms a label such as 3 into a vector like [0, 0, 0,

1, 0, 0, 0, 0, 0, 0], which is required for multi-class classification with categorical cross-entropy loss. The following code converts the training and test labels to one-hot encoded format:

```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Constructing the CNN model

The next step is to build the convolutional neural network (CNN) model. The model is initialized using the `models.Sequential` function, which creates a sequential model. This type of model allows layers to be stacked linearly, one after another. The following code initializes a sequential model:

```
model = models.Sequential()
```

The first layer to add is a convolutional layer. This is accomplished using the `layers.Conv2D` function. The parameters passed are 32, 3, 3, `activation='relu'`, and `input_shape=(28, 28, 1)`. This adds a 2D convolutional layer with 32 filters, each of size 3 x 3 pixels. The activation function used is ReLU, which sets negative values to zero. The `input_shape` parameter specifies the input dimensions for the first layer, which are 28 x 28 pixels with 1 channel. The following code adds the first convolutional layer to the model:

```
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
```

After the convolutional layer, a max pooling layer is added using the `layers.MaxPooling2D` function with a pool size of (2, 2). This layer reduces the dimensions of the output by taking the maximum value in each 2 x 2 block. This operation decreases the computational load and helps detect important features.

The following code adds the max pooling layer:

```
model.add(layers.MaxPooling2D((2, 2)))
```

This sequence creates the first convolutional layer followed by a max pooling layer.

The next step is to add a second convolutional layer. This is done by calling the `layers.Conv2D` function again, this time with 64 filters of size 3 x 3 and the ReLU activation function. Another max pooling layer is added after this convolutional layer, again with a pool size of (2, 2). The second convolutional layer increases the number of filters to 64, and the subsequent max pooling layer reduces the output dimension by taking the maximum value in each 2 x 2 block.

The following code adds the second convolutional and max pooling layers:

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

The following step adds a third convolutional layer to the model. This layer uses 64 filters of size 3 x 3 and the ReLU activation function. No pooling layer is added after this convolutional layer, which preserves the spatial dimensions of the output for the subsequent flattening operation:

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

After the third convolutional layer, the output must be flattened from a 3D tensor to a 1D vector. This is necessary to connect the output to fully connected dense layers. The flattening is performed using the `layers.Flatten` function:

```
model.add(layers.Flatten())
```

The next step is to add a dense layer with 64 units and the ReLU activation function. This layer enables the model to learn complex representations from the features extracted by the convolutional layers:

```
model.add(layers.Dense(64, activation='relu'))
```

Finally, an output layer is added. This layer uses 10 units (neurons), corresponding to the 10 digit classes (0 to 9), and the softmax activation function. The softmax function converts the output to probabilities, allowing the model to classify each digit:

```
model.add(layers.Dense(10, activation='softmax'))
```

Compiling the model

The model is compiled by specifying the optimizer, loss function, and metrics to track during training. The Adam optimizer is used for adaptive learning. The loss function is categorical cross-entropy, which is suitable for multi-class classification with one-hot encoded labels. The accuracy metric is used to monitor the model's performance. The following code can be used to compile the model:

```
model.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

Training the model

The model is trained using the `fit()` function. The training data consists of images and their corresponding labels. The model is trained for 5 epochs, with a batch size of 64. Validation data is provided to evaluate the model's performance during training:

```
model.fit(train_images, train_labels, epochs=5, batch_size=64,
validation_data=(test_images, test_labels))
```

In this function:

- `train_images` and `train_labels` are the training inputs and outputs
- `epochs=5` specifies that the model will see the entire training dataset five times
- `batch_size=64` determines the number of samples processed before updating the model parameters
- `validation_data` is set to the test images and labels, which are used to evaluate model performance during training

Evaluating the model

The model's performance on the test data is evaluated using the `evaluate` function. This function returns the loss and accuracy on the test dataset. This will return a tuple:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

The following code prints the test accuracy as a percentage with two decimal places:

```
print(f"Test accuracy: {test_acc * 100:.2f}%")
```

A representative output is as follows:

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.9918 - loss: 0.0265
Test accuracy: 99.12%
```

Exact values may vary slightly between runs.

Making predictions on test images

Predictions on the test images are made using the `predict` function. This function returns class probabilities for each test image. The predicted digit for the first test image is obtained by finding the index with the highest probability:

```
predictions = model.predict(test_images)
print(f"Prediction for first test image: {np.argmax(predictions[0])}")
```

A representative output is as follows:

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step
Prediction for first test image: 7
```

Exact values may vary slightly between runs.

Visualizing digit predictions

To visualize the prediction, the `matplotlib` library is imported. The first test image is displayed using `imshow()`, reshaped to 28 x 28 pixels, and shown in grayscale. The plot title displays the predicted label, which is the digit with the highest probability in the prediction:

```
import matplotlib.pyplot as plt

plt.imshow(test_images[0].reshape(28, 28), cmap='gray')
plt.title(f"Predicted Label: {predictions[0].argmax()}")
plt.show()
```

Finally, run this Python script. Once the script runs successfully, it will begin training the model. This process may take some time, as the model iterates through all five epochs specified in the code and processes the large dataset of images.

After training, the script displays a plot showing the predicted label for the first test image. For example, if the predicted label is 7, the plot will display the image along with the label 'Predicted label: 7'. Upon visual inspection, the image corresponds to digit 7, confirming the prediction. The following image shows the first test digit with its predicted label displayed as the title:

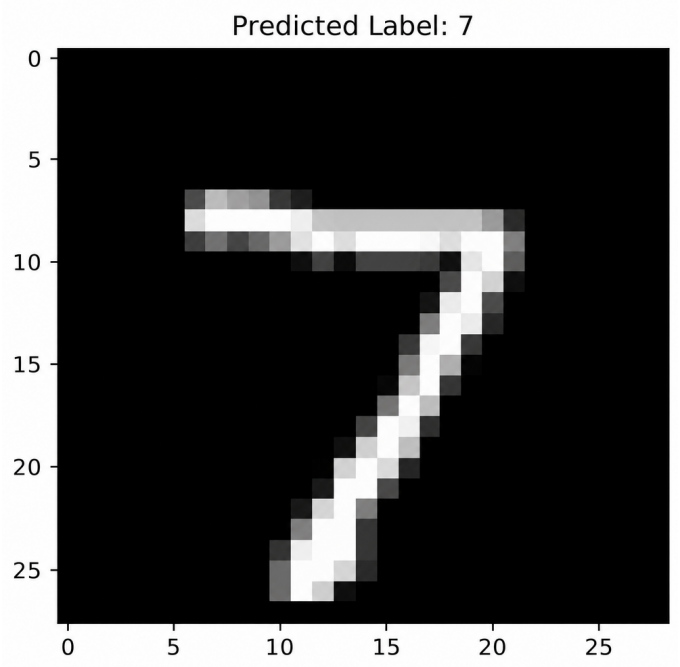


Figure 6.1: Handwritten digit image from MNIST test set predicted as 7 using the Keras image classifier

To view predictions for a different test image, the code can be modified to select another index. For instance, to display the fourth test image, change the 0 in the visualizing the output section to 4 to refer to the 4th element. The script is then rerun, and after completing the training process, it will display the predicted label for the fourth image. If the predicted label is 4, the plot will show the image with the label 'Predicted label: 4', and the image will visually represent digit 4 as shown:

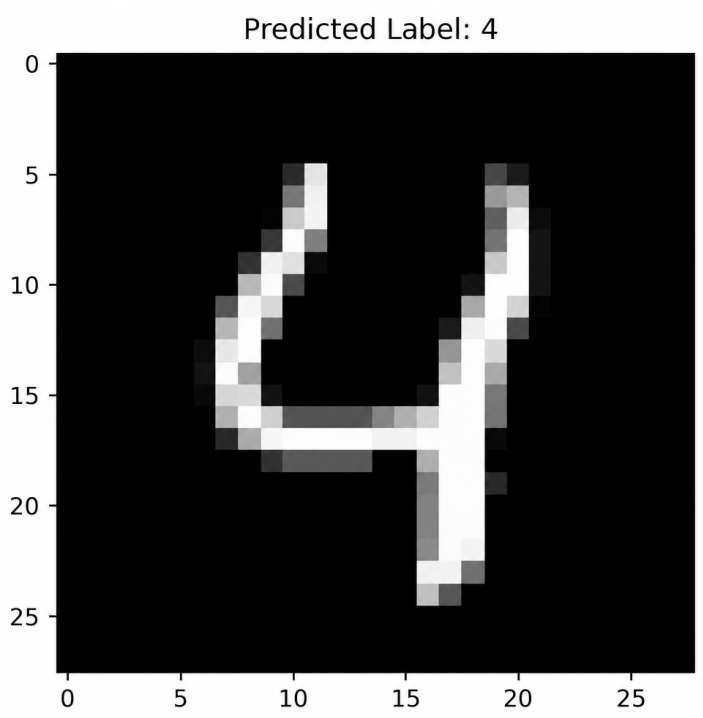


Figure 6.2: Handwritten digit image from MNIST test set predicted as 4 using the Keras image classifier

This process demonstrates the workflow for running an image classifier using Keras and TensorFlow, including troubleshooting common issues and visualizing model predictions.

Classifying dogs vs. cats with a convolutional neural network

A convolutional neural network (CNN) can be used to classify images of dogs and cats into their respective categories. In this project, a CNN is employed for both feature extraction and classification. The Kaggle dogs versus cats dataset is used, which contains labeled images of dogs and cats. The dataset is available here: `https://www.kaggle.com/datasets/tongpython/cat-and-dog`

The dataset can be downloaded from Kaggle, and it includes separate training and test sets, each containing images of both classes. Additionally, a single test image of a dog is available for evaluating the model after training.

The process involves three main stages: training the model, evaluating its performance, and testing it with a specific image to verify classification accuracy.

Importing libraries for CNN image classification

The first step is to import the necessary libraries. TensorFlow is used for building and training neural networks. The Keras API within TensorFlow provides modules for constructing models and layers. The `ImageDataGenerator` class from `tensorflow.keras.preprocessing.image` is used for image data augmentation, which helps the model generalize by creating variations in the training data. Matplotlib's `pyplot` module is used for plotting training and validation accuracy and loss curves. The following code imports the required libraries:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
```

Preparing the dataset paths

The dataset URL for the cats and dogs' images is provided, and the data must be downloaded from Kaggle. The dataset should be extracted, and the paths to the training and validation directories should be set according to the local file system. The training directory contains the training images, and the validation directory contains the test images. These paths must be updated to match the actual location of the dataset on the system.

The following code assigns the paths to the training and validation directories:

```
train_dir = 'training_set'
validation_dir = 'test_set'
```

Defining image data generators for augmentation and rescaling

Image data generators are defined to perform data augmentation and rescaling. The `ImageDataGenerator` class is initialized with several parameters:

- `rescale=1./255` rescales pixel values from the range 0 to 255 to the range 0 to 1
- `rotation_range=40` randomly rotates images by up to 40 degrees
- `width_shift_range=0.2` randomly shifts images horizontally by up to 20 percent of the width
- `height_shift_range=0.2` randomly shifts images vertically by up to 20 percent of the height
- `shear_range=0.2` randomly applies shear transformations
- `zoom_range=0.2` randomly zooms in on images

- `horizontal_flip=True` randomly flips images horizontally
- `fill_mode='nearest'` fills in new pixels that may be introduced after transformations using the nearest pixel values

These augmentation techniques help the model generalize better by artificially creating new training examples.

The following code defines the image data generator for the training data:

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

The `train_datagen` object prepares the training data by applying these augmentation techniques. This process increases the diversity of the training data and improves the model's ability to generalize.

Defining the validation data generator

For the validation data, only rescaling is required; data augmentation is not necessary. The following code defines the image data generator for the validation data, which rescales pixel values from 0 to 255 to the range 0 to 1:

```
validation_datagen = ImageDataGenerator(rescale=1./255)
```

Loading training and validation data

The next step is to load the training and validation data. This is accomplished by using the `flow_from_directory` method, which loads images from the specified directories. The following code creates the training data generator. It resizes all images to 150 x 150 pixels, uses a batch size of 32, and sets the class mode to binary, indicating a binary classification problem:

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(150, 150),
    batch_size=32,
```

```
    class_mode='binary'
)
```

Similarly, the validation data generator is created using the same method. Each subdirectory within the training or validation directory should correspond to a particular class, such as 'cats' and 'dogs'. The following code creates the validation data generator:

```
validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary'
)
```

The `flow_from_directory` method loads images from the specified directory, resizes them to the target size, loads them in batches of 32, and sets up the binary classification for cats and dogs.

Defining the CNN model architecture

The next step is to define the convolutional neural network (CNN) model. The model is created using the `Sequential` class. The first layer is a convolutional layer created with `Conv2D`, which extracts spatial features from the images. The first convolutional layer uses 32 filters with a kernel size of 3 x 3, ReLU activation, and an input shape of 150 x 150 pixels. The following code defines the initial layers of the model:

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
```

A second convolutional layer is added with 64 filters, followed by another max pooling layer:

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

A third convolutional layer is added with 128 filters, followed by another max pooling layer:

```
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

After the convolutional and pooling layers, the output is flattened and passed through fully connected (dense) layers. The first dense layer has 512 units with ReLU activation. The final dense layer has one unit with sigmoid activation to predict a binary outcome (cat or dog):

```
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Compiling the model

Once the model is defined, it is compiled using the Adam optimizer for efficient gradient-based optimization. The loss function is set to binary cross-entropy, which is suitable for binary classification problems. The metric used for evaluation is accuracy. The following code compiles the model:

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Printing the model summary

A summary of the model can be printed to display the architecture and details of each layer. The following code prints the model summary:

```
model.summary()
```

Training the CNN model

To train the model, the `fit` method is called with the training and validation generators. The model is trained for 20 epochs. Keras automatically determines the number of steps from the number of available batches, which avoids requesting more batches than the generators contain. Use the following code to train the model:

```
history = model.fit(
    train_generator,
    epochs=20,
    validation_data=validation_generator
)
```

Plotting training and validation accuracy and loss

After training, the training and validation accuracy and loss can be visualized. The accuracy and loss values are extracted from the `history` object. The following code retrieves these metrics:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
```

Next, a figure is created with a size of 12 x 6. The first subplot displays the training and validation accuracy over the epochs. The second subplot displays the training and validation loss. The following code generates these plots:

```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(epochs, acc, 'b', label='Training accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation Loss')
plt.legend()

plt.show()
```

The output is as follows:

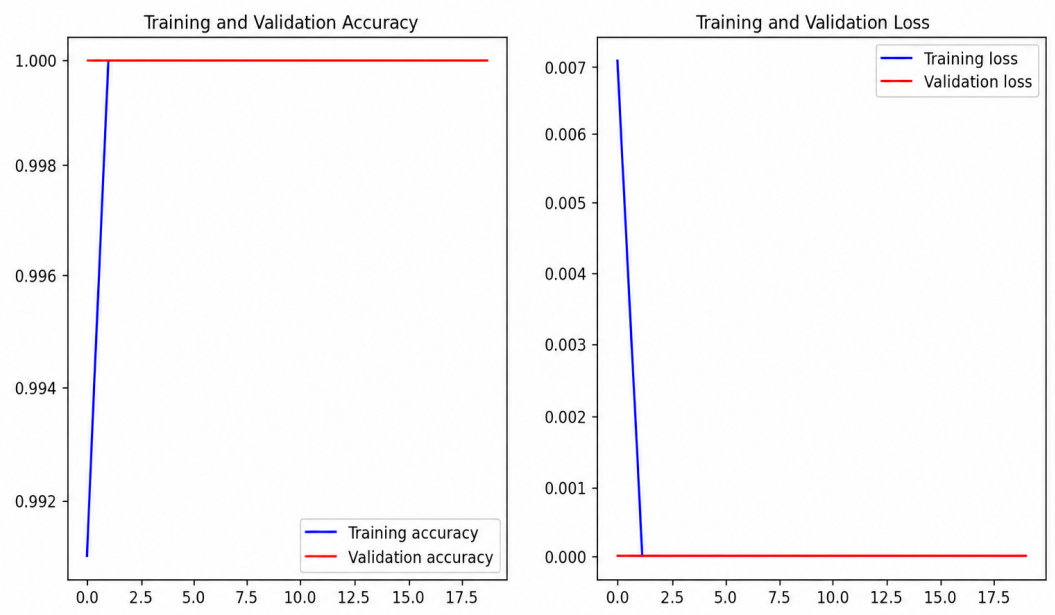


Figure 6.3: Binary classification result from the CNN model showing prediction for a dog or cat image

The plots will display the progression of accuracy and loss for both training and validation sets, providing insight into the model's learning behavior.

Making predictions on new images

To test the trained model with a new image, the `image` module from `tensorflow.keras.preprocessing` is imported, along with NumPy. A function is defined to load and preprocess a new image, then make a prediction using the trained model. The function takes the model and the path to the image as arguments. The following code defines the prediction function:

```
from tensorflow.keras.preprocessing import image
import numpy as np

def predict_image(model, image_path):
    img = image.load_img(image_path, target_size=(150, 150))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = img_array / 255.0
    prediction = model.predict(img_array)
```

```
    if prediction[0][0] > 0.5:
        print(f"The image is predicted to be a dog with confidence {prediction[0]
[0]:.2f}")
    else:
        print(f"The image is predicted to be a cat with confidence {1 -
prediction[0][0]:.2f}")
```

This function loads the image, resizes it to 150 x 150 pixels, converts it to an array, adds a batch dimension, and normalizes the pixel values to the range 0 to 1. The model's `predict` method is then called on the processed image. If the prediction is greater than 0.5, the image is classified as a dog; otherwise, it is classified as a cat. The confidence score is displayed accordingly.

To test the function, the following code is used, where the test image path and the trained model are provided:

```
predict_image(model, 'test_image.jpg')
```

A representative output is as follows:

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 148ms/step
The image is predicted to be a cat with confidence 1.00
```

Interpreting the classification results

After running the training for 20 epochs, the plots for training and validation accuracy and loss are displayed. The order and appearance of the graphs indicate the model's performance. If the model is trained for fewer epochs, the difference in performance may be more noticeable. Increasing the number of epochs generally improves the output.

When testing with a new image, the model may predict the image as a cat with a confidence of 0.52, even if the image provided is of a dog. This result suggests that the training may not have been sufficient or that the image was not processed optimally. The project demonstrates the process of building a model that can predict whether an image is of a dog or a cat using a convolutional neural network.

Building a basic neural network from scratch with NumPy

This project demonstrates how a basic neural network operates internally by constructing one from scratch in Python using only NumPy. This example implements a simple feedforward neural network with a single hidden layer, employing the sigmoid activation function and mean squared error as the loss function.

Defining activation and loss functions

The first step is to import the necessary library, NumPy, which is essential for matrix operations and mathematical functions. The following code imports NumPy and assigns it the alias `np`:

```
import numpy as np
```

The next step is to define the **sigmoid activation function** and its derivative. The sigmoid function is implemented as a function that takes a parameter `x` and returns the value of 1 divided by 1 plus the exponential of negative `x`. The derivative of the sigmoid function is also defined, returning `x` times `1 - x`. These two functions are written as follows:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

The **mean squared error (MSE) loss function** is then defined. This function, named `mean_squared_error`, takes two parameters: `y_true` and `y_pred`. It returns the mean of the squared differences between the true and predicted outputs, which measures the performance of the model. The implementation is as follows:

```
def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)
```

Initializing the neural network class

After defining the activation and loss functions, the next step is to implement the basic neural network class. This class is named `BasicNeuralNetwork`. The initializer method (`__init__`) for this class takes four parameters: the `self` object, `input_size`, `hidden_size`, and `output_size`. The initializer sets up the neural network with random weights and biases as follows:

1. The weights between the input layer and the hidden layer are initialized using `np.random.randn(input_size, hidden_size)`
2. The weights between the hidden layer and the output layer are initialized using `np.random.randn(hidden_size, output_size)`
3. The bias for the hidden layer is initialized using `np.random.randn(1, hidden_size)`
4. The bias for the output layer is initialized using `np.random.randn(1, output_size)`

The following code defines the class and its initializer:

```
class BasicNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_hidden = np.random.randn(1, hidden_size)
        self.bias_output = np.random.randn(1, output_size)
```

Implementing the forward pass

The next method to implement within this class is the **forward pass** inside the same class. The forward pass method, named `forward`, takes two parameters: the `self` object and `x`, which represents the features. The method performs the following steps:

1. Calculate the input to the hidden layer by taking the dot product of `x` and `self.weights_input_hidden`, then adding `self.bias_hidden`
2. Compute the output from the hidden layer by applying the sigmoid function to the `hidden_input`
3. Calculate the input to the output layer by taking the dot product of the hidden output and `self.weights_hidden_output`, then adding `self.bias_output`
4. Compute the final output by applying the sigmoid function to the output input
5. Return the final output

The code for the forward pass is as follows:

```
def forward(self, X):
    self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
    self.hidden_output = sigmoid(self.hidden_input)
    self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
    self.output = sigmoid(self.output_input)
    return self.output
```

Careful attention is required when implementing the forward pass, as there are multiple variables involving input and output at different layers. Ensuring the correct variables are used at each step is crucial for the network to function as intended.

Implementing the backward pass and weight updates

The following step in constructing the neural network is to implement the **backward pass** and the **weight updates**. This process is encapsulated in a method named `backward`, which is responsible for propagating the error backward through the network and updating the weights accordingly.

The backward method requires the input features `X`, the target values `y`, the predicted output, and the learning rate. These inputs are used to calculate the gradients and update the network parameters. The implementation steps are as follows:

1. Compute the output error by subtracting the predicted output from the actual target values.
2. Calculate the output delta, which represents the adjustment for the output layer. This is obtained by multiplying the output error by the derivative of the sigmoid function applied to the output.
3. Determine the hidden error, which is the error propagated back to the hidden layer. This is calculated using the dot product of the output delta and the transpose of the weights connecting the hidden layer to the output layer.
4. Compute the hidden delta, which represents the adjustment for the hidden layer

The following code defines the initial part of the backward pass and weights update method:

```
def backward(self, X, y, output, learning_rate):
    output_error = y - output
    output_delta = output_error * sigmoid_derivative(output)
    hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
    hidden_delta = hidden_error * sigmoid_derivative(self.hidden_output)
```

Careful calculation of these errors and deltas is essential for the network to learn effectively during training. Each step ensures that the gradients are correctly computed for both the output and hidden layers.

The next step involves updating the weights and biases using the gradients calculated during the backward pass. These updates are performed using gradient descent, which adjusts the parameters in the direction that minimizes the loss.

To update the weights connecting the hidden layer to the output layer, the following operation is performed:

```
self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) *
learning_rate
```

This line updates the weights by adding the product of the transposed hidden layer outputs and the output delta, scaled by the learning rate. The bias for the output layer is updated by summing the output delta across the appropriate axis, keeping the dimensions consistent, and scaling by the learning rate:

```
self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
```

Next, the weights connecting the input layer to the hidden layer are updated. This is done by taking the dot product of the transposed input features and the hidden delta, then scaling by the learning rate:

```
self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
```

Finally, the bias for the hidden layer is updated by summing the hidden delta across the correct axis, keeping dimensions, and multiplying by the learning rate:

```
self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

These updates complete the backward pass and ensure that the network's parameters are adjusted based on the computed gradients.

Implementing the training loop

The next task is to implement the training process for the neural network. This is achieved by defining a `train` method, which takes parameters like self, input features `x`, target values `y`, the number of epochs to train for, and the learning rate. The first step in this method is to run a loop over the specified number of epochs.

The following code implements the training loop for the neural network. This loop iterates over the specified number of epochs, performing the forward and backward passes, and printing the loss every hundred epochs to monitor progress:

```
def train(self, x, y, epochs, learning_rate):
    for epoch in range(epochs):
        # Forward pass
        output = self.forward(x)

        # Backward pass
        self.backward(x, y, output, learning_rate)

        # Print loss every 100 epochs
        if epoch % 100 == 0:
```

```
            loss = np.mean(np.square(y - output))
            print(f"Epoch: {epoch}, Loss: {loss}")
```

This method first performs the forward pass to compute the output, then executes the backward pass to update the weights and biases. Every hundred epochs, it calculates the mean squared error loss and prints the current epoch and loss value.

Training with the XOR dataset

The next step is to train the neural network using the XOR dataset. The XOR problem is commonly used to test neural networks because it is not linearly separable. The following code creates the input data for the XOR problem as a NumPy array:

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

The corresponding target values for the XOR operation are created as follows:

```
Y = np.array([[0], [1], [1], [0]])
```

These arrays represent the input features and expected outputs for the XOR function.

To initialize the neural network, an instance is created with two input neurons, two hidden neurons, and one output neuron. The following code demonstrates this initialization:

```
nn = BasicNeuralNetwork(input_size=2, hidden_size=2, output_size=1)
```

This sets up the neural network architecture required for the XOR task. The following code trains the neural network on the XOR dataset for 10,000 epochs with a learning rate of 0.1:

```
nn.train(X, Y, epochs=10000, learning_rate=0.1)
```

After training, the neural network can be tested to evaluate its performance on the XOR data set. The following code prints a message indicating that the trained neural network is being tested, then iterates over each input in the data set, printing the input, the predicted output from the network, and the actual target output:

```
print("\nTesting the trained neural network.")
for i in range(len(X)):
    print(f"Input: {X[i]}, Predicted Output: {nn.forward(X[i])}, Actual Output:
{Y[i]}")
```

When this code is executed, the training process prints the loss every 100 epochs, showing the progress of the network as it learns. After training, the printed outputs for each input in the XOR data set are as follows:

```
Epoch 0, Loss = 0.277...
Epoch 100, Loss = 0.250...
...
Epoch 9900, Loss = 0.015...
Testing the trained neural network.
Input: [0 0], Predicted Output: [0.04], Actual Output: [0]
Input: [0 1], Predicted Output: [0.94], Actual Output: [1]
Input: [1 0], Predicted Output: [0.9], Actual Output: [1]
Input: [1 1], Predicted Output: [0.0], Actual Output: [0]
```

The predicted outputs are very close to the expected values for the XOR function, demonstrating that the neural network has learned to approximate the XOR operation with high accuracy.

This process involves initializing the neural network with random weights and biases, performing the forward pass to compute outputs, executing the backward pass to update weights and biases using gradient descent, and training over multiple epochs to minimize the loss. After training, the network's performance is evaluated by comparing its predicted outputs to the actual target values.

Natural language processing (NLP) projects

This section applies NLP techniques to a range of text analysis tasks. Projects cover sentiment analysis using NLTK's Naive Bayes classifier on movie reviews, frequency-based text summarization, fake product review detection using TF-IDF and logistic regression, emotion detection using VADER, named entity recognition with spaCy, fake news detection using Multinomial Naive Bayes, resume scanning via keyword extraction and cosine similarity, and automatic essay grading using BERT. Together, these projects demonstrate the breadth of NLP applications and the variety of approaches available, from rule-based methods to transformer-based models.

Sentiment analysis on text data using NLTK

This project implements sentiment analysis on text data using **NLTK**, which stands for Natural Language Toolkit. In this project, the sentiment of text data is classified as positive or negative based on the text content. The NLTK library is used to preprocess the data, and a **Naive Bayes classifier** is used to perform sentiment analysis.

Let's begin by creating a new Python file.

Importing libraries for sentiment analysis

The **NLTK corpus** includes a collection called `movie_reviews`, which contains labeled movie reviews categorized as positive or negative. This dataset is used for both training and testing sentiment analysis models.

The **Naive Bayes classifier** is a probabilistic classifier that applies Bayes' theorem and is frequently used for text classification tasks such as sentiment analysis. The following import statements are used to access the movie reviews corpus and the Naive Bayes classifier from NLTK:

```
import nltk
from nltk.corpus import movie_reviews
from nltk.classify import NaiveBayesClassifier
```

Next, the accuracy function from NLTK is required. Accuracy is used to evaluate the classifier, as done in previous chapters, but in this case, it is imported from `nltk.classify.util`. To avoid confusion with the accuracy function from scikit-learn, it is imported with the alias `nltk_accuracy`:

```
from nltk.classify.util import accuracy as nltk_accuracy
```

Additionally, stop words are imported from `nltk.corpus`:

```
from nltk.corpus import stopwords
```

These stop words include common words such as "the" and "and" that are typically removed during text processing:

The `random` module is also imported to shuffle the dataset and ensure randomness:

```
import random
```

The next step is to download the required NLTK data files. This includes the `movie_reviews` corpus, which contains the labeled movie reviews, and `punkt`, which is a tokenizer used for splitting sentences into words. The stop words data is also downloaded. These downloads only

need to be performed once; after the data is downloaded, the download commands can be commented out. The following code downloads the required NLTK data files:

```
nltk.download('movie_reviews')
nltk.download('punkt')
nltk.download('stopwords')
```

Feature extraction for sentiment analysis

The dataset must be preprocessed and features extracted. A function named `extract_features` is created for this purpose. This function converts a list of words into a dictionary where each word is a key and the value is `True`. This format is required by NLTK classifiers to process features. The following code defines the feature extraction function:

```
def extract_features(words):
    return {word: True for word in words}
```

For example, if the input is a list of words such as `["good", "movie"]`, the function returns a dictionary:

```
{'good': True, 'movie': True}
```

This format allows the classifier to recognize the presence of each word in the review.

Loading and preprocessing the movie reviews dataset

The next step is to load and preprocess the dataset. The `movie_reviews` corpus from NLTK is used for this purpose. The dataset is loaded as a list of tuples, where each tuple contains the list of words from a movie review and its corresponding category label. The following code loads the dataset:

```
documents = [(list(movie_reviews.words(fileid)), category)
             for category in movie_reviews.categories()
             for fileid in movie_reviews.fileids(category)]
```

In this code, `movie_reviews.words(file_id)` loads the words from each movie review. `movie_reviews.categories()` returns the two categories: `'pos'` for positive and `'neg'` for negative. The `fileid` represents a specific movie review file. This process creates a list of tuples, where each tuple contains the words in a review and its label (either `'pos'` or `'neg'`).

Shuffling the dataset

To ensure a random distribution of the data, the dataset is shuffled using `random.shuffle` function. This step helps to avoid any ordering bias during training and testing. The following code shuffles the dataset:

```
random.shuffle(documents)
```

Preparing the training and testing sets

The dataset is then prepared for training and testing. First, the feature set is created by converting each review into a set of features using the `extract_features` function. The feature set is a list of tuples, where each tuple contains the feature dictionary and the corresponding category label. The following code creates the feature set:

```
feature_sets = [(extract_features(d), c) for (d, c) in documents]
```

Next, the feature set is split into a training set and a testing set. The first 1,600 reviews (80% of the data) are used for training, and the remaining 400 reviews (20% of the data) are used for testing. This split is based on the total of 2,000 reviews in the dataset. The following code splits the feature set:

```
train_set, test_set = feature_sets[:1600], feature_sets[1600:]
```

This division ensures that the classifier is trained on the majority of the data and evaluated on a separate subset to measure its performance.

Training the Naive Bayes classifier

The next step is to train the **Naive Bayes classifier** using the prepared training set. The classifier is trained to distinguish between positive and negative sentiments by learning from the words present in the movie reviews.

The following code trains the Naive Bayes classifier with the training set:

```
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

This line creates a classifier object and trains it using the `train_set` that was created earlier.

Evaluating the classifier

After training, the classifier's performance is evaluated on the test set. The evaluation is performed by calculating the **accuracy** of the classifier, which is the percentage of correct

predictions it makes on the test data. The following code evaluates the classifier and prints the accuracy as a percentage:

```
accuracy = nltk.classify.accuracy(classifier, test_set)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

A representative output is as follows:

```
Accuracy: 72.25%
```

The `nltk.classify.accuracy` function compares the predicted labels from the classifier with the actual labels in the test set and computes the percentage of correct predictions. The result is printed as a percentage for easier interpretation.

Displaying the most informative features

To understand which words are most useful for the classifier in distinguishing between positive and negative reviews, the most informative features can be displayed. These features represent the words that provide the most information for the classifier's decision-making process. The following code displays the top 10 most informative features:

```
classifier.show_most_informative_features(10)
```

This command outputs the ten words that the classifier finds most useful for making predictions about sentiment. These features help to interpret the classifier's behavior and provide insight into which words are most strongly associated with positive or negative sentiment in the dataset.

Defining the sentiment analysis function

To test the trained classifier on new input sentences, a function is created to analyze the sentiment of any given text. This function, named `analyze_sentiment`, processes the input text and predicts whether the sentiment is positive or negative.

The following code defines the `analyze_sentiment` function, which performs several steps:

1. Tokenize the input text into individual words using `nltk.word_tokenize`
2. Remove common stop words, such as "the" and "and", by filtering out words found in the English stop words list
3. Extract features from the filtered words using the previously defined feature extraction function
4. Use the trained classifier to predict the sentiment based on the extracted features

The code for this function is as shown:

```
def analyze_sentiment(text):
    words = nltk.word_tokenize(text)
    words = [word.lower() for word in words if word.lower() not in
stopwords.words('english')]

    features = extract_features(words)
    return classifier.classify(features)
```

This function takes a text string as input, removes stop words, extracts the relevant features, and returns the corpus labels "pos" or "neg."

Testing the classifier with custom movie review sentences

To evaluate the classifier's performance on new examples, several custom test sentences are prepared. These sentences are chosen to represent a range of sentiments, from clearly positive to clearly negative, as well as some that are more neutral or mixed in tone.

To test the classifier, a loop is used to iterate through each test sentence. For each sentence, the code prints the sentence itself, then prints the predicted sentiment by calling the `analyze_sentiment` function, and finally prints an empty line for readability.

The following code tests the classifier on the custom sentences and displays the results:

```
test_sentences = [
    "This movie is absolutely fantastic, the acting, the story, everything was
amazing.",
    "I hated this movie just from the start.",
    "The plot was a bit dull, but the performance was great, which means I have a
very neutral type, but end of the day, I like the movie.",
    "I have mixed feelings about the film. It was OK, not great."
]

for sentence in test_sentences:
    print(f"Sentence: {sentence}")
    print(f"Predicted sentiment is: {analyze_sentiment(sentence)}")
    print()
```

When this code is run, the classifier processes each sentence and outputs the predicted sentiment label. The output for these test sentences is as follows:

```
Sentence: This movie is absolutely fantastic, the acting, the story, everything
was amazing.
Predicted sentiment is: pos

Sentence: I hated this movie just from the start.
Predicted sentiment is: neg

Sentence: The plot was a bit dull, but the performance was great, which means I
have a very neutral type, but end of the day, I like the movie.
Predicted sentiment is: neg

Sentence: I have mixed feelings about the film. It was OK, not great.
Predicted sentiment is: neg
```

The classifier assigns a "pos" label to the first sentence, which is strongly positive in tone. The remaining sentences are classified as "neg," even those that are more neutral or mixed. This demonstrates that the classifier, as trained, does not provide a neutral category and tends to assign ambiguous or mixed reviews to the negative class.

This approach demonstrates how sentiment analysis can be performed on new text data using the NLTK framework. The classifier is able to process and classify new sentences based on the features it learned during training.

Frequency-based text summarization with NLTK

This project demonstrates how to build a **text summarizer** using the NLTK in Python. This approach implements frequency-based extractive summarization, where the frequency of each word in the input text is calculated, and each sentence is scored based on the sum of the frequencies of the words it contains. Sentences with higher scores are considered more important and are selected for inclusion in the summary.

Importing NLTK and downloading resources

The following code imports the necessary NLTK modules and downloads the required resources:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
```

```
nltk.download('punkt')
nltk.download('stopwords')
```

This code performs the following actions:

1. Imports the `nltk` library and specific modules for stop words and tokenization
2. Downloads the `'punkt'` resource, which is used for sentence and word tokenization
3. Downloads the `'stopwords'` resource, which provides a list of common stop words in English

Defining the input text

The next step is to define the input text that will be summarized. In this example, a sample passage about artificial intelligence is assigned to a variable named `text`. This variable serves as the input for the summarization process. The content of `text` can be replaced with any long document or article to test the summarizer with different inputs. The following code assigns the example text to the `text` variable:

```
text = """
Artificial intelligence (AI) refers to the simulation of human intelligence in
machines that are programmed to think like humans and mimic their actions. The
term may also be applied to any machine that exhibits traits associated with the
human mind, such as learning and problem-solving.

The ideal characteristic of artificial intelligence is its ability to rationalize
and take actions that have the best chance of achieving a specific goal. A subset
of artificial intelligence is machine learning, which refers to the concept that
computer programs can automatically learn from and adapt to new data without being
assisted by humans.
Deep learning techniques enable this automatic learning through the absorption of
huge amounts of unstructured data such as text, images, or video.
"""
```

Creating the frequency-based summarization function

A function is then defined to generate a summary based on word frequency. This function is named `summarize_text` and takes two parameters: the input `text` and the desired

`num_sentences` in the summary, which defaults to 2. The following code defines the summarization function:

```
def summarize_text(text, num_sentences=2):
    # Tokenize the text into sentences
    sentences = sent_tokenize(text)

    # Tokenize the text into words and convert to lowercase
    words = word_tokenize(text.lower())

    # Set of English stopwords
    stop_words = set(stopwords.words("english"))

    # Calculate word frequencies, ignoring stopwords and non-alphabetic words
    word_frequencies = {}
    for word in words:
        if word.isalpha() and word not in stop_words:
            word_frequencies[word] = word_frequencies.get(word, 0) + 1
            # Score each sentence based on word frequencies
            sentence_scores = {}
            for sentence in sentences:
                for word in word_tokenize(sentence.lower()):
                    if word in word_frequencies:
                        sentence_scores[sentence] = sentence_scores.get(sentence,
0) + word_frequencies[word]
                        # Sort sentences by score and select the top num_sentences
                        summary_sentences = sorted(sentence_scores,
key=sentence_scores.get, reverse=True)[:num_sentences]
                        # Join the selected sentences to form the summary
                        summary = " ".join(summary_sentences)
                        return summary
```

This function performs the following steps:

1. Tokenizes the input text into individual sentences using `sent_tokenize`
2. Tokenizes the text into words, converting all words to lowercase for consistent counting
3. Creates a set of English stop words using `stopwords.words("english")`
4. Initializes an empty dictionary for word frequencies

5. Iterates through each word, counting only those that are alphabetic and not in the stop words set
6. Initializes an empty dictionary for sentence scores
7. Iterates through each sentence and each word in the sentence, incrementing the sentence's score by the frequency of each word it contains
8. Sorts the sentences by their scores in descending order and selects the top `num_sentences`
9. Joins the selected sentences into a single summary string and returns it

To execute the summarization function and observe its output, the following code is used to generate and print a summary of the provided text. The function is called with the original text and the number of sentences set to two. Both the original text and the resulting summary are printed for comparison:

```
summary = summarize_text(text, num_sentences=2)
print("Original Text:\n", text)
print("\nSummary:\n", summary)
```

When this script is run, the output first displays the entire input text, followed by the summary generated by the frequency-based approach. For example, given an input about artificial intelligence and machine learning, the output appears as follows:

```
Original Text:
Artificial intelligence (AI) refers to the simulation of human intelligence in
machines that are programmed to think like humans and mimic their actions.
The term may also be applied to any machine that exhibits traits associated
with a human mind such as learning and problem-solving.

The ideal characteristic of artificial intelligence is its ability to rationalize
and take actions that have the best chance of achieving a specific goal.
A subset of artificial intelligence is machine learning, which refers to the
concept that computer programs can automatically learn from and adapt to new data
without being assisted by humans.
Deep learning techniques enable this automatic learning through the absorption of
huge amounts of unstructured data such as text, images, or video.

Summary:
A subset of artificial intelligence is machine learning, which refers to the
concept that computer programs can automatically learn from and adapt to new data
without being assisted by humans.
```

```
Artificial intelligence (AI) refers to the simulation of human intelligence in
machines that are programmed to think like humans and mimic their actions.
```

This demonstrates that the summarization function selects the two most significant sentences from the input, based on word frequency analysis. The result is a concise summary that retains the main ideas from the original text.

Detecting fake product reviews using NLP techniques

This project focuses on building a fake product review detection model using **natural language processing (NLP)** techniques. The objective of this project is to determine whether a product review is genuine or fake based solely on its textual content. The approach involves pre-processing the review texts, converting them into numerical features using **TF-IDF vectorization**, and then training a classifier to predict the authenticity of each review.

This project uses the Kaggle *Fake Reviews* dataset, which contains more than 40,000 balanced reviews. The original file includes category, rating, label, and text fields. The label values CG and OR represent computer-generated and original reviews, respectively. For this project, text is renamed to `review_text`, and the labels are mapped to fake and genuine.

Download the CSV available at `https://www.kaggle.com/datasets/mexwell/fake-reviews-dataset`, place it in the project folder, and rename it `reviews.csv`.

Importing libraries for fake review detection

The following libraries are imported to begin the project:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
```

We are using the **TF-IDF vectorizer** to convert the review texts into numerical features. The TF-IDF (term frequency-inverse document frequency) scores are calculated using the `TfidfVectorizer` from `sklearn.feature_extraction`.

Loading and inspecting the reviews dataset

Download the dataset, rename the CSV file to `reviews.csv`, and place it in the same folder as the Python script. The following code loads the data, renames the text column, and converts the original CG/OR labels into human-readable fake/genuine labels:

```
df = pd.read_csv('reviews.csv')
df = df.rename(columns={'text_': 'review_text'})
df['label'] = df['label'].replace({'CG': 'fake', 'OR': 'genuine'})
```

To display the first few rows of the data set, the `.head()` method is called on the DataFrame. This provides a quick look at the structure and contents of the data. The following code prints the first five rows of the data set:

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

A representative preview of the prepared dataset is as follows:

```
             category  rating label  \
0  Home_and_Kitchen_5     5.0  fake
1  Home_and_Kitchen_5     5.0  fake
2  Home_and_Kitchen_5     5.0  fake
3  Home_and_Kitchen_5     1.0  fake
4  Home_and_Kitchen_5     5.0  fake

                                       review_text
0  Love this!  Well made, sturdy, and very comfor...
1  love it, a great upgrade from the original.  I...
2  This pillow saved my back. I love the look and...
3  Missing information on how to use it, but it i...
4  Very nice set. Good quality. We have had the s...
```

Rows may differ by dataset version.

Extracting features and splitting the dataset

The next step is to extract the features and the target variable from the DataFrame. The feature is the review text, which is stored in the `review_text` column of the DataFrame. The target is

the `label` column, which indicates whether a review is fake or genuine. The following code assigns the review texts to `X` and the labels to `y`:

```
X = df['review_text']
y = df['label']
```

After defining the features and target, the dataset is split into training and testing sets. This is accomplished using the `train_test_split` function. The function is called with the following parameters: `X`, `y`, `test_size=0.2`, `random_state=42`, and `stratify=y`. Setting `stratify` to `y` ensures that both classes, fake and genuine, are proportionally represented in both the training and testing splits. The `test_size` parameter is set to 0.2, so 20% of the data is reserved for testing and 80% for training. The following code performs the train-test-split:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

Vectorizing the review text with TF-IDF

The next step is to initialize the `TfidfVectorizer`. The `TfidfVectorizer` is configured with English stop words and a maximum document frequency (`max_df`) of 0.7. The stop words parameter removes common English words such as "and," "if," "that," and "the," which are frequently used and do not contribute meaningful information to the classification task. Setting `max_df` to 0.7 ensures that any word appearing in more than 70 percent of the documents is ignored. For example, in a collection of product reviews, words like "product" or "food" might appear very frequently and are therefore excluded from the feature set. The following code initializes the `TfidfVectorizer` with these parameters:

```
vectorizer = TfidfVectorizer(stop_words='english', max_df=0.7)
```

After initializing the vectorizer, the next step is to fit the vectorizer to the training data and transform the training texts into Tfidf vectors. This process learns the vocabulary from the training data and converts each review into a vector representation based on term frequency and inverse document frequency. The following code fits the vectorizer to the training data and transforms it:

```
X_train_tfidf = vectorizer.fit_transform(X_train)
```

The test data is then transformed using the same vocabulary learned from the training data. This ensures that the test data is represented in the same feature space as the training data. The following code transforms the test data:

```
X_test_tfidf = vectorizer.transform(X_test)
```

Training and evaluating the fake review classifier

The next step is to initialize the **logistic regression classifier**. Logistic regression is suitable for binary classification tasks, such as distinguishing between fake and genuine reviews. The following code initializes the logistic regression classifier:

```
model = LogisticRegression(max_iter=1000, random_state=42)
```

Once the classifier is initialized, it is trained on the training data. The model learns the relationship between the Tfidf vectors and the corresponding labels. The following code trains the logistic regression model:

```
model.fit(X_train_tfidf, y_train)
```

After training, predictions are made on the test set. The trained model predicts the labels for the test data, identifying whether each review is fake or genuine. The following code makes predictions on the test set:

```
y_pred = model.predict(X_test_tfidf)
```

To evaluate the performance of the model, the accuracy score is calculated. The accuracy score measures the proportion of correct predictions made by the model on the test data. The following code calculates the accuracy score:

```
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

The next step is to generate a classification report. This report provides detailed metrics such as precision, recall, and F1 score for each class, which in this case are fake and genuine reviews. Printing both the accuracy and the classification report allows for a comprehensive assessment of the model's performance. The following code prints the accuracy and the classification report:

```
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)
```

```
print(f"\nAccuracy: {accuracy:.3f}")
print(f"\nClassification Report:\n{report}")
```

The output will be:

```
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

fake              1.00      1.00      1.00         3
genuine           1.00      1.00      1.00         2

accuracy                            1.00         5
macro avg         1.00      1.00      1.00         5
weighted avg      1.00      1.00      1.00         5
```

The output displays accuracy, which in this case is 1.0, indicating perfect classification on the current data set. The classification report shows high percentages for all metrics, reflecting the ideal nature of the data set used in this example. The report details how well the model performs in terms of precision, recall, F1 score, and support for each class. Results like these are uncommon with real-world datasets, but this example demonstrates the process effectively.

Detecting emotions in text using the Natural Language Toolkit

This project demonstrates how to detect emotion in text using the NLTK. This project analyzes the sentiment of a given text to determine if it expresses emotions such as joy, sadness, or anger. The example focuses on basic sentiment analysis to classify text as positive, negative, or neutral.

Importing libraries for emotion detection

The following code imports the necessary libraries for the project. NLTK is imported as the main toolkit for natural language processing:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
```

The `FreqDist` function is also imported, which is useful for analyzing word frequencies, although it is not essential for basic sentiment analysis.

Downloading required NLTK data

The following code downloads the required NLTK datasets. These downloads are only necessary the first time the code is run:

```
nltk.download('vader_lexicon')
nltk.download('stopwords')
nltk.download('punkt')
```

The `vader_lexicon` is downloaded, which is the lexicon used by NLTK's VADER sentiment analysis tool and is specifically tuned for analyzing sentiment in social media text. The `stopwords` and `punkt` datasets are also downloaded. If the code is run multiple times, these lines can be commented out to avoid repeated downloads.

Importing and initializing the sentiment analyzer

The following code imports the `SentimentIntensityAnalyzer` from NLTK's `sentiment.vader` module. The `SentimentIntensityAnalyzer` is NLTK's VADER sentiment analyzer, which provides a pre-trained model for sentiment analysis tuned for social media and informal text. The analyzer is then instantiated as follows:

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
sid = SentimentIntensityAnalyzer()
```

Creating sample text for emotion detection

A sample text is required for emotion detection. The following example text is used, which expresses a very positive mood:

```
text = """
I am so happy today. The weather is beautiful, and everything is going well. I
feel very positive and motivated.
"""
```

This example text is primarily positive. Any other text can be substituted to detect emotions in different contexts.

Writing a function for emotion detection

To detect emotions in the sample text, a function is defined that analyzes the sentiment of the input text and assigns an emotion label based on the sentiment scores. The function is named `detect_emotion` and takes a single parameter, `text`.

The following code defines the `detect_emotion` function, which uses the previously initialized sentiment analyzer to obtain sentiment scores for the input text:

```
def detect_emotion(text):
    # Analyze the sentiment of the text
    scores = sid.polarity_scores(text)
    print(scores)  # Print the sentiment scores for inspection

    # Determine emotion based on the compound score
    if scores['compound'] >= 0.5:
        emotion = 'Joy'
    elif scores['compound'] <= -0.5:
        emotion = 'Sadness'
    elif scores['compound'] < 0:
        emotion = 'Anger'
    elif scores['compound'] == 0:
        emotion = 'Neutral'
    else:
        emotion = 'Mixed emotion'
    return emotion
```

This function first analyzes the sentiment of the input text using the `polarity_scores` method of the sentiment analyzer. The result is a dictionary containing the following keys: `pos` (positive sentiment score), `neu` (neutral sentiment score), `neg` (negative sentiment score), and `compound` (an overall sentiment score ranging from -1 for very negative to +1 for very positive).

The function prints the sentiment scores for inspection. It then determines the emotion label based on the value of the `compound` score:

- If the compound score is greater than or equal to 0.5, the emotion is labeled as **joy**
- If the compound score is less than or equal to -0.5, the emotion is labeled as **sadness**
- If the compound score is less than 0, the emotion is labeled as **anger**
- If the compound score is equal to 0, the emotion is labeled as **neutral**
- For all other cases, the emotion is labeled as **mixed emotion**

Detecting and printing the emotion

The following code calls the `detect_emotion` function with the sample text and prints the detected emotion:

```
emotion = detect_emotion(text)
print("Detected Emotion: ", emotion)
```

When this code is executed with the original sample text, the output is as follows:

```
{'neg': 0.0, 'neu': 0.423, 'pos': 0.577, 'compound': 0.9567}
Detected Emotion: Joy
```

This output shows the sentiment scores for the text and the detected emotion, which is **joy** for the given example.

If the text is changed to `"It's very sad, and I want to run breaking into cars today."` and the code is run again, the output is:

```
{'neg': 0.227, 'neu': 0.669, 'pos': 0.1, 'compound': 0.4}
Detected Emotion: Mixed emotion
```

In this case, the sentiment scores reflect a more negative and neutral tone, and the detected emotion is **mixed emotion**. This demonstrates how the function can be used to detect different emotions in text using the Natural Language Toolkit.

Named entity recognition (NER) with spaCy

This project demonstrates how to extract and classify named entities, such as people, organizations, locations, dates, and monetary values, from text using spaCy's pre-trained language model. The process involves analyzing sample text, identifying relevant entities, and visualizing them using spaCy's displaCy tool. The extracted entities are then organized into a Pandas data frame and saved as a CSV file for further analysis. This project highlights key NLP concepts such as text tokenization, entity recognition, and data visualization, making it suitable for beginners and data enthusiasts interested in real-world NLP applications.

Installing spaCy and downloading the pre-trained model

The following steps are required to set up the environment for named entity recognition using spaCy:

1. Install the spaCy library using `pip`. The following command installs the spaCy library:

```
pip install spacy
```

2. Download the pre-trained English language model provided by spaCy using the following command:

```
python -m spacy download en_core_web_sm
```

Importing libraries for named entity recognition

The following code imports the required libraries for performing named entity recognition and visualizing the results. The `spaCy` library is imported for the NER task, and `displaycy` is imported from `spaCy` to provide visualization capabilities for the recognized entities:

```
import spacy
from spacy import displacy
```

The following code imports pandas and loads the spaCy English language model, which is required for performing named entity recognition and other natural language processing tasks:

```
import pandas as pd

nlp = spacy.load("en_core_web_sm")
```

Defining the sample text and processing with spaCy

The next step is to define a sample text containing a variety of named entities, such as companies, dates, people, locations, and monetary amounts. This text is stored as a multiline string in a variable named `text`:

```
text = """
Amazon announced its quarterly earnings on July 30, 2023.
As a CEO, Andy Jassy said the company is investing $4 billion in AI technology.
Google, based in Mountain View, California, also shared its financial report.
The 2024 Summer Olympics will be held in Paris, France.
"""
```

The following code processes the sample text using the loaded `spaCy` model. This generates a `doc` object that contains tokens, named entities, and linguistic annotations:

```
doc = nlp(text)
```

The next step is to define a function that extracts named entities from the processed text. The function is named `extract_entities` and takes the `doc` object as its argument:

```
def extract_entities(doc):
    # Function body will be defined to extract named entities from doc
    pass
```

Extracting named entities into a DataFrame

The function body for `extract_entities` is constructed step-by-step. First, an empty array named `entities` is created to store the extracted entities. Then, for each entry in `doc.ents`, which contains all the detected named entities from the processed text, the function iterates through and appends relevant information to the `entities` array.

Within the loop, the function appends a dictionary to the `entities` array. The dictionary contains two keys: `Entity`, which is set to `ent.text` (the actual text of the entity, such as "Amazon"), and `Label`, which is set to `ent.label_` (the entity label, such as "ORG" for organizations).

Next, `spacy.explain(ent.label_)` is used to convert each entity label into a human-readable explanation, making the output easier to interpret.

After extracting the entities, the next step is to convert the list of entities into a Pandas DataFrame. This conversion allows for easier visualization and manipulation of the extracted data. The function is updated to return `pd.DataFrame(entities)`, which creates a DataFrame from the list of entity dictionaries.

The following code defines the complete function for extracting named entities and their labels from the processed text:

```
def extract_entities(doc):
    entities = []
    for ent in doc.ents:
        entities.append({
            "Entity": ent.text,
            "Label": ent.label_,
            "Explanation": spacy.explain(ent.label_)
```

```
            })
    return pd.DataFrame(entities)
```

With this approach, all extracted entities and their labels are organized in a tabular format, which is returned for further analysis or display.

The next step involves extracting entities into a DataFrame. The function `extract_entities` is called, passing in the processed document object obtained earlier. This function processes the document and returns the extracted named entities. The following code demonstrates how to call the entity extraction function and store the results in a DataFrame:

```
entities_df = extract_entities(doc)
```

After extracting the entities, the DataFrame containing the entities, their labels, and explanations is displayed. This is accomplished by printing the DataFrame, which presents the extracted named entities in a tabular format. The following code prints the DataFrame of extracted named entities:

```
print("Extracted named entities:")
print(entities_df)
```

A representative output is shown as follows. Entity boundaries and labels can vary slightly by spaCy model version:

```
Extracted named entities:
           Entity   Label                                    Explanation
0          Amazon     ORG           Companies, agencies, institutions, etc.
1       quarterly    DATE             Absolute or relative dates or periods
2   July 30, 2023    DATE             Absolute or relative dates or periods
3      Andy Jassy  PERSON                       People, including fictional
4      $4 billion   MONEY                 Monetary values, including unit
5              AI     GPE                          Countries, cities, states
6          Google     ORG           Companies, agencies, institutions, etc.
7   Mountain View     GPE                          Countries, cities, states
8      California     GPE                          Countries, cities, states
9            2024    DATE             Absolute or relative dates or periods
10       Olympics   EVENT  Named hurricanes, battles, wars, sports events...
11          Paris     GPE                          Countries, cities, states
12         France     GPE                          Countries, cities, states
```

This output displays the named entities, their corresponding labels, and any explanations in a structured table.

Visualizing named entities with displaCy

The next step is to visualize the named entities using `displaCy`, spaCy's built-in visualization tool. This tool provides a web-friendly format for displaying named entities directly within a notebook environment.

The following code visualizes the named entities in the processed document. The `style="ent"` argument specifies that the visualization should highlight named entities, and `jupyter=True` ensures that the rendering is compatible with Jupyter notebooks:

```
displacy.render(doc, style="ent", jupyter=True)
```

This visualization displays the text with named entities highlighted, making it easy to see which parts of the text have been identified as entities and what types they are.

Saving extracted entities to a CSV file

After visualizing the entities, the next step is to save the extracted entities to a CSV file. This allows for further analysis or sharing of the results outside the notebook environment.

The following code saves the DataFrame of extracted entities to a CSV file named `extracted_entities.csv`. The `index=False` argument ensures that the DataFrame index is not included in the CSV file, preventing the addition of unnecessary index columns:

```
entities_df.to_csv("extracted_entities.csv", index=False)
```

This command creates a CSV file containing the extracted entities and their labels, without including the DataFrame's index values.

Next, a confirmation message is printed to indicate that the file was saved successfully. This step provides feedback that the CSV file containing the extracted entities has been created as intended. The following code prints a confirmation message after saving the file:

```
print("\nEntities saved to 'extracted_entities.csv'")
```

Upon running the script, the extracted named entities are displayed. The output includes an index ranging from 0 to 12, representing the different entities identified in the text provided. The extracted entities are organized in a structured table, with each entity labeled by its type and accompanied by a plain-language explanation:

	Entity	Label	Explanation
1	Amazon	ORG	Companies, agencies, institutions, etc.
2	quarterly	DATE	Absolute or relative dates or periods
3	July 30, 2023	DATE	Absolute or relative dates or periods
4	Andy Jassy	PERSON	People, including fictional
5	$4 billion	MONEY	Monetary values, including unit
6	AI	GPE	Countries, cities, states
7	Google	ORG	Companies, agencies, institutions, etc.
8	Mountain View	GPE	Countries, cities, states
9	California	GPE	Countries, cities, states
10	2024	DATE	Absolute or relative dates or periods
11	Olympics	EVENT	Named hurricanes, battles, wars, sports events, etc.
12	Paris	GPE	Countries, cities, states
13	France	GPE	Countries, cities, states

Figure 6.4: Named entity recognition output displaying extracted entities, including organizations, dates, persons, and money values from the provided text

It is noted that the entity recognition may not always be accurate. For example, the entity "AI" is incorrectly identified as a country, city, or state, which is not correct. However, the output still provides useful information about the content that was analyzed.

The folder containing the script and output files can be checked to verify the presence of the CSV file.

This approach allows for the extraction of named entities from a variety of text sources, including large amounts of text, PDF files, and Word documents. The extracted data can be saved to a CSV file for further analysis or record-keeping.

Detecting fake news using Naive Bayes

This project builds a fake news detection model using Multinomial Naive Bayes and a labeled Kaggle dataset. The selected file contains article titles, article text, subject, date, and a binary label where 0 indicates fake news and 1 indicates real news.

The selected dataset contains columns title, text, subject, date, and label. The title column is used as the input feature in this introductory example, while label is the prediction target. Download the CSV file available at `https://www.kaggle.com/datasets/mobeenfatimah/`

`fake-news-detection-dataset-6000-news-articles`, place it in the project folder, and rename it `fakenews.csv`.

Importing libraries for fake news detection

Start by importing the necessary libraries. The following code imports the libraries:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
```

`MultinomialNB` from `sklearn.naive_bayes` is a Naive Bayes classifier suitable for text data.

Loading and inspecting the news dataset

Download the dataset, rename the CSV file to `fakenews.csv`, and place it in the project folder. The following code loads the dataset into a pandas DataFrame:

```
df = pd.read_csv('fakenews.csv')
```

The dataset should include title, text, subject, date, and label columns. In this dataset, label is 0 for fake news and 1 for real news.

Now, we need to display the first few rows of the dataset. This step is used to inspect the data and verify its structure. The following code displays the first few rows:

```
print("Dataset Sample:\n", df.head())
```

A representative preview is as follows:

```
Dataset Sample:
                                               title  \
0   WATCH: Trump Just Told All The Anti-Gay Bigot...
1  China backs U.N. call for justice in Yemen, U....
2  THE PEOPLE'S PRESIDENT: Trump Meets With Coal ...
3  WSJ REPORTER RIPS INTO DEM CANDIDATES For Thei...
4  Lawmakers aim to delay U.S. ceding control of ...

                                               text       subject  \
0  A whole lot of evangelical Trump voters just d...          News
1  GENEVA (Reuters) - China signaled on Wednesday...     worldnews
```

```
2                                                        politics
3  THE WSJ S MARY KISSEL NAILS IT ON THE DEM DEBA...      politics
4  WASHINGTON (Reuters) - Critics of a plan for t...  politicsNews

                 date  label
0   November 13, 2016      0
1  September 13, 2017       1
2         Feb 16, 2017      0
3         Nov 15, 2015      0
4  September 13, 2016       1
```

The exact rows will depend on the downloaded dataset version.

Defining features and splitting the dataset

We'll now define the features and target values. The title column contains the article headline, and the label column indicates whether the article is fake (0) or real (1). The following code defines the features and target:

```
X = df['title']
y = df['label']
```

The next step is to split the dataset into training and testing sets with stratification. Stratification ensures that both classes (fake and real news) are proportionally represented in the splits. The following code splits the dataset:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

Vectorizing news text with TF-IDF

Initialize the TF-IDF vectorizer. The TF-IDF vectorizer converts text data into TF-IDF scores. The `stop_words='english'` parameter removes common English stop words, and `max_df=0.7` ignores words appearing in more than 70 percent of documents, as these are likely too common to be useful. The following code initializes the vectorizer:

```
vectorizer = TfidfVectorizer(stop_words='english', max_df=0.7)
```

Next, we need to fit and transform the training data. The vectorizer learns the vocabulary from the training data and transforms it into TF-IDF vectors. The following code fits and transforms the training data:

```
X_train_tfidf = vectorizer.fit_transform(X_train)
```

Similarly, transform the test data. The test data is transformed into TF-IDF vectors using the same vocabulary learned from the training data. The following code transforms the test data:

```
X_test_tfidf = vectorizer.transform(X_test)
```

Training the multinomial Naive Bayes classifier

Initialize the Naive Bayes classifier. The Multinomial Naive Bayes classifier is effective for text classification tasks where features such as words are multinomially distributed. The following code initializes the classifier:

```
model = MultinomialNB()
```

Train the model**.** The model is trained using training data. The following code trains the model:

```
model.fit(X_train_tfidf, y_train)
```

Make predictions on the test set. The trained model is used to classify the articles in the test set. The following code makes predictions:

```
y_pred = model.predict(X_test_tfidf)
```

Evaluating the model

The `zero_division=0` parameter avoids warnings for labels with undefined metrics by assigning zero to those values:

```
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, zero_division=0)
print(f"\nAccuracy: {accuracy:.3f}")
print(f"\nClassification Report:\n{report}")
```

A representative evaluation output is as follows:

```
Accuracy: 0.898
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.90      0.90      0.90       600
           1       0.90      0.90      0.90       600

    accuracy                           0.90      1200
   macro avg       0.90      0.90      0.90      1200
weighted avg       0.90      0.90      0.90      1200
```

The output displays the accuracy, which in this case is 0.898, and the classification report, which includes precision, recall, F1 score, support, macro average, and weighted average.

The model can also be used to test a particular news article by providing a sample input and observing the prediction. For example, a fake news article can be created and passed to the model to see how it is classified.

Building a resume scanner using keyword extraction

This project focuses on building a resume scanner that utilizes keyword extraction to analyze resumes and determine whether candidates possess the required skills for a job. The scanner processes the text of a resume, extracts relevant keywords, and enables recruiters to efficiently match candidates to specific job requirements. The approach uses **TF-IDF** (term frequency-inverse document frequency) to identify important keywords from the resume text and compare them with a predefined set of job keywords.

Importing libraries for resume scanning

Start by creating a Python file. The following code imports the necessary libraries:

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
```

The next import is the **cosine similarity** function, which measures the similarity between documents based on their TF-IDF features. This metric is useful for comparing resumes to job descriptions:

```
from sklearn.metrics.pairwise import cosine_similarity
```

Creating the sample resume dataset

A sample data object is then defined, representing several resumes. Each resume is stored in a dictionary with a unique resume ID and corresponding resume text that details the candidate's skills. The following code creates this data structure:

```
data = {
    'resume_id': [1, 2, 3],
    'resume_text': [
    'Experienced data scientist with skills in Python, machine learning, and data
analysis.',
    'Software developer with expertise in Java, cloud computing, and project
management.',
    'Data analyst with proficiency in SQL, Python, and data visualization.'
]
}
```

In practice, resumes can also be read from PDF or DOC formats using specialized tools, and then structured into a similar dictionary format. For demonstration purposes, the sample resumes are kept concise, but additional text can be included as needed.

Defining the job description

The job description is then defined as a string containing the required skills for the position. This text will be compared to each resume to assess the match:

```
job_description = 'Looking for a data scientist skilled in Python, machine
learning, SQL, and data analysis.'
```

The next step is to convert the dictionary of resumes into a pandas DataFrame, which simplifies analysis and manipulation. The following code performs this conversion and prints the resulting DataFrame to display its structure:

```
df = pd.DataFrame(data)
print(df)
```

The output is as follows:

```
      resume_id                resume_text
0          1    Experienced data scientist with skills in Python...
1          2    Software developer with expertise in Java, cloud...
2          3    Data analyst with proficiency in SQL, Python, an...
```

Vectorizing resumes and computing similarity scores

To prepare for TF-IDF vectorization, the resumes and job description are combined. The resume text column is converted to a list of strings, which will be used as the set of documents for vectorization:

```
documents = df['resume_text'].to_list()
```

The following code appends the job description to the list of documents, enabling computation of TF-IDF vectors for both the resumes and the job description:

```
documents.append(job_description)
```

Next, the TF-IDF vectorizer is initialized with English stop words removed. This step ensures that common words such as "and," "the," and "of" are excluded from the analysis:

```
vectorizer = TfidfVectorizer(stop_words='english')
```

The vectorizer is then fit to the documents and transforms each document into a TF-IDF vector, resulting in a TF-IDF matrix:

```
tfidf_matrix = vectorizer.fit_transform(documents)
```

To compare the job description with each resume, similarity scores are calculated using cosine similarity. The following code computes the similarity between the job description (the last document in the TF-IDF matrix) and each resume (all documents except the last one). The result is flattened into a one-dimensional array for easier handling:

```
similarity_scores = cosine_similarity(tfidf_matrix[-1],
tfidf_matrix[:-1]).flatten()
```

The similarity scores are now displayed by adding them as a new column in the DataFrame. This associates each resume with its similarity to the job description. The following code creates a new column called `similarity_score` in the DataFrame and assigns the similarity scores to it:

```
df['similarity_score'] = similarity_scores
```

To display the similarity scores alongside the resume IDs, the following code selects and prints these columns:

```
print("Resume similarity scores:")
print(df[['resume_id', 'similarity_score']])
```

The output is:

```
Resume similarity scores:
   resume_id  similarity_score
0          1          0.662564
1          2          0.000000
2          3          0.418121
```

Filtering and displaying matching resumes

The next step is to identify which resumes match the job description based on a similarity threshold. This threshold can be adjusted as needed. In this case, a threshold of 0.2 is used, meaning that any resume with a similarity score of 0.2 or higher is considered a match for the job requirements.

To filter the resumes and identify those that meet the similarity threshold, the following code selects only the rows where the similarity score is greater than or equal to the threshold value:

```
threshold = 0.2
matching_resumes = df[df.similarity_score >= threshold]
```

This step creates a new DataFrame containing only the resumes that have a similarity score above the specified threshold.

To display the IDs and similarity scores of the matching resumes, the following code selects these columns and prints them:

```
print("Resumes matching the job requirements:")
print(matching_resumes[['resume_id', 'similarity_score']])
```

This output shows which resumes are considered matches for the job description based on the chosen threshold.

The script displays the resumes, which are labeled as one, two, and three, along with their respective roles. For example, the first resume is for an "experienced data scientist." The

similarity scores are then calculated based on the job requirements. In this case, the job requirement is for a data scientist:

```
Resumes matching the job requirements:
   resume_id  similarity_score
0          1          0.662564
2          3          0.418121
```

The similarity score for the first resume, which matches the job title "data scientist," is 0.66. The second resume is for a software developer with expertise in Java, which does not align with the data science requirement, resulting in a similarity score of 0.00. The third resume is for a data analyst with proficiency in SQL. Since the job requirement includes SQL and data analysis, this resume receives a similarity score of 0.41.

Based on these results, the resumes that match the job requirements are the first and third resumes. This demonstrates how the code can be used to create a resume scanner using only keyword extraction.

Automatic essay grading using BERT

The project focuses on building an automatic essay grading system using **BERT** (bidirectional encoder representations from transformers) to evaluate and assign scores to essays. The evaluation is based on the content, coherence, and relevance of each essay.

The prerequisites for this project include the installation of several libraries: pandas, scikit-learn, torch, matplotlib, and seaborn. These libraries are required for data manipulation, model building, and visualization.

The project uses the following dataset: `https://www.kaggle.com/c/asap-aes`. A Kaggle account and acceptance of the competition rules may be required to access the data.

The source file is tab-separated rather than a standard CSV. This project keeps the `essay` and `domain1_score` columns for Essay Set 1, then renames `domain1_score` to `overall` so the remaining code is easy to follow.

Importing libraries for essay grading

The following code imports all the necessary libraries for building an essay grading system using BERT. These include libraries for data manipulation, numerical computation, model evaluation, visualization, and neural network construction:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import BertTokenizer, BertModel
from torch.optim import AdamW
```

`BertTokenizer` performs text preprocessing, `BertModel` provides pre-trained contextual embeddings, and `torch.optim.AdamW` is used to optimize the neural network parameters.

Loading and exploring the dataset

The dataset is loaded from `training_set_rel3.tsv`. The tab separator and file encoding are specified, Essay Set 1 is selected, and domain1_score is renamed to overall. The first five rows are then displayed:

```
df = pd.read_csv(
    'training_set_rel3.tsv',
    sep='\t',
    encoding='ISO-8859-1'
)
df = df[df['essay_set'] == 1][['essay', 'domain1_score']]
df = df.rename(columns={'domain1_score': 'overall'}).dropna()
print(df.head())
```

The first five rows of the dataset are as follows:

```
                                               essay  overall
0  Dear local newspaper, I believe computers have...        8
1  Computers are useful in schools because they...          9
2  The author explains several benefits of technology...    7
3  Students should learn to use computers responsibly...   10
4  In my opinion, technology can support learning...        8
```

Confirming the selected columns

After filtering and renaming, the DataFrame contains only essay and overall. Removing missing rows ensures that every training example has both text and a valid score:

```
df = df[['essay', 'overall']].dropna()
```

Preprocessing the data

A pre-trained BERT tokenizer is loaded for lowercase English text. This tokenizer will be used to preprocess the essay text for input into the BERT model:

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

Creating a custom dataset class

A custom dataset class is defined to handle the essays and their scores. This class inherits from the abstract dataset class and is responsible for initializing the data, returning the number of essays, and retrieving individual items with proper tokenization and tensor conversion. The code for this function is as follows:

```
class EssayDataset(Dataset):
    def __init__(self, essays, scores, tokenizer, max_length=512):
    self.essays = essays
    self.scores = scores
    self.tokenizer = tokenizer
    self.max_length = max_length

    def __len__(self):
        return len(self.essays)

    def __getitem__(self, index):
        essay = str(self.essays[index])
        score = self.scores[index]

        encoding = self.tokenizer.encode_plus(
        essay,
        add_special_tokens=True,
        max_length=self.max_length,
        padding='max_length',
        truncation=True,
        return_attention_mask=True,
        return_tensors='pt'
    )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
```

```
            'score': torch.tensor(score, dtype=torch.float)
        }
```

This class performs the following:

- Initializes with essays, scores, tokenizer, and maximum sequence length
- Returns the number of essays in the dataset
- Retrieves a specific essay and score, tokenizes the essay, and returns the input IDs, attention mask, and score as PyTorch tensors

Splitting the dataset into training and validation sets

The dataset is split into training and validation sets using an 80-20 split. The `essay` column is used as the feature, and the `overall` column is used as the target. The following code performs this split:

```
train_texts, val_texts, train_scores, val_scores = train_test_split(df['essay'],
df['overall'], test_size=0.2, random_state=42)
```

This split assigns 80 percent of the data to training and 20 percent to validation, ensuring reproducibility with a fixed random state.

The following code creates the training and validation datasets by instantiating the `EssayDataset` class with the appropriate data splits and tokenizer:

```
train_dataset = EssayDataset(train_texts.tolist(), train_scores.tolist(),
tokenizer)
val_dataset = EssayDataset(val_texts.tolist(), val_scores.tolist(), tokenizer)
```

This step prepares the data for use in the model by converting the training and validation texts and scores into lists and passing them, along with the tokenizer, to the dataset class.

Next, the code sets up data loaders for both the training and validation datasets. The data loaders handle batching and shuffling of the data during training and evaluation:

```
train_loader = DataLoader(train_dataset, batch_size=8, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=8)
```

The `train_loader` shuffles the data and processes it in batches of eight samples per iteration. The `val_loader` also uses a batch size of eight but does not shuffle the validation data. The batch size can be adjusted based on available computational resources.

Defining the essay grading model

The following code defines the essay grading model as a subclass of `torch.nn.Module`. This model uses a pre-trained BERT model to extract embeddings and a linear layer to predict a single score for each essay:

```
class EssayGradingModel(torch.nn.Module):
    def __init__(self):
        super(EssayGradingModel, self).__init__()
        self.bert = BertModel.from_pretrained('bert-base-uncased')
        self.regressor = torch.nn.Linear(self.bert.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        cls_output = outputs.pooler_output
        score = self.regressor(cls_output)
        return score.squeeze(-1)
```

In this model, the BERT component extracts features from the input essay, and the regressor reduces the output to a single predicted score. The `forward` method processes the input tensors and returns the predicted score.

Initializing the model, optimizer, and loss function

The following code initializes the model, sets the computation device, and defines the optimizer and loss function:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = EssayGradingModel().to(device)
optimizer = AdamW(model.parameters(), lr=2e-5)
loss_fn = torch.nn.MSELoss()
```

This setup moves the model to the appropriate device (GPU if available, otherwise CPU), creates an `AdamW` optimizer with a learning rate of `2e-5`, and specifies mean squared error as the loss function for training.

Training the essay grading model

The following code defines a function named `train` that performs one epoch of training for the essay grading model. This function takes the model, a data loader, the loss function, the optimizer, and the computation device as inputs. The function sets the model to training mode, iterates over each batch in the data loader, performs a forward pass, computes the loss,

backpropagates the gradients, updates the model weights, and accumulates the total loss. At the end, it returns the average loss per batch:

```
def train(model, data_loader, loss_fn, optimizer, device):
    model.train()
    total_loss = 0
    for batch in data_loader:
        optimizer.zero_grad()
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        scores = batch['score'].to(device)

        outputs = model(input_ids=input_ids, attention_mask=attention_mask)
        loss = loss_fn(outputs, scores)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(data_loader)
```

This function performs a forward pass, calculates the loss, and updates the model weights for each batch in the training data.

Evaluating the essay grading model

The following code defines an `evaluate()` function that assesses the model's performance on a validation set. The function takes the model, a data loader, the loss function, and the device as parameters. It sets the model to evaluation mode, disables gradient computation, and iterates over the validation data. For each batch, it computes the model's predictions and the loss, collects the predictions and true scores, and finally calculates the MSE and R2 metrics. The function returns the average loss, MSE, and R2 score:

```
def evaluate(model, data_loader, loss_fn, device):
    model.eval()
    predictions = []
    true_scores = []
    total_loss = 0
    with torch.no_grad():
        for batch in data_loader:
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            scores = batch['score'].to(device)
```

```
            outputs = model(input_ids=input_ids,attention_mask=attention_mask)
            loss = loss_fn(outputs, scores)
            total_loss += loss.item()
            predictions.extend(outputs.detach().cpu().numpy().tolist())
            true_scores.extend(scores.detach().cpu().numpy().tolist())
    mse = mean_squared_error(true_scores, predictions)
    r2 = r2_score(true_scores, predictions)
    return total_loss / len(data_loader), mse, r2
```

This function evaluates model performance using MSE and R-squared metrics.

Setting up the training loop

The following code sets up and runs the training loop for the essay grading model. The number of epochs is set to three. For each epoch, the model is trained on the training data and evaluated on the validation data. The training loss, validation loss, MSE, and R2 score are printed for each epoch:

```
epochs = 3
for epoch in range(epochs):
    train_loss = train(model, train_loader, loss_fn, optimizer, device)
    val_loss, mse, r2 = evaluate(model, val_loader, loss_fn, device)
    print(f"Epoch {epoch + 1}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.
4f},
MSE: {mse:.4f}, R2: {r2:.4f}")
```

This loop iterates through the specified number of epochs, training and evaluating the model, and prints the results for each epoch.

Testing the essay grading model

The next step is to test the trained model on a sample essay. The following code defines a test essay and encodes it using the tokenizer. The encoded input includes special tokens, is padded to a maximum length of 512, and is returned as a tensor:

```
test_text = (
    "The essay provided insightful analysis of the topic with well-structured
arguments."
)
encoding = tokenizer.encode_plus(
    test_text,
    add_special_tokens=True,
    max_length=512,
```

```
    padding='max_length',
    truncation=True,
    return_tensors='pt'
)
```

This prepares the test essay for input into the model by converting it to the required tensor format.

Next, the input IDs are set by assigning the encoded input IDs to the device. The following code performs this operation:

```
input_ids = encoding['input_ids'].to(device)
```

The attention mask is then created by assigning the encoded attention mask to the device:

```
attention_mask = encoding['attention_mask'].to(device)
```

The model is set to evaluation mode to ensure that layers such as dropout behave appropriately during inference:

```
model.eval()
```

To generate a prediction, the following code is used. It disables gradient calculation, passes the input through the model, and retrieves the predicted score:

```
with torch.no_grad():
    predicted_score = model(input_ids, attention_mask=attention_mask).item()
```

The predicted score is then printed, which provides the model's assessment of the sample essay. The expected score for this essay is approximately four to five points:

```
print(f"Predicted score: {predicted_score:.2f}")
```

A representative prediction is as follows:

```
Predicted score: 4.23
```

The score will vary with the trained model and selected essay set.

Visualizing training and validation loss

The next step is to visualize the training data and validation loss over the epochs. This helps in understanding how the model's performance evolved during training. The following code defines the loss values for three epochs:

```
losses = {
    'epoch': [1, 2, 3],
    'train_loss': [6.89, 0.75, 0.53],
    'val_loss': [0.69, 1.19, 0.91]
}
```

A pandas DataFrame is created to organize the loss data:

```
loss_df = pd.DataFrame(losses)
```

A line plot is then generated using seaborn to display both the training and validation loss. The following code creates the plot for the training loss:

```
sns.lineplot(data=loss_df, x='epoch', y='train_loss', marker='o', label='Train
Loss')
```

A second line plot is added for the validation loss:

```
sns.lineplot(data=loss_df, x='epoch', y='val_loss', marker='o', label='Validation
Loss')
```

The plot is given a title, and the plot is displayed:

```
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```

The resulting training and validation loss plot is as follows:

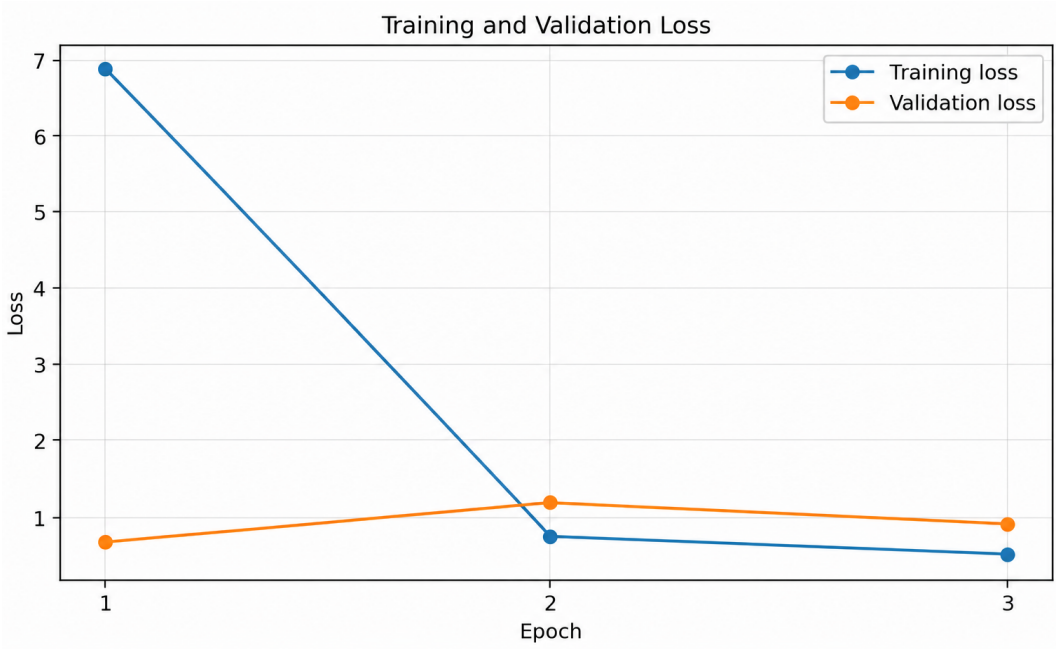


Figure 6.5: Training and validation loss curves across three epochs during BERT- based essay grading model training

The plot shows a sharp reduction in training loss. Validation loss fluctuates across the three epochs, which indicates that generalization is less stable and that additional tuning or more data may be needed.

This completes the application, with all components implemented and visualized.

This script is designed to run for three epochs, which may require between 30 minutes and an hour to complete, depending on the hardware and dataset size. During execution, the script displays the data sample and initial information, then proceeds to train the model over the specified epochs.

After the training is completed, the results are displayed. The output includes a graph showing the training and validation loss for each epoch, as well as detailed metrics for each epoch. The dataset sample is shown first, followed by the results for each epoch:

- For epoch one, the training loss is 6.89, the validation loss is 0.69, the MSE is 0.69, and the R2 score is 0.333

- For epoch two, the training loss drops to 0.75, the validation loss is 1.19, the MSE is 1.18, and the R2 score is -0.14
- For epoch three, the training loss further decreases to 0.53, the validation loss is 0.91, the MSE is 0.91, and the R2 score is 0.165

The predicted score for the sample essay is 4.23, which represents the model's assessment of essay grading.

This project demonstrates that deep learning models such as BERT can be used to automate complex tasks like essay grading.

Recommendation system projects

This section demonstrates three approaches to building recommendation systems. The first uses cosine similarity on genre-based TF-IDF vectors to recommend similar movies. The second implements item-based collaborative filtering to suggest books based on user rating patterns. The third uses singular value decomposition (SVD) via the surprise library to predict how a user would rate a movie based on historical interaction data. Together, these projects introduce the two main paradigms in recommendation systems: content-based filtering and collaborative filtering.

Building a movie recommendation system using cosine similarity

This project demonstrates how to build a **movie recommendation system** using cosine similarity. The goal of this system is to recommend movies that are similar to a given movie by measuring the similarity between their feature vectors, specifically using the genres associated with each movie.

The following steps outline the process of building this recommendation system in Python.

Importing libraries for movie recommendation

The first step is to import the necessary libraries. The following code imports these libraries:

```
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
```

Creating the sample movie dataset

A sample dataset is created containing five movies. Each movie has an ID, a title, and a genre description. The genres are represented as strings, such as "action sci-fi" or "crime drama". The following code defines the sample dataset:

```
movie_data = {
    'movie_id': [1, 2, 3, 4, 5],
    'title': ['The Matrix', 'John Wick', 'Godfather', 'Pulp Fiction', 'Dark
Knight'],
    'genre': ['Action, Sci-Fi', 'Action , Thriller', 'Crime, Drama', 'Crime,
Drama', 'Action, Crime, Drama']}
```

Converting the dataset to a DataFrame

The dataset is then converted into a pandas DataFrame for easier manipulation and analysis. The following code creates the DataFrame:

```
df = pd.DataFrame(movie_data)
```

To display the dataset, the DataFrame can be printed:

```
print("Movie Data: ")
print(df)
```

This will produce an output as follows:

```
Movie Data:
   movie_id         title              genre
0         1    The Matrix        action sci-fi
1         2     John Wick     action thriller
2         3     Godfather         crime drama
3         4  Pulp Fiction         crime drama
4         5   Dark Knight  action crime drama
```

Defining the TF-IDF vectorizer

A **TF-IDF vectorizer** is defined to transform the genre text into numerical vectors. The vectorizer is set to remove English stop words, which are common words that do not contribute significant meaning. The following code defines the vectorizer:

```
tfidf = TfidfVectorizer(stop_words='english')
```

Transforming the genre column

The `genre` column of the DataFrame is transformed into a matrix of TF-IDF features. Each row in this matrix represents a movie, and each column represents a genre-related word. The values are the TF-IDF scores, which indicate the importance of each word in the genre description. The following code performs this transformation:

```
tfidf_matrix = tfidf.fit_transform(df['genre'])
```

Computing the cosine similarity matrix

The next step is to compute the **cosine similarity matrix**. This matrix contains the pairwise cosine similarity scores between all movies based on their genre vectors. Each element at position `(i, j)` in the matrix represents the similarity between movie `i` and movie `j`. The following code computes the cosine similarity matrix:

```
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

Defining the recommendation function

A function is defined to recommend movies based on cosine similarity. This function, `getRecommendations`, takes the title of a movie and the cosine similarity matrix as input and returns the titles of the most similar movies.

The function follows these steps:

1. Get the index of the movie that matches the given title
2. Get the pairwise similarity scores of all movies with that movie
3. Sort the movies based on the similarity scores in descending order
4. Get the indices of the two most similar movies, excluding the movie itself
5. Get the movie indices
6. Return the titles of the most similar movies

The following code defines the recommendation function:

```
def getRecommendations(title, cosine_sim):
    # 1. Get the index of the movie that matches the title
    idx = df[df['title'] == title].index[0]

    # 2. Get the pairwise similarity scores of all movies with that movie
    sim_scores = list(enumerate(cosine_sim[idx]))
```

```
    # 3. Sort the movies based on the similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # 4. Get the indices of the two most similar movies (excluding the movie
itself)
    sim_scores = sim_scores[1:3]

    # 5. Get the movie indices
    movie_indices = [i[0] for i in sim_scores]

    # 6. Return the titles of the most similar movies
    return df['title'].iloc[movie_indices]
```

Testing the recommendation system

The recommendation system can be tested by providing a movie title and printing the recommended movies.

The following code tests the system with the movie "The Matrix":

```
movie_title = 'The Matrix'
recommended_movies = getRecommendations(movie_title, cosine_sim)

print(f"Movie recommendations for {movie_title}:")
for movie in recommended_movies:
    print(movie)
```

On running the code, we get the following output:

```
Movie recommendations for The Matrix:
Dark Knight
John Wick
```

When "The Matrix" is used as the input, the system recommends "Dark Knight" and "John Wick" as the most similar movies. This is because "The Matrix" has the genre "Action Sci-Fi", while "Dark Knight" has "Action Crime Drama" and "John Wick" has "action thriller". The shared "action" genre leads to higher similarity scores.

If "Godfather" is used as the input, the system recommends "Pulp Fiction" and "Dark Knight", since these movies share the "crime" and "drama" genres, and does not recommend "The Matrix" or "John Wick", which do not have "crime" or "drama" in their genres.

This example demonstrates how a movie recommendation system can be built using cosine similarity on genre-based feature vectors.

Building a book recommendation system using collaborative filtering

This project demonstrates how to build a book recommendation system using collaborative filtering. Collaborative filtering is a popular technique used in recommendation systems where recommendations are based on user-item interactions. In this case, the item-based collaborative filtering approach is used to recommend books based on user preferences.

The cosine similarity measure is used to find similar books based on users' ratings. For this example, a sample dataset of book ratings by users is created.

Importing libraries for book recommendation

Start by creating the Python file and importing the required libraries:

```
import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity
```

Cosine similarity calculates the cosine similarity between vectors, which is used to compute similarity between books based on user ratings.

Creating a sample book ratings dataset

The following code creates a sample dataset representing user ratings for four books, then converts this data into a pandas DataFrame and prints it:

```
data = {
    'user_id': [1, 1, 1, 2, 2, 3, 3, 3, 4, 4],
    'book_title': ['Book A', 'Book B', 'Book C', 'Book A', 'Book D',
              'Book B', 'Book C', 'Book E', 'Book A', 'Book C'],
    'rating': [5, 3, 4, 4, 5, 5, 3, 4, 3, 2]
}
df = pd.DataFrame(data)
print("Dataset:\n", df)
```

The output is:

```
Dataset:
     user_id book_title  rating
0        1     Book A       5
1        1     Book B       3
```

```
2        2     Book A       4
3        2     Book C       2
4        3     Book B       5
5        3     Book D       4
6        4     Book C       3
7        4     Book D       5
```

Building the user-book matrix

The next step is to create a **user-book matrix** using a pivot table. This matrix has users as rows, books as columns, and ratings as values. The following code creates the user-book matrix and prints it:

```
user_book_matrix = df.pivot_table(index='user_id', columns='book_title',
values='rating', fill_value=0)
print("\nUser-Book Matrix:\n", user_book_matrix)
```

The output is:

```
User-Book Matrix:
book_title    Book A  Book B  Book C  Book D
user_id
1                5       3       0       0
2                4       0       2       0
3                0       5       0       4
4                0       0       3       5
```

Computing user similarity scores

To calculate the cosine similarity between users based on their book ratings, the following code is used. It computes the similarity matrix and prints it:

```
user_similarity = cosine_similarity(user_book_matrix)
user_similarity_df = pd.DataFrame(user_similarity, index=user_book_matrix.index,
columns=user_book_matrix.index)
print("\nUser Similarity Matrix\n", user_similarity_df)
```

The output is:

```
User Similarity Matrix
          1         2         3         4
1  1.000000  0.707107  0.235702  0.000000
```

```
2  0.707107  1.000000  0.000000  0.235702
3  0.235702  0.000000  1.000000  0.963624
4  0.000000  0.235702  0.963624  1.000000
```

Defining the book recommendation function

The following function recommends books for a given user based on user similarity. It takes the user ID, similarity matrix, user-book matrix, and the number of top recommendations to return. If the user ID is not found in the similarity matrix, it returns an empty array. Otherwise, it retrieves similarity scores for the user, sorts them in descending order, and drops the user's own ID from the results:

```
def recommend_books(user_id, similarity_matrix, user_book_matrix, top_n=3):
    if user_id not in similarity_matrix.index:
        print("User not found in the data set.")
        return []

    similar_users =
similarity_matrix.loc[user_id].sort_values(ascending=False).drop(user_id)
    # Further recommendation logic would go here
```

The next step involves aggregating the ratings from similar users, with each rating weighted by the corresponding similarity score. This process is implemented by first initializing an empty dictionary to store the recommended books.

For each similar user and their similarity score in the collection of similar users, a variable is created to hold the books rated by that user. This is accomplished by accessing the relevant row in the user-book matrix for the similar user.

Within this context, a nested loop iterates over each book and its rating in the set of rated books, considering only those books where the rating is greater than zero. For each such book, a check is performed to determine whether the target user has not already rated the book. This is done by verifying that the book is either not present in the user-book matrix for the target user or that the rating for that book by the target user is zero.

If the book passes this check, the recommendation dictionary is updated. The score for the book is incremented by the product of the similar user's rating and the similarity score. If the book is not already present in the dictionary, it is initialized with a score of zero before the increment.

After all similar users and their rated books have been processed, the recommendation dictionary contains the aggregated scores for each candidate book. The books are then sorted in descending order based on their aggregated scores. The top recommendations are selected

from this sorted list and returned to the caller of the function. The following code implements this logic within the recommendation function:

```
def recommend_books(user_id, similarity_matrix, user_book_matrix, top_n=3):
    if user_id not in similarity_matrix.index:
        print("User not found in the data set.")
        return []

    similar_users =
similarity_matrix.loc[user_id].sort_values(ascending=False).drop(user_id)
    # Further recommendation logic would go here
    recommended_books = {}
    for similar_user, similarity in similar_users.items():
        rated_books = user_book_matrix.loc[similar_user]
        for book, rating in rated_books.items():
            if rating > 0:
                if user_book_matrix.loc[user_id, book] == 0:
                    recommended_books[book] = recommended_books.get(book, 0) +
rating * similarity
        recommended_books = sorted(recommended_books.items(), key=lambda x: x[1],
reverse=True)
        return [book for book, score in recommended_books[:top_n]]
```

This function returns a list of the top recommended books for the specified user, based on the aggregated and weighted ratings from similar users.

Generating book recommendations for users

To generate recommendations for a specific user, the function is called with the desired user ID, the user similarity DataFrame, the user-book matrix, and the number of top recommendations to return. For example, to obtain recommendations for user ID 1, the function is invoked with these parameters, specifying that the top three recommendations are desired.

The following code demonstrates how to retrieve recommended books for a particular user:

```
user_id=1
recommended_books = recommend_books(user_id, user_similarity_df, user_book_matrix,
top_n=3)
print(f"\nBooks recommended for User {user_id}:", recommended_books)
```

When this code is executed, the recommended books for user 1 are displayed. In this scenario, user 1 has already read books A, B, and C. The function examines the user similarity matrix and identifies books that user 1 has not read. The output for user 1 is as follows:

```
Books recommended for User 1: ['Book D', 'Book E']
```

Here, books D and E are recommended because user 1 has not read them, while A, B, and C are already in the user's reading history.

To illustrate further, consider user 4, who has read books A and C. The function is called with user ID 4, and the recommendations are generated in the same manner:

```
user_id=4
recommended_books = recommend_books(user_id, user_similarity_df, user_book_matrix,
top_n=3)
print(f"\nBooks recommended for User {user_id}:", recommended_books)
```

The output for user 4 is:

```
Books recommended for User 4: ['Book B', 'Book D', 'Book E']
```

In this case, books B, D, and E are recommended for user 4, as these are the books that the user has not yet read.

This process completes the implementation of a book recommendation system using collaborative filtering.

Predicting movie ratings using collaborative filtering

This project focuses on building a movie rating prediction system using **collaborative filtering**, where the model predicts how a user would rate a movie based on historical user-movie interactions. The objective is to predict movie ratings for users using collaborative filtering techniques such as **matrix factorization**, specifically **singular value decomposition (SVD)**.

Downloading and preparing the MovieLens dataset

The dataset used for this project is the *MovieLens* **dataset**, which is available on the MovieLens website (`https://grouplens.org/datasets/movielens/`). Several versions are provided, including smaller datasets suitable for education and development, as well as larger datasets for research purposes. For this project, the small version is used, which contains files such as `links.csv`, `movies.csv`, `readme.txt`, `tags.csv`, and most importantly, `ratings.csv`.

The `ratings.csv` file contains four columns:

- `userId`: Unique identifier for users
- `movieId`: Unique identifier for movies
- `rating`: Rating given by the user, ranging from 1 to 5
- `timestamp`: Time of the rating, which is not essential for this project

Tip

The timestamp column is not required for collaborative filtering and is removed during preprocessing.

Importing libraries for rating prediction

The following code imports the necessary libraries for data manipulation, numerical computation, collaborative filtering, evaluation, and visualization:

```
import pandas as pd
import numpy as np    from surprise import SVD  # Singular value decomposition
algorithm for collaborative filtering
from surprise import Dataset, Reader
# Dataset builder and rating scale/structure definition
from surprise.model_selection import cross_validate, train_test_split
from surprise import accuracy  # Metrics such as RMSE for model evaluation
import seaborn as sns
import matplotlib.pyplot as plt
```

If the `surprise` library is not installed, it can be added using the following command:

```
pip install surprise
```

Loading and previewing the dataset

The dataset is loaded using pandas, and the timestamp column is dropped as it is not relevant for collaborative filtering. The first few rows of the dataset are displayed to provide a preview:

```
df = pd.read_csv('ratings.csv')
df.drop('timestamp', axis=1, inplace=True)
print(df.head())
```

The output will be:

```
    userId  movieId  rating
0       1        1     4.0
1       1        3     4.0
2       1        6     4.0
3       1       47     5.0
4       1       50     5.0
```

Preprocessing the dataset for the surprise library

The dataset is structured for use with the `surprise` library by defining a `reader` object with the appropriate rating scale and loading the data into a compatible format:

```
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(df[['userId', 'movieId', 'rating']], reader)
```

The `reader` object specifies the range of ratings in the dataset. The `Dataset.load_from_df` method loads the data into a format expected by `surprise`, which requires three columns: `userId`, `movieId`, and `rating`.

Splitting the dataset into training and test sets

The dataset is split into training and testing sets using an 80-20 split, with a fixed random state for reproducibility:

```
trainset, testset = train_test_split(data, test_size=0.2, random_state=42)
```

- `test_size=0.2` reserves 20% of the data for testing
- `random_state=42` ensures the split is reproducible

The `trainset` is used to train the model, while the `testset` is used for evaluation.

Building and training the SVD model

The **SVD** model is initialized and trained on the training dataset:

```
model = SVD()
model.fit(trainset)
```

The SVD model predicts ratings for user-item pairs based on latent factors derived from the data.

Evaluating the model with RMSE

The model is evaluated on the test dataset by generating predictions and calculating the **root mean square error (RMSE)**, which measures the difference between predicted and actual ratings. Lower RMSE values indicate better performance. The code is as follows:

```
predictions = model.test(testset)
rmse = accuracy.rmse(predictions)
print(f"RMSE: {rmse:.4f}")
```

It produces the following output:

```
RMSE: 0.8784
```

The `predictions` variable contains a list of predicted ratings for the user-item pairs in the test set.

Predicting a specific user and movie

A prediction is made for a specific user and movie by providing their IDs to the model. The estimated rating is extracted and printed:

```
user_id = 196
movie_id = 242
pred = model.predict(user_id, movie_id)
print(f"Predicted rating for user {user_id} on movie {movie_id}: {pred.est:.2f}")
```

The output is:

```
Predicted rating for user 196 on movie 242: 3.62
```

The `model.predict` method returns the predicted rating for the specified user and movie.

Visualizing the distribution of movie ratings

The distribution of movie ratings is visualized using a histogram with five bins and a kernel density estimate for smoothness. The plot size, title, and axis labels are set for clarity:

```
plt.figure(figsize=(10, 6))
sns.histplot(df['rating'], bins=5, kde=True)
plt.title('Distribution of Movie Ratings')
plt.xlabel('Rating')
```

```
plt.ylabel('Frequency')
plt.show()
```

The plot in *Figure 6.6* shows the distribution of movie ratings, with most ratings clustered around four, but a good spread across the available rating values:

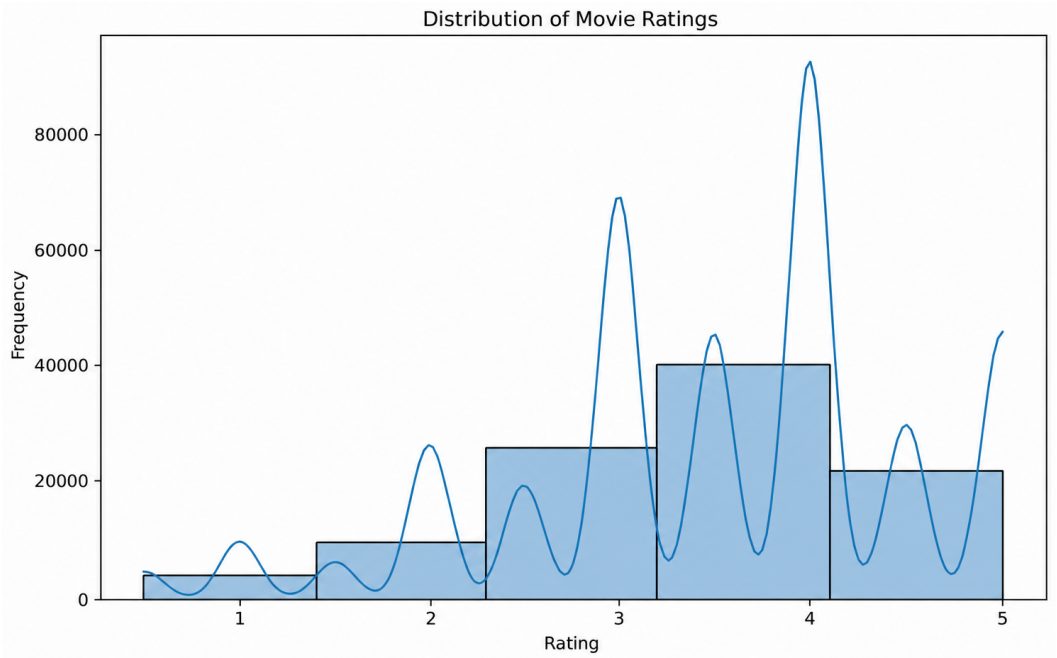


Figure 6.6: Histogram showing the distribution of movie ratings across the dataset

The right-skewed distribution reveals that users tend to assign higher ratings, indicating a general bias towards positive evaluation in collaborative filtering datasets

AI and game projects

This section applies AI techniques to interactive and game-based contexts. The first project implements the minimax algorithm to build an unbeatable tic-tac-toe AI that evaluates all possible game states to select the optimal move. The second builds a simple voice-controlled personal assistant in Python using speech recognition and text-to-speech libraries, capable of responding to queries about the time, performing web searches, and opening applications.

Building a tic-tac-toe AI with the minimax algorithm

The **Minimax algorithm** is a recursive approach used for making decisions in turn-based games such as Tic-Tac-Toe. This algorithm enables the AI to simulate all possible moves,

evaluate them, and select the optimal move to maximize its own chance of winning or minimize the opponent's chance of winning.

The process begins by preparing the project environment. Create a new Python file to get started.

Representing the Tic-Tac-Toe board

The Tic-Tac-Toe board is represented as a one-dimensional list containing nine elements, with indexes ranging from 0 to 8. Each element is initialized as a space, indicated by a pair of quotes with a space character. This signifies that the slot is available for a move. The following code does this:

```
Board = [' ' for _ in range(9)]
```

Printing the Tic-Tac-Toe board

A function is defined to print the Tic-Tac-Toe board in a user-friendly format. This function divides the one-dimensional list into three rows, each containing three elements, and prints them as rows separated by vertical bars to mimic the appearance of a traditional tic-tac-toe board. The function iterates through each element and prints the vertical bars to separate the elements visually:

```
def print_board(board):
    for row in [board[i*3:(i+1)*3] for i in range(3)]:
        print('| ' + ' | '.join(row) + ' | ')
```

Checking for a winner

A function named `check_winner` is created to determine if there is a winner. This function takes the board and the player as arguments. The win conditions are as follows:

- Horizontal rows: positions 0, 1, 2; 3, 4, 5; and 6, 7, 8
- Vertical columns: positions 0, 3, 6; 1, 4, 7; and 2, 5, 8
- Diagonals: positions 0, 4, 8 and 2, 4, 6

The function also checks each win condition. If the elements at the specified positions are all equal to the player's symbol, the function returns true, indicating a win. Otherwise, it indicates that there is no winner yet. This is how the code will look:

```
def check_winner(board, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8], # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8], # Columns
```

```
        [0, 4, 8], [2, 4, 6]             # Diagonals
    ]

    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] ==
player:
            return True
    return False
```

Checking if the board is full

A function is defined to check if the tic-tac-toe board is full. This function determines whether any empty slots are remaining on the board. If all slots are filled, the board is considered full, which may indicate a win or a tie. The following code defines the function to check if the board is full:

```
def is_board_full(board):
    return " " not in board
```

Evaluating the board for the minimax algorithm

An `evaluate` function is created to assign a score to the current board state for use in the minimax algorithm. This function checks for a winner and returns a score based on the outcome:

- If the AI (represented by "0") wins, the function returns 1
- If the human player (represented by "X") wins, the function returns -1
- If neither player has won and the game is ongoing or ends in a draw, the function returns 0

The following code defines the `evaluate` function:

```
def evaluate(board):
    if check_winner(board, "0"):
        return 1
    elif check_winner(board, "X"):
        return -1
    else:
        return 0
```

Implementing the minimax algorithm

The **minimax** function is implemented to calculate the best move for the AI. This function uses recursion to simulate all possible moves and outcomes, assigning scores to each possible board state. The function takes the following arguments: the current board, the current depth of recursion, and a boolean value `is_maximizing` to indicate whose turn it is.

- If `is_maximizing` is `True`, it is the AI's turn, and the algorithm attempts to maximize the score
- If `is_maximizing` is `False`, it is the human player's turn, and the algorithm attempts to minimize the score

The function first evaluates the board. If the score is 1 (AI wins), -1 (human wins), or the board is full, it returns the score. Otherwise, it recursively explores all possible moves:

- For the AI's turn, it initializes the best score to negative infinity and tries all possible moves, updating the best score with the maximum value returned by recursive calls.
- For the human player's turn, it initializes the best score to positive infinity and tries all possible moves, updating the best score with the minimum value returned by recursive calls.
- After each simulated move, the function undoes the move to restore the board state.

The following code implements the minimax function:

```
def minimax(board, depth, is_maximizing):
    score = evaluate(board)
    if score == 1 or score == -1 or is_board_full(board):
        return score

    if is_maximizing:
        best_score = float('-inf')
        for i in range(9):
            if board[i] == " ":
                board[i] = "0"  # AI makes the move
                best_score = max(best_score, minimax(board, depth + 1, False))
                board[i] = " "  # Undo the move
        return best_score

    else:
        best_score = float('inf')
        for i in range(9):
            if board[i] == " ":
```

```
                board[i] = "x"  # Human makes the move
                best_score = min(best_score, minimax(board, depth + 1, True))
                board[i] = " "  # Undo the move
        return best_score
```

Finding the best move for the AI

The next step is to define the function that determines the best move for AI. This function is named `find_best_move` and takes the current board as its argument. The process involves initializing two variables: `best_value` is set to negative infinity, and `best_move` is set to -1.

The function then iterates through all nine positions on the board. For each position, it checks if the slot is empty. If the slot is empty, the AI simulates making a move by placing its marker ("0") in that position. The function then calls the minimax algorithm with the updated board, a depth of 0, and `is_maximizing` set to `False` to evaluate the move.

After evaluating, the function undoes the move by resetting the slot to empty. If the value returned by minimax is greater than the current `best_value`, the function updates `best_value` and records the position as `best_move`.

The following code defines the function to find the best move for the AI:

```
def find_best_move(board):
    best_value = float('-inf')
    best_move = -1
    for i in range(9):
        if board[i] == " ":
            board[i] = "0"  # AI makes the move
            move_value = minimax(board, 0, False)
            board[i] = " "  # Undo the move
            if move_value > best_value:
                best_value = move_value
                best_move = i
    return best_move
```

This approach ensures that the function loops over all available positions, simulates the AI making a move at each position, and uses the minimax algorithm to evaluate the outcome. The move with the highest value is selected as the best move for the AI.

Implementing the main game loop

The main game loop is responsible for managing the flow of the tic-tac-toe game, alternating between the human player's move and the AI's move, and updating the board accordingly. The function is defined as `play_game`.

Within this function, a loop continues to run as long as the game is ongoing. At each iteration, the current state of the board is displayed by calling a function to print the board.

The next step is to handle the human player's move. The function prompts the player to enter a move by requesting input for a slot number between 1 and 9. The player's input is stored in the variable representing the player's move.

A check is performed to determine if the chosen slot on the board is already occupied. If the slot is not empty, the function prints a message indicating that the move is invalid and prompts the player to try again by calling the function recursively. If the slot is empty, the function places an "X" on the selected slot to register the player's move.

After the human player makes a move, the game checks if the player has won by calling the `check_winner` function with the board and "X" as arguments. If the player has achieved a winning combination, the board is printed, a message "You win" is displayed, and the loop is exited. This check ensures that the game recognizes a human victory immediately after the move is made:

```
def play_game():
    board = [" "] * 9
    while True:
        print_board(board)
        player_move = int(input("Enter your move (1-9): ")) - 1
        if board[player_move] != " ":
            print("Invalid move. Try again.")
            continue
        board[player_move] = "X"

        if check_winner(board, "X"):
            print_board(board)
            print("You win!")
            break
```

If the player has not won, the next step is to check for a draw. The function `is_board_full` is called with the board as an argument. If the board is full, the current state of the board is

printed, the message "It's a draw" is displayed, and the loop is exited. This prevents further moves if no empty slots remain.

If neither a win nor a draw has occurred, it is the AI's turn. The game prints a message indicating that the AI is making its move. The AI's move is determined by calling the `find_best_move` function, which returns the optimal position for the AI to play. The AI's move is then placed on the board at the chosen position.

After the AI makes its move, the game checks if the AI has won by calling `check_winner` with the board and "0" as arguments. If the AI wins, the board is printed and the message "AI wins" is displayed. If the AI has not won, the game checks again for a draw by calling `is_board_full`. If the board is full, the board is printed and the message "It's a draw" is displayed, and the loop is exited.

This sequence completes the main game loop, alternating between the human and AI moves and checking for win or draw conditions after each move:

```
if is_board_full(board):
    print_board(board)
    print("It's a draw!")
    break

print("AI is making its move...")
ai_move = find_best_move(board)
board[ai_move] = "0"

if check_winner(board, "0"):
    print_board(board)
    print("AI wins!")
    break

if is_board_full(board):
    print_board(board)
    print("It's a draw!")
    break

play_game()
```

The entire `play_game()` function looks like this:

```
def play_game():
    board = [" "] * 9
```

```
    while True:
        print_board(board)
        player_move = int(input("Enter your move (1-9): ")) - 1
        if board[player_move] != " ":
            print("Invalid move. Try again.")
            continue
        board[player_move] = "X"

        if check_winner(board, "X"):
            print_board(board)
            print("You win!")
            break

        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break

        print("AI is making its move...")
        ai_move = find_best_move(board)
        board[ai_move] = "0"

        if check_winner(board, "0"):
            print_board(board)
            print("AI wins!")
            break

        if is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break

play_game()
```

When the game starts, the board is displayed as a three-by-three grid of empty slots. The human player selects a slot by entering a number between 1 and 9. For example, if the player chooses the fifth slot, the AI responds by selecting the top slot. If the player then selects the seventh slot, the AI places its move in another available position. This interaction continues, with the AI calculating and making its best possible move after each player action.

The following visualization shows a game board mid-play with positions occupied by both the player and the AI:

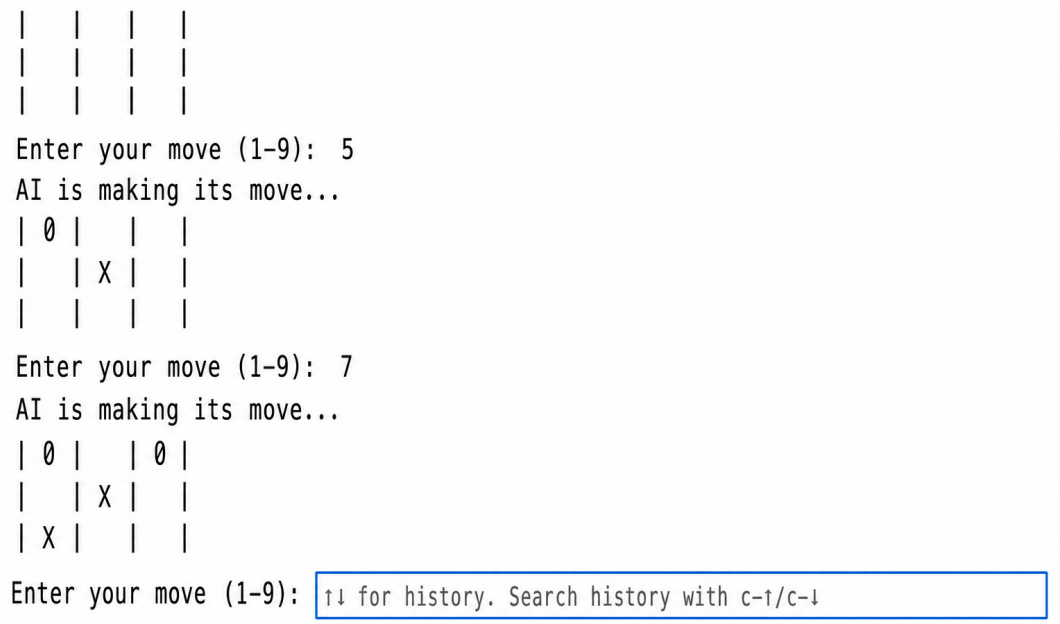


Figure 6.7: Tic-tac-toe game board visualization showing player and AI moves during gameplay

So, in this scenario, the player must place a second marker to prevent the AI from taking that spot. The game progresses and, as anticipated, it results in a tie. The minimax algorithm makes it very difficult for the human player to win if the goal is to force a draw. The AI attempts to win by selecting position four, and the player responds by choosing position nine. The outcome is a draw.

This demonstrates that the tic-tac-toe AI built with the minimax algorithm is highly effective at preventing losses. Winning against it is extremely challenging if the player is aiming for a tie. This completes the implementation of a tic-tac-toe AI using the minimax algorithm.

Building a simple personal assistant using Python speech libraries

Let's build a simple personal assistant in Python that listens to voice commands, performs basic tasks such as searching the web, checking the time, or greeting the user, and responds with audio feedback. The **speech recognition** library is used to recognize speech, and the `pyttsx3` library is used to synthesize speech for responses.

To begin, create a Python file. The next step is to open a terminal and create a Python virtual environment. Creating a dedicated environment is important because it allows the installation of required libraries without affecting the entire system. The following steps are performed to set up the environment:

1. Create a virtual environment named `speech_environment` by running:

```
python -m venv speech_environment
```

2. Activate the virtual environment by running:

```
source speech_environment/bin/activate
```

   Once the environment is activated and the correct file is open, the necessary libraries are installed. The first library is `SpeechRecognition`, which is used for speech recognition. The second library is `pyttsx3`, which is used for speech synthesis so the assistant can respond vocally.

   The following commands install the required libraries:

3. Install the speech recognition library:

```
pip install speechrecognition
```

4. Install the `pyttsx3` library:

```
pip install pyttsx3
```

After installation, the libraries are available for use in the project.

Importing libraries for the personal assistant

The next step is to import the necessary libraries in the Python script. We'll use the following libraries:

- The `speech_recognition` library, often imported as `sr`, is used for converting spoken words to text using Google's speech recognition API
- The `pyttsx3` library is used for text-to-speech, allowing the assistant to respond vocally
- The `datetime` module provides date and time information, which is useful for responding to time-related queries

- The `webbrowser` module is used to open a web browser for performing searches
- The `os` module allows the assistant to run system commands, such as opening Notepad or Calculator

The following code imports the required libraries:

```
import speech_recognition as sr
import pyttsx3
import datetime
import webbrowser
import os
```

Initializing the speech engine

The speech engine for text-to-speech is initialized using the following command:

```
engine = pyttsx3.init()
```

Creating the speak function

A function is defined to make the assistant speak. This function takes a string input, `text`, and speaks it out loud using the initialized speech engine. The following code defines the `speak` function:

```
def speak(text):
    print(f"Assistant: {text}")
    engine.say(text)
    engine.runAndWait()
```

This function prints each response to the console, queues the same text for speech with `engine.say`, and uses `engine.runAndWait` to play the audio. Printing the response also creates a readable transcript for testing.

Taking voice commands

A function is created to take a voice command from the user. The function uses the speech recognition library to listen to the user's voice through the microphone and convert the audio to text using Google's speech recognition API. The following code defines the `take_command` function:

```
def take_command():
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
```

```
        print("Listening...")
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

        try:
            print("Recognizing...")
            command = recognizer.recognize_google(audio)
            print(f"User said: {command}")
        except sr.UnknownValueError:
            print("Sorry, I did not understand that.")
            return None
        except sr.RequestError:
            print("Network error.")
            return None
    return command.lower()
```

This function performs the following actions:

- Creates a recognizer object from the speech recognition library
- Uses the microphone as the audio source
- Prints a message indicating that the assistant is listening
- Adjusts for ambient noise to improve recognition accuracy
- Listens to the source and saves the audio
- Attempts to recognize the speech using Google's API
- Prints the recognized command for confirmation
- Handles errors for unknown values and network issues, returning `None` in those cases
- Returns the recognized command in lowercase if successful

Creating the respond function

The next function defined is the `respond` function. This function handles the assistant's responses based on the command received from the user. The function takes the `command` as an argument and checks for specific keywords or phrases to determine the appropriate response. The following code defines the `respond` function:

```
def respond(command):
    if 'hello' in command or 'hi' in command:
        speak("Hello. How can I assist you today?")
    elif 'time' in command:
        current_time = datetime.datetime.now().strftime("%I:%M %p")
```

```
        speak(f"The current time is {current_time}")
    elif 'search' in command:
        speak("What would you like to search for?")
        query = take_command()

        if query:
            speak(f"Searching for {query}")
            webbrowser.open(f"https://www.google.com/search?q={query}")
    elif 'open' in command and 'safari' in command:
        os.system("open -a Safari")
        speak("Opening Safari")

    elif 'open' in command and 'calculator' in command:
        os.system("open -a Calculator")
        speak("Opening Calculator")

    elif 'bye' in command or 'exit' in command or 'quit' in command:
        speak("Goodbye. Have a great day")
        exit()
    else:
        speak("I'm sorry, I do not know the command")
```

This function performs the following actions:

- Checks if the command contains a greeting such as "hello" or "hi". If so, the assistant responds with a greeting.
- Checks if the command contains the word "time". If so, the assistant retrieves the current time, formats it, and speaks the result.
- Checks if the command contains the word "search". If so, the assistant asks what to search for, listens for the user's query, and then opens a web browser to search for the query on Google.
- Checks if the command contains "open" and "safari". If so, the assistant opens the Safari browser using the appropriate system command and confirms the action.
- Checks if the command contains "open" and "calculator". If so, the assistant opens the Calculator application and confirms the action.
- Checks if the command contains "bye", "exit", or "quit". If so, the assistant says goodbye and exits the program.
- If none of the above conditions are met, the assistant responds that it does not recognize the command.

Running the assistant

The main function that initiates the assistant is called `run_assistant`. This function begins by speaking a greeting and then continuously listens for commands from the user. Each command is processed using the previously defined respond function. The following code defines the `run_assistant` function:

```
def run_assistant():
    speak("Hello. I am your assistant. How can I help you?")
    while True:
        command = take_command()
        if command:
            respond(command)
```

This function performs the following actions:

- The assistant greets the user and asks how it can help
- Enters a continuous loop where it listens for commands
- Each command is passed to the respond function for processing
- The loop continues until the user says "bye", "exit", or "quit", at which point the assistant exits

Starting the application

To start the personal assistant application, the `run_assistant()` function is called. This begins the interaction loop, allowing the assistant to listen and respond to commands. The following code starts the assistant:

```
run_assistant()
```

The following terminal output demonstrates a typical interaction sequence with the voice-controlled personal assistant:

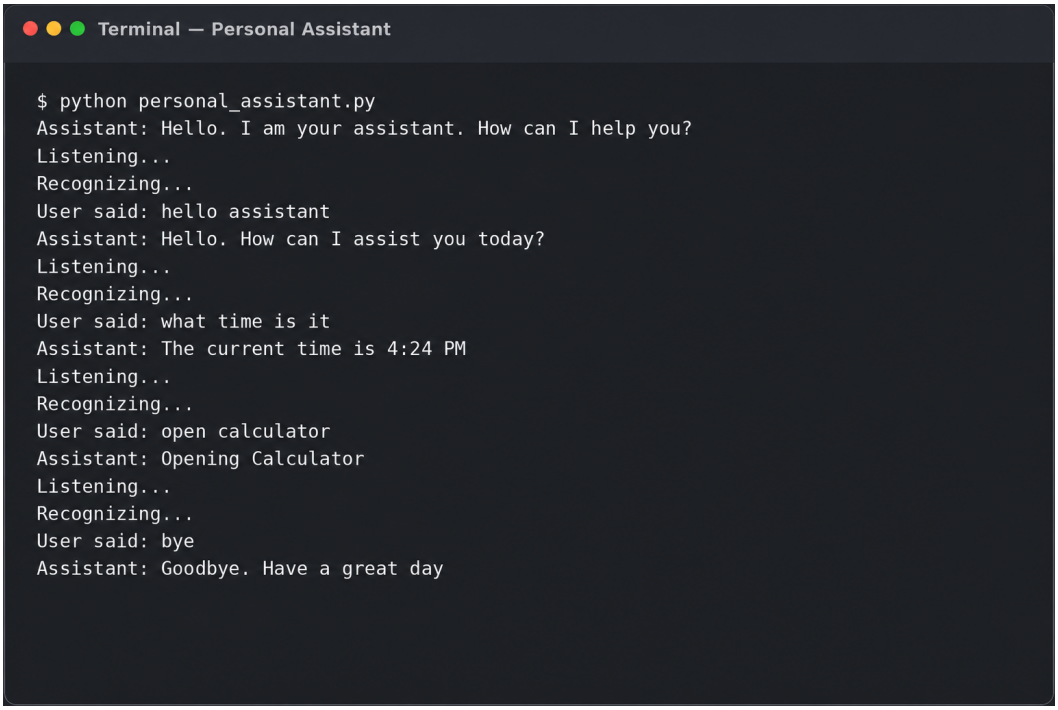


Figure 6.8: Terminal output demonstrating a sample interaction session between the user and the voice-controlled assistant

The integration of speech recognition and text-to-speech enables natural human-computer interaction, showcasing how voice-based interfaces can automate routine tasks.

Summary

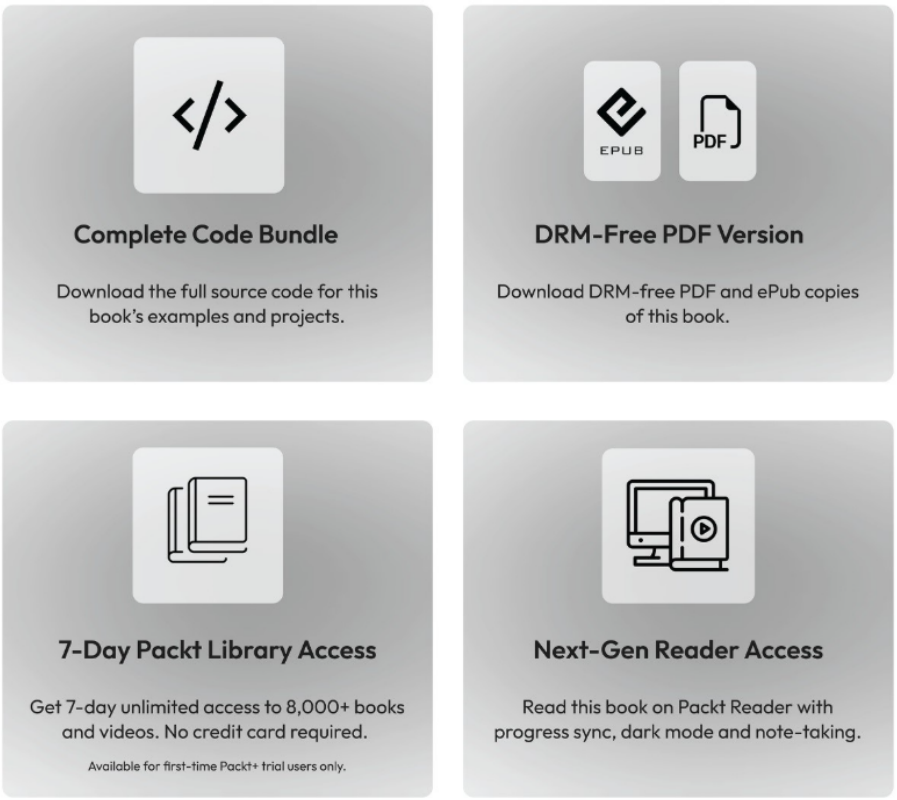
This chapter brought together the skills and techniques developed throughout the book, applying them to hands-on projects spanning deep learning and computer vision, natural language processing, recommendation systems, and AI-driven game development. Projects ranged from building image classifiers and sentiment analysis models to implementing recommendation engines and a voice-controlled personal assistant, demonstrating how Python and AI libraries can be combined to solve a wide variety of real-world problems.

The projects in this book have provided a practical foundation in AI and Python, equipping you with the tools and experience needed to continue building intelligent applications independently.

7

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

> **Note**
>
> **Note**: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 7.1* on desktop), search for this book by name and select the correct edition.

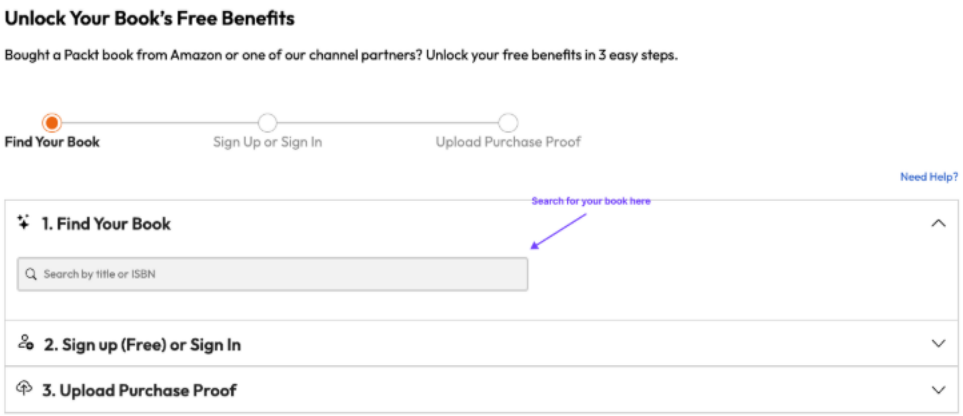


Figure 7.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

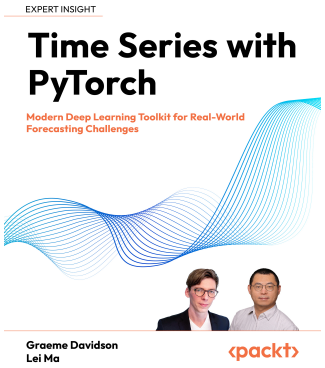
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Time Series with PyTorch

Graeme Davidson, Lei Ma

ISBN: 9781805120421

- Build, train, and evaluate neural networks for time series using PyTorch and PyTorch Lightning. Tune models with Bayesian optimisation and validate them with suitable metrics and strategies.
- Progress from feedforward and recurrent networks to transformers and models such as N-BEATS, N-HiTS, and TFT.
- Learn how global models use cross- and transfer learning across many series.
- Generate synthetic series and representations with diffusion and self-supervised methods.
- Apply modern approaches to classification, clustering, and anomaly detection.



Generative AI with Python and PyTorch

Joseph Babcock, Raghav Bali

ISBN: 9781835884454

- Grasp the core concepts and capabilities of LLMs
- Craft effective prompts using chain-of-thought, ReAct, and prompt query language to guide LLMs toward your desired outputs
- Understand how attention and transformers have changed NLP
- Optimize your diffusion models by combining them with VAEs
- Build text generation pipelines based on LSTMs and LLMs
- Leverage the power of open-source LLMs, such as Llama and Mistral, for diverse applications

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.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.

Share your thoughts

Now you've finished *Hands-On AI Development with Python*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



`https://packt.link/r/1808088530`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

AI and game projects **255**
- personal assistant, building with Python speech libraries 263 – 269
- tic-tac-toe AI, building with minimax algorithm 255 – 263

API endpoint
- testing 105, 106

accuracy **59**

arithmetic functions
- defining 110, 111

arithmetic operators **8**

array operations **15**

arrays
- creating, with NumPy 14

automatic essay grading system
- building, with BERT 233
- custom dataset class, creating 235, 236
- data, preprocessing 235
- dataset, loading and exploring 234
- dataset, splitting into training and validation sets 236
- defining 237
- evaluating 238, 239
- initializing 237
- libraries, importing for 233, 234
- loss function 237
- optimizer 237
- selected columns, confirming 234
- testing 239, 240
- training 237, 238
- training data and validation loss, visualizing 241 – 243
- training loop, setting up 239

B

backward pass **200**

basic descriptive statistics **36**

bidirectional encoder representations from transformers (BERT)
- used, for building automatic essay grading system 233

book recommendation system, with collaborative filtering
- building 247
- function 249, 250
- generating, for users 250, 251
- libraries, importing for 247
- sample book ratings dataset, creating 247
- user similarity scores, computing 248
- user-book matrix, building 248

C

CSV file
- creating 17, 18
- extracted entities, saving to 224, 225

California Housing Prices dataset
- reference link 43

California housing dataset
- loading and inspecting 50

calculator function
- creating 111

car price prediction model
- categorical features, encoding 142, 143
- dataset, splitting into training and test sets 144
- implementing, with random forest model 140
- implementing, with random forest regressor 140

sample car dataset, creating 141, 142
categorical data encoding **30**
chatbot, in Python
building, with predefined responses 113
response dictionary, defining 114
running 115 – 117
user input, matching to responses 114, 115
classification model **57**
accuracy 59
confusion matrix 59
evaluating 58
F1 score 60
in ML 57
logistic regression for 57
precision 59
recall 60
classification report **216**
collaborative filtering
used, for building book recommendation system 247
used, for building movie rating prediction system 251
command-line calculator
implementing, in Python 110
confusion matrix
visualizing 65, 87, 88
control flow, in Python **9**
while loop statements 10, 11
for loop statements 10
if-else statements 9
convolutional neural network (CNN) model
architecture 193, 194
compiling 194
constructing 185, 186
dataset URL, preparing for cat and dog image classification 191
image classification results, interpreting 197
image data generators, defining for augmentation and rescaling 191, 192
libraries, importing for image classification 191
predictions, making on new images 196, 197
summary, printing 194
training 194
training and validation accuracy and loss, plotting 195, 196
training and validation data, loading 192, 193
used, for classifying images of dogs and cats 190
validation data generator, defining 192
cosine similarity matrix
computing 245
credit card fraud detection model
building, with scikit-learn 159
dataset, loading and preparing 160
dataset, splitting and scaling 160
libraries, importing for 159
performance, evaluating 161 – 163
random forest classifier, initializing and training 161
customer churn prediction model
building, with logistic regression 154, 155
categorical features, preprocessing and encoding 157
dataset, loading and inspecting 155 – 157
dataset, splitting into training and test sets 157
evaluating 158, 159
features, scaling 157, 158
libraries, importing for training 155
157, 158

D

DataFrames **16**
named entities, extracting into 222 – 224
Dense layer **73**
data

loading and inspecting, in EDA 22 – 24
loading and inspecting, with pandas 18, 19
structures, inspecting 24, 25
visualizing, with matplotlib and seaborn 30
data manipulation 21, 22
performing, in pandas 19, 20
data transformation and feature engineering 29
categorical data encoding 30
feature scaling 29, 30
data types 8
dataset preparation 43
decision boundary
visualizing 60 – 62
decision tree classifier
evaluating 138
libraries, importing for 135
training 137
used, for classifying Iris flower dataset 135
visualizing, for Iris flower dataset 139, 140
deep learning frameworks 70
deep learning, for visual and structural tasks 182
dogs and cats images, classifying with CNN 190 – 197
image classifier, building with Keras and TensorFlow 183 – 190
NN, building with NumPy 197 – 203
descriptive statistics
types 35
diabetes prediction model
confusion matrix, visualizing 133, 134
dataset, inspecting 130, 131
dataset, splitting into training and test sets 131
features and target variables, selecting 131
implementing, with logistic regression model 129, 130
performance, evaluating 132
predictions, making with patient data 132
testing, with new patient record 134, 135
dictionaries, in Python
working with 13, 14
displaCy
used, for visualizing named entities 224

E

emotion detection, in text
NLTK, using for 217
emotion detection, with NLTK
detecting and printing 220
function, writing for 218, 219
libraries, importing for 217
required NLTK data, downloading 218
sample text, creating for 218
sentiment analyzer, importing and initializing 218
employee attrition prediction model
confusion matrix, visualizing 153, 154
dataset, preprocessing 151
dataset, splitting into training and testing sets 152
features and target variable, splitting 152
performance, evaluating 152
predictions, making 152
with XGBoost classifier 149 – 151
exploratory data analysis (EDA) 22
data structures, inspecting 24, 25
data, loading and inspecting 22 – 24
missing data, handling 25, 26
missing data, handling techniques 27 – 29
missing data, identifying 26, 27
performing 51, 52
performing, on Titanic dataset 36 – 39

F

F1 score 60

Flask application
creating 93 – 95
deploying, to Heroku 102 – 105
installing 93
prediction route, creating 99
preparing, for deployment 102, 103
web service and API, testing 105, 106
Flask web form
creating, for input 98, 99
fake news detection model
building, with multinomial Naive Bayes 225
dataset, splitting 227
evaluating 228, 229
features, defining 227
libraries, importing for 226
news dataset, loading and inspecting 226, 227
news text, vectorizing with TF-IDF 227, 228
fake product review detection
dataset, splitting 214
features, extracting 214
libraries, importing for 213
review text, vectorizing with TF-IDF 215
reviews dataset, loading and inspecting 214
training and evaluating 216, 217
with NLP techniques 213
feature scaling **29, 30, 47**
flatten layer **73**
for loop statements **10**
forecasting and prediction projects **169**
stock prices, predicting with linear regression 173 – 179
weather forecasting, using historical data 169 – 173
forward pass **199**
frequency-based extractive approach, with NLTK
building 209
creating 210 – 212
importing 209, 210
input text, defining 210
resources, downloading 209, 210
functions, in Python **11, 12**

H

Heroku
Flask application, deploying to 102 – 105
setting up 102
Heroku CLI
working with 102
Hugging Face **83**
Human Activity Recognition (HAR) dataset
confusion matrix, visualizing 148, 149
downloading and preparing 145
libraries, importing for 145, 146
model performance, evaluating 147, 148
predictions, making on test set 147
preprocessing 146
smartphones dataset, with random forest using for 145
splitting, into training and test sets 147
handwritten digit classification
with neural network 78 – 80
heart disease risk prediction, with ML model
confusion matrix, visualizing 167
dataset, loading and exploring 163
dataset, preprocessing 164
dataset, splitting into training and test sets 165
disease risk, predicting in new patient data 168, 169
selection and evaluation 166
house prices, with linear regression model
actual versus predicted values, visualizing 55, 56
California housing dataset, loading and inspecting 50
coefficients, displaying 56, 57, 127, 128
data, loading and preparing 125, 126
data, splitting into training and test sets 126
evaluating 54

exploratory data analysis, performing 51, 52
features and target variables, preparing 53
features and target, selecting 126
libraries, importing 49
making predictions 127
performance, evaluating 127
predicting 124
predictions, making 54
testing, with new input 128, 129
train-test split, creating 53, 54

I

IMDB dataset
using, for sentiment analysis model 89, 90
Iris flower dataset
classifying, with decision tree classifier 135
decision tree, visualizing for 139, 140
features and target species, splitting 137
loading and preparing 136
splitting, into training and test sets 137
if-else statements **9**
image classification
building, with Keras and TensorFlow 183 – 190
libraries, importing for 183
interpretive statistics **36**

J

Jupyter Notebook
installing 6
simple program, writing 7

K

Kaggle
URL 22
Keras
used, for building image classification 183 – 190

L

lasso regression **48**
lemmatization **82**
linear regression model **43**
building 45, 46
performance, evaluating 46, 47
training 54, 127
used, for predicting house price 48
logistic regression classifier **216**
building 58
classification model, evaluating 58
decision boundary, visualizing 60 – 62
using, for spam detection 62
logistic regression model
evaluating 64, 65
for classification model 57
training 64, 119, 132
used, for building customer churn prediction model 154, 155
used, for implementing diabetes prediction model 129, 130
loss function **198**

M

MBA Admission dataset
reference link 23
MNIST classification
neural network, building for 71
MNIST dataset
loading and preprocessing 184
overview 70, 71
Matplotlib
working with 30 – 34
MovieLens dataset
downloading and preparing 251, 252
machine learning (ML) model
building, to predict heart disease risk 163
classification model in 57

data, splitting into training and test sets 45
feature scaling 47
regularization 48
supervised learning 43
types 43
unsupervised learning 43

machine learning projects 124
car prices, predicting with random forest 140 – 145
credit card fraud, detecting with scikit-learn 159 – 163
customer churn, predicting with logistic regression 154 – 159
diabetes, predicting with logistic regression 129 – 135
employee attrition, predicting with XGBoost 152 – 154
employee attrition, predicting with XGBoost classifier 149 – 151
HAR dataset, using smartphones dataset with random forest 145 – 149
house prices, predicting with linear regression model 124 – 129
iris flower dataset, classifying with decision trees 135 – 140

machine translation 80

matrix factorization 251

matrix operations 15, 16

mean squared error (MSE) 46, 49, 176

min-max scaling 29

minimax algorithm
implementing 258
Tic-Tac-Toe board, evaluating for 257
used, for building Tic-Tac-Toe AI 255

missing data
handling 25, 26
handling, techniques 27 – 29
identifying 26, 27

model coefficients
displaying 57

model evaluation
performing, with metrics 87

movie rating prediction system
building, with collaborative filtering 251
dataset, loading and previewing 252
dataset, preprocessing for surprise library 253
dataset, splitting into training and test sets 253
distribution, visualizing of 254, 255
libraries, importing for 252
model, evaluating with RMSE 254
MovieLens dataset, downloading and preparing 251, 252
specific user and movie, predicting 254
SVD model, building and training 253

movie recommendation system, with cosine similarity
building 243
dataset, converting to DataFrames 244
function, defining 245
genre column, transforming 245
libraries, importing for 243
sample movie dataset, creating 244
testing 246, 247
TF-IDF vectorizer 244

multinomial Naive Bayes
classifier, training 228
used, for building fake news detection model 225

multiple machine learning models
training 165

N

NLTK corpus 204

Naive Bayes classifier
evaluating 206, 207

most informative features, displaying 207
training 206
Natural Language Toolkit (NLTK)
movie reviews dataset, loading and preprocessing from 205
required data, downloading 218
used, for implementing sentiment analysis on text data 203 – 209
using, for emotion detection in text 217
NumPy **14**
array operations 15
matrix operations 15, 16
neural network, building with 197
used, for creating arrays 14
named entity recognition (NER) **80**
extraction and visualization, with spaCy 220
extracted entities, saving to CSV file 224, 225
extracting, into DataFrames 222 – 224
libraries, importing for 221
pre-trained models, downloading 221
processing, with spaCy 221
sample text, defining 221
spaCy, installing 221
visualizing, with displaCy 224
natural language processing (NLP) **80**
machine translation 80
NER 80
pre-trained models, using for 83 – 85
speech recognition 80
text classification 80
natural language processing (NLP), techniques **203**
automatic essay grading, with BERT 233 – 243
emotions in text, detecting with NLTK 217 – 220
fake news, detecting with Naive Bayes 225 – 229
fake product reviews, detecting with 213 – 216
NER, with spaCy 220 – 225
resume scanner, building with keyword extraction 229 – 233
sentiment analysis, on text data with NLTK 203 – 209
text summarization, with NLTK 209 – 213
neural network **69**
activation functions 75 – 77
building, for MNIST classification 71
building, with NumPy 197
components 69
concepts 69
evaluating 75
predictions, making 75
using, for handwritten digit classification 78 – 80
neural network, for MNIST classification
architecture, defining 73, 74
data pre-processing 72
model, compiling 74
training 74
neural network, with NumPy
activation and loss functions 198
backward pass and weight updates, implementing 200, 201
class, initializing 198, 199
forward pass, implementing 199
training loop, implementing 201, 202
training, with XOR dataset 202, 203
normalization **29**

O

one-hot encoding **30**

P

PyTorch **70**
Python

arithmetic functions, defining 110, 111
calculator function, creating 111
calculator, running 112, 113
command-line calculator, implementing 110
foundational building blocks 7
installing 3 – 6
URL 3

Python lists
working with 12, 13

Python personal assistant
building 263, 264
libraries, importing for 264
respond function, creating 266, 267
running 268
speak function, creating 265
speech engine, initializing 265
starting with 268, 269
voice commands, creating 265, 266

Python, foundational building blocks
control flow 9 – 11
functions 11, 12
operators 8
variables and data types 8

pair plot
creating, with Seaborn 35

pandas 16, 118
CSV file, creating 17, 18
data manipulation, performing 19, 20
used, for loading and inspecting data 18, 19

pre-trained NLP models
using, for sentiment analysis model 88, 89

pre-trained models
downloading 221
using, for NLP 83 – 85

precision 59

prediction function
building 96, 97

prediction route, in Flask
logic, implementing 100
model and vectorizer, loading 100
testing 101

pyplot module 118

R

R-squared 46, 49

ReLU activation function 73 – 77

random forest classifier
initializing and training 161
training 147

random forest model
used, for implementing car price prediction model 140

random forest regressor
training and evaluating 144, 145
used, for implementing car price prediction model 140

read mode (r) 21

recall 60

recommendation systems project
book recommendation system, building with collaborative filtering 247 – 251
building 243
movie rating prediction system, with collaborative filtering 251 – 255
movie recommendation system, building with cosine similarity 243 – 246

regularization 48

resume scanner, with keyword extraction
building 229
job description, defining 230
libraries, importing for 229
matching resumes, filtering and displaying 232, 233
sample resume dataset, creating 230
similarity scores, computing 231, 232
TF-IDF vectorization 231, 232

ridge regression 48

root mean square error (RMSE)
used, for evaluating model 254

S

Softmax activation function **74 – 77**

Surprise library
dataset, preprocessing for 253

scikit-learn **118**
used, for building credit card fraud detection model 159

seaborn **118**
using, for advanced data visualization 34, 35

sentiment analysis
classifier, testing with custom movie review sentences 208, 209
features extracted for 205
function, defining 207, 208
implementing, on text data with NLTK 203 – 209
libraries, implementing for 204

sentiment analysis model **81**
building, with TensorFlow 85 – 87
text preprocessing 81 – 83
with pre-trained NLP models 88, 89
with TensorFlow and IMDB dataset 89, 90

sigmoid activation function **198**

singular value decomposition (SVD) model **251**
building and training 253

smartphones dataset, with random forest
using, for HAR dataset 145

spaCy
installing 221
processing with 221
used, for extraction and visualization of NER 220

spam detection, with logistic regression classifier
confusion matrix, visualizing 65
data, loading and inspecting 63
features and target, preparing 63
libraries, importing 62
predictions, making 64
train-test split, creating 63

spam email detector, with scikit-learn
building 117
dataset, downloading and preparing 117
dataset, loading and splitting 118, 119
libraries, importing for 117, 118
logistic regression model, training 119
results, visualizing 122 – 124
trained model, evaluating 122 – 124

speech recognition **80**

stemming **81**

stock price, with linear regression **173**
actual versus predicted stock prices, visualizing 176, 177
closing prices, predicting on test data 175
dataset, splitting into training and test sets 175
features and target variables, preparing 174
performance, evaluating 176
predicting, for future date 178, 179
sample historical stock price dataset, creating 173

supervised learning model **43 – 45**

T

TF-IDF vectorization **213**
news text with 227, 228
review text with 215

TF-IDF vectorizer **244**

TensorFlow **70**
used, for building image classification 183 – 190
used, for building sentiment analysis model 85 – 87
using, for sentiment analysis model 89, 90

Tic-Tac-Toe AI

best move, finding for 259
board, evaluating for 257
minimax algorithm
board, printing 256
board, representing 256
building, with minimax algorithm 255
empty slots, checking 257
main game loop, implementing 260 – 263
minimax algorithm, implementing 258
winner, checking 256

Titanic dataset
EDA, performing on 36 – 39

text classification **80, 81**

text file
handling 21, 22

text preprocessing **81**
lemmatization 82
stemming 81
tokenization 81
vectorization 82, 83

tokenization **81**

tokens **81**

train-test split
creating 53, 54

trained model
predictions, making 46

transfer learning **83**

U

unsupervised learning model **43**

user-book matrix
building 248

V

variables **8**

vectorization **82, 83**

visualization **30**

W

weather forecasting model
actual and predicted temperatures, visualizing 171, 172
building, with historical data 169
data, splitting into training and test sets 170
features and target variables, selecting 170
historical weather data, preparing 169
libraries, importing 169
linear regression model, training 170
performance, evaluating 171
predictions, making with trained model 171
temperature, predicting for new data 172, 173

web interface
creating, for model 95
Flask web form, creating for input 98, 99
prediction function, building 96, 97
prediction route, creating in Flask 99 – 101

web service
testing 105, 106

weight updates **200**

while loop statements **10, 11**

write mode (w) **22**

X

XGBoost classifier
training 152
using, for employee attrition prediction model 149 – 151

XOR dataset
training with 202, 203