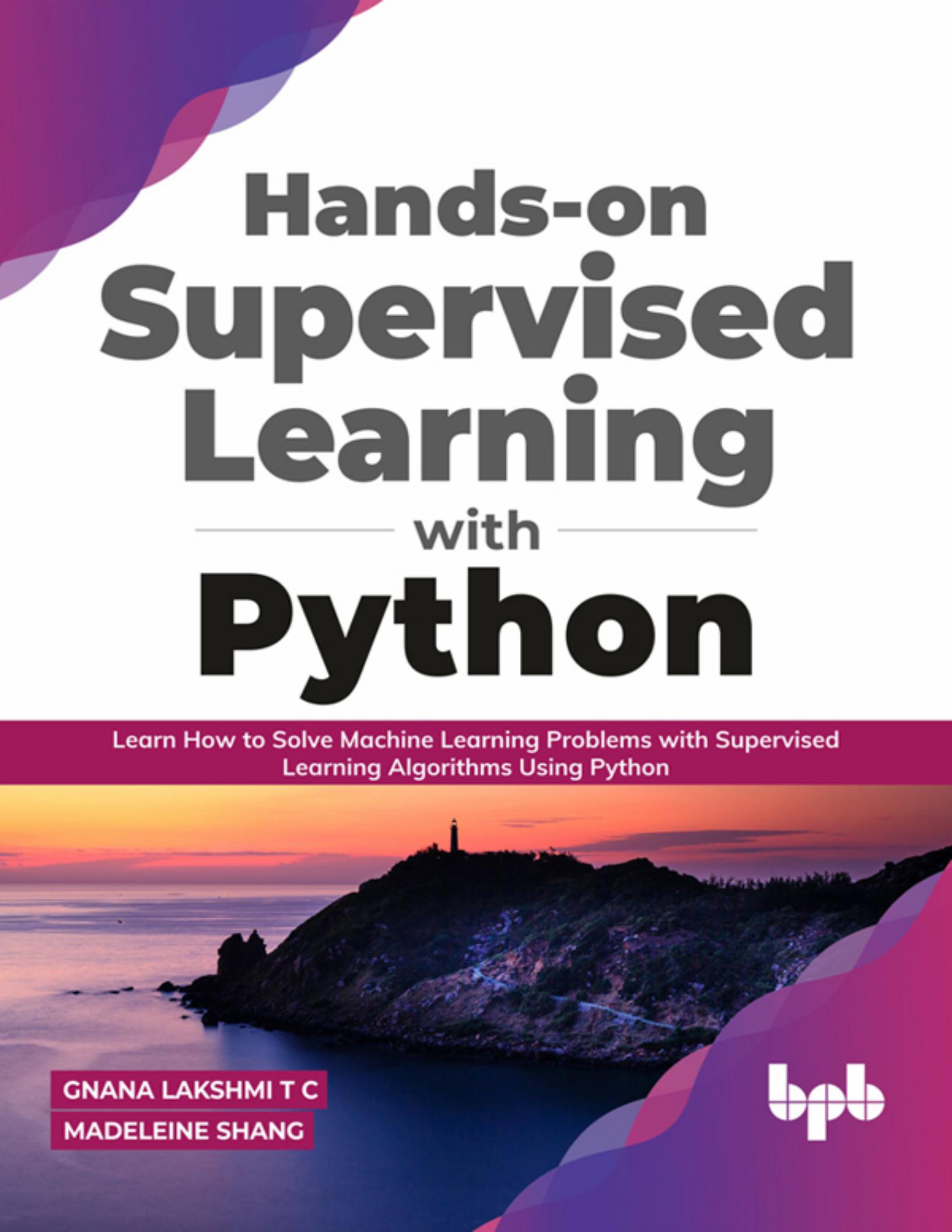


# **Hands-on Supervised Learning with Python**

Learn How to Solve Machine Learning Problems with Supervised Learning Algorithms Using Python

The background of the book cover features a scenic sunset over a coastal cliff. A lighthouse is visible on top of the cliff, silhouetted against the orange and yellow sky. The ocean is calm in the foreground, and the overall atmosphere is peaceful.

**GNANA LAKSHMI T C  
MADELEINE SHANG**





# **Hands-on Supervised Learning with Python**

---

Learn How to Solve Machine Learning Problems with Supervised  
Learning Algorithms Using Python



**GNANA LAKSHMI T C  
MADELEINE SHANG**



# **Hands-on Supervised Learning With Python**

---

*Learn How to Solve Machine Learning  
Problems with Supervised Learning  
Algorithms Using Python*

---

**Gnana Lakshmi T C  
Madeleine Shang**



[www.bpbonline.com](http://www.bpbonline.com)

**FIRST EDITION 2021**

**Copyright © BPB Publications, India**

**ISBN: 978-93-89328-97-4**

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

#### **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

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

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

#### **Distributors:**

##### **BPB PUBLICATIONS**

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

##### **MICRO MEDIA**

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

##### **DECCAN AGENCIES**

4-3-329, Bank Street,

Hyderabad-500195

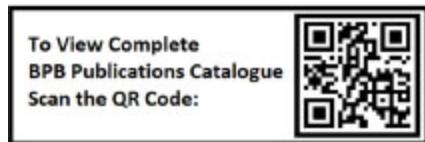
Ph: 24756967/24756400

##### **BPB BOOK CENTRE**

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747



Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002  
and Printed by him at Repro India Ltd, Mumbai

[www.bpbonline.com](http://www.bpbonline.com)

# Dedicated to

*My resilient, strong and loving mother*  
**Mrs. Uma Chandrasekara Barathi**

*And my patient, supportive and loving father*  
**Mr. T.A. Chandrasekara Barathi**

# About the Authors

**Gnana Lakshmi T C** (Gyan) is an emerging technology enthusiast and has given several technical talks across the globe, both in the space of Artificial Intelligence as well as Blockchain. She has taught several courses in the Emerging technology space and is actively looking to explore further at the intersection of Artificial Intelligence, Blockchain and Quantum Computing.

She is a passionate writer and has written several medium articles on various technologies. She co-chaired the Data Science and Artificial Intelligence track for Grace Hoppers Celebration India 2019 conducted by AnitaB.org, which saw an attendance of 6000+ women in technology. She continues her passion for teaching by conducting webinars over the weekend for various communities and is also the core member of Women in AI – Lean In Bangalore community.

She was responsible for creating and leading the Blockchain community for Women Who Code (WWCode) across the globe. With over 8+ years of experience as a Software Developer in the corporate industry, Gyan has been extremely passionate about women in technology and has been a part of the Bangalore network of WWCode Women Who Code for 5+ years. She has conducted several meetups and has given several tech talks on various technology topics like Blockchain and Machine Learning. She is also a hands-on coder and loves exploring new technologies.

**Madeleine Shang** is a Recommender Systems Team Lead @OpenMined. She started the Data Science and Machine Learning community at WomenWhoCode which is now successfully running with 2147 members. She is an expert in AI and Blockchain Research. She has been involved in many startups as a Founder. She is an Adventurer and Futurist at heart.

# About the Reviewer

**Khushi** has a total +6 years of experience in the field of web, automation, and AI technologies. A Machine Learning enthusiast and a strong believer in life long learning. Currently working in IBM as Security Specialist. Few areas of expertise are Web Development, Automation, Python, Chatbots, and NLP.

**Jamuna Vignesh** is a research scholar with 5+ years of research experience in interpreting and analyzing technical reports, research papers, data in order to understand the technology standpoint and its implementation to successfully arrive at conclusions. Proficient knowledge in statistics and analytics. Excellent understanding of research problems and analytics tools for effective analyses of data and interpretation.

**Kavitha Yogaraj** is a technical lead in IBM with key skills in learning new technologies while being delivery focused. She has spent a major part of carriers in the startup company working closely with stakeholders. She has built the data crawling infrastructure, designing end-to-end architecture with implementation. She has also delivered machine learning projects for a new classification and has built a data pipeline making it operationally work. She has recently developed an interest in quantum Qiskit libraries for the role of quantum solutions engineer at IBM. Scikit libraries for machine learning, java and microservices, event-driven systems Kafka and Dockers Open Shift platform.

# Acknowledgement

No task is a single man's effort. Cooperation and coordination of various people at different levels go into the successful implementation of this book.

There is always a sense of gratitude that everyone expresses to others for the help they render during difficult phases of life and to achieve the goal already set. It is impossible to thank individually, so I am, hereby, making a humble effort to thank and acknowledge some of them.

My gratitude to the entire BPB Publications for the opportunity to work with them on this project.

I would also like to thank my family members (special mentions: my sister Sri Lakshmi and my husband Sudheendra R), my supportive friends (special mention: Soundarya and Sharanya), and my readers for providing all the encouragement and motivation.

Finally, I want to thank everyone who has directly or indirectly contributed to complete this authentic piece of work.

It is said, "**To err is human, to forgive is divine.**" In this light, I wish that the shortcomings of the book will be forgiven. At the same time, I am open to all constructive criticisms, feedback, corrections, and suggestions for further improvement. All intelligent suggestions are welcome, and I will try my best to incorporate all the valuable suggestions in the subsequent editions of this book.

# Preface

Almost all of us have interacted with chatbots on one website or the other—while booking tickets, reserving a hotel, or interacting with customer service at the first level etc. I am sure that some of you reading this book might have even heard of/own an Alexa or a Google Home. Sometimes, you might wonder, how does this work? How are e-commerce websites like Amazon, Flipkart, or Myntra providing us with the recommendations based on what we browsed earlier on their websites? How are the movie streaming websites like Netflix/Amazon Prime recommending our favorites based on our watching patterns?

The answer to all these questions is very simple: The Machines are being trained to understand human behavior to help humans have a seamless experience, both online and offline at the click of a button. The technology behind this is known as Machine Learning, where machines learn from the data patterns and try to predict the future data patterns based on the algorithms we design.

Artificial Intelligence is the broader umbrella that which covers machine learning under it. With advanced technologies like GPT-3 (Generative Pre-training that aims toward improving language understanding which is being introduced by organizations like OpenAI, it is clear that Machine Learning is changing the way the industries and humans work. Machine Learning has optimized supply chains, helped launch various digital products and services, and transformed the overall customer experience. Every industry is turning towards AI-ML-based solutions to improve their offerings and services.

One of the most common types of Machine Learning is ‘Supervised Machine Learning,’ where we deal with labeled datasets. There is a lot of research going on in the unlabeled datasets’ domain as well, but Supervised Machine Learning algorithms are much more popular due to their relative ease of use as well as the accuracy of the results that they produce.

This book teaches the concept of Supervised Machine Learning in detail and gives hands-on experience in developing Machine Learning models.

This book promises to be a very good starting point for complete novice learners and is quite an asset to advanced readers too. The author has written the book so that the beginners will learn the concepts related to the Python Programming and Supervised Machine Learning.

**We have developed eight sections where you can find the following topics:**

**Section 1:** This section is an introduction to Machine Learning and Python. The section is divided into three chapters to cover the above topics.

You will understand the basics of python and machine learning and how python is used in machine learning.

**Section 2:** This section is named as “Supervised and Unsupervised Learning.” In this section, we will discuss the primary difference between supervised learning and unsupervised learning and understand which one of them should be used in what context.

**Section 3:** This section focuses on the basic machine learning algorithms. The section is divided into three chapters, where the first chapter focuses on Linear Regression, explaining the algorithm with associated case studies. The second and third chapters focus on Logistic Regression and Support Vector Machines with supporting case studies and step-by-step python code to solve the case studies.

**Section 4:** This section explains the Decision Trees and Random Forest algorithms. This section is split into 2 chapters, each of which explains the Decision Trees and Random Forest Algorithms with real-time scenarios and case studies supported by python code. The section also covers examples of where these algorithms can be applied in an industry scenario.

**Section 5:** This section titled “Time Series” focuses on the Time series algorithm and explains the detailed modelling and functioning behind time series. This section is also supported by a case study.

**Section 6:** This section introduces the learners to “Neural Networks”. This section is the beginning of Deep-learning, which is becoming very popular these days, for dealing with large datasets. This section is split into three chapters, where we introduce neural networks or ANN in the first chapter, RNN(Recurrent Neural Networks) in the second chapter, and Convolutional

Neural Networks(CNN) in the third chapter. Each chapter has its supporting code snippets that learners can try out practically.

**Section 7:** This section titled “Performance Metrics” focuses on some of the most important performance metrics for evaluating machine learning models.

**Section 8:** In this section, titled “Design Thinking for Machine Learning,” we help to shape the learners’ minds toward designing and building solutions for ML problems. This section is split into two chapters, namely, “Introduction to Design Thinking” and “Case Study for Machine Learning.” Design thinking chapter helps the learners to understand how to design an ML solution, and the Case Study chapter walks them through an end-to-end case study in Machine Learning.

# Downloading the code bundle and coloured images:

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

**<https://rebrand.ly/bf1f0>**

## Errata

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

**[errata@bpbonline.com](mailto:errata@bpbonline.com)**

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

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at **[www.bpbonline.com](http://www.bpbonline.com)** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **[business@bpbonline.com](mailto:business@bpbonline.com)** for more details.

At **[www.bpbonline.com](http://www.bpbonline.com)**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.



## **BPB is searching for authors like you**

If you're interested in becoming an author for BPB, please visit [www.bpbonline.com](http://www.bpbonline.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.

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Hands-on-Supervised-Learning-With-Python>. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

## **PIRACY**

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

## **If you are interested in becoming an author**

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit [www.bpbonline.com](http://www.bpbonline.com).

## **REVIEWS**

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

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).

# Table of Contents

## 1. Introduction to Python Programming

Introduction

Structure

Objective

The origin story

Python 2 versus Python 3

Python 3 installation

PIP – The package manager for Python

Installing Python packages

Ways to run a Python program

Using the Python interpreter

Using a Text Editor

Basics of Python programming

Properties

Data types

None

Converting from one type to another

List

Accessing the elements of a list

Tuple

Sets

Dictionary

Strings

String manipulation functions

Concatenate

Replace

Slice

Split

Join

Flow control statements

If...Else statement

Looping statements

[for loop syntax](#)  
[while loop syntax](#)

[Functions](#)

[Advanced Python programming](#)

[Libraries](#)

[Classes](#)

[Exception handling](#)

[Example Python programs](#)

[Program 1 – Write a Python script to print all the prime numbers that are given an interval](#)

[Program 2 – Print the Fibonacci series up to the given n-th term](#)

[Program 3 – Given a string as input, find out if it's a palindrome or not](#)

[A quick python refresher quiz](#)

## **2. Python for Machine Learning (ML)**

[Introduction](#)

[Structure](#)

[Objective](#)

[Why Python?](#)

[Python libraries for ML](#)

[Pandas](#)

[Pandas DataFrames](#)

[Introduction](#)

[Indexing DataFrames](#)

[Pandas loc and iloc](#)

[Numpy](#)

[Numpy array syntax and parameters](#)

[Numpy ndarray creation](#)

[Subsetting numpy arrays](#)

[SciPy](#)

[Functions of SciPy](#)

[Linear algebra with SciPy](#)

[Matplotlib](#)

[Python matplotlib – Different plots](#)

[Bar graph](#)

[Histogram](#)

Scatter plot

Pie chart

Scikit-learn

Scikit-learn – Datasets

Scikit-learn – Principal component analysis

Scikit-learn – Pre-processing

Seaborn

Seaborn – Different plots

Seaborn: Scatter Plots

Tensorflow and Keras

PyTorch

Conclusion

A quick refresher quiz

### 3. Introduction to Machine Learning (ML).

Introduction

Structure

Objectives

The origin story

The big picture

A brief glimpse into machine learning data

Types of data

A glimpse into the dataset

Variables in a dataset

Quantitative variables

Categorical variables

Branches of machine learning

Regression/Classification

Classification

Examples for classification

Regression

Examples for regression

Neural networks

Examples for neural networks

Reinforcement learning

Example of reinforcement learning

Natural language processing

Examples of natural language processing

Applications of machine learning

Exploratory data analysis

Why feature selection/generation?

Overfitting and underfitting

Overfitting

Underfitting

Desired output

Variance and Bias

Variance

Bias

Predictive modeling

Steps that take place in predictive modeling

Step 1 – Collecting data

Step 2 – Splitting the data into training and test data

Step 3 – Creating a machine learning model

Key elements of machine learning

Representation

Evaluation

Optimization

Machine learning in practice

Travel website case study

Data

Exploratory data analysis

Feature engineering

Building the machine learning model

Conclusion

A quick research quiz

## 4. Supervised and Unsupervised Learning in Python

Introduction

Structure

Objectives

Difference between supervised learning and unsupervised learning

Supervised learning

Unsupervised learning

The role of Python in machine learning

## Deep-dive into supervised Learning with examples

Classification

Regression

### Supervised learning example 1– Classification of fruits

Problem statement

Data required

Features

Supervised learning

Accuracy and next steps

### Supervised learning example 2 – Predicting quarterly attrition rate of women in an organization

Problem statement

Data required

Features

Supervised learning

Accuracy and next steps

## Deep-dive into unsupervised Learning with examples

Clustering

Autoencoders

Features separation techniques

Expectation maximization algorithms

### Unsupervised learning example 1 – Dividing pictures of people based on their facial pattern

Problem statement

Data required

Features

Unsupervised learning

Accuracy and next steps

### Unsupervised learning example 2 – Creating groups for your customer based on customer usage patterns

Problem statement

Data required

Features

Unsupervised learning

How it works with K-Means

Accuracy and next steps

## Conclusion

## A quick quiz

### **5. Linear Regression -A Hands-on Guide**

Introduction

Structure

Objectives

What is the linear regression?

Statistics in regression analysis

*Residual sum of squares*

*R<sup>2</sup> score/ R2 score*

*Root Mean Squared Error (RMSE)*

Simple linear regression

Case study – I

*About the data*

*Python code and step-by-step regression analysis*

*Conclusion*

Case study – II

*About the data*

*Python code and step-by-step regression analysis*

*Conclusion*

Multiple linear regression

Case study – I

*About the data*

*Python code and step-by-step regression analysis*

*Conclusion*

Case study – II

*About the data*

*Python code and step-by-step regression analysis*

*Conclusion*

Quiz – What did you learn about linear regression

### **6. Logistic Regression – An Introduction**

Introduction

Structure

Objectives

Logistic regression with an example

Inner workings of logistic regression

[Logistic regression equation](#)

[Gradient descent](#)

[How does gradient descent work?](#)

[Step 1](#)

[Step 2](#)

[Step 3](#)

[Case study - I](#)

[About the data](#)

[Attribute information](#)

[Python code and step-by-step regression analysis](#)

[Summary](#)

[Case Study - II](#)

[About the data](#)

[Attribute information](#)

[Python code and step-by-step regression analysis](#)

[Summary](#)

[Practical examples of logistic regression](#)

[Conclusion](#)

[Quiz](#)

## [7. A Sneak Peek into the Working of Support Vector Machines](#)

[Introduction](#)

[Structure](#)

[Objectives](#)

[Why do we need an optimal decision boundary?](#)

[Workings of the SVM](#)

[Applications of support vector machines](#)

[Face detection](#)

[Text categorization/classification](#)

[Image classification](#)

[Bioinformatics](#)

[Maximal margin classifier](#)

[Soft margin classifier](#)

[Kernels](#)

[Introduction to SVM kernels](#)

[Linear kernels](#)

[Polynomial kernels](#)

[Radial kernels](#)  
[Case study - I](#)  
    [About the data](#)  
[Python code and step-by-step regression analysis](#)  
    [Case study conclusion](#)  
[Case study - II](#)  
    [About the data](#)  
    [Attribute information](#)  
[Python code and step-by-step regression analysis](#)  
    [Case study conclusion](#)  
[Conclusion](#)  
[Quiz](#)

## **8. Decision Trees**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[What are decision trees?](#)  
[CART algorithm \(classification and regression trees\).](#)  
    [Advantages of CART](#)  
[Decision tree construction](#)  
[Truncation and pruning](#)  
[Calculating information gain](#)  
    [What is Entropy?](#)  
    [What is the Gini index?](#)  
    [What is the information gain?](#)  
[Case Study - I](#)  
    [About the data](#)  
    [Python code and step-by-step regression analysis](#)  
[Summary](#)  
[Conclusion](#)  
[Quiz - Chapter review](#)

## **9. Random Forests**

[Introduction](#)  
[Structure](#)  
[Objectives](#)

What are random forests?

Here is how we do it

Hyperparameters used in random forest algorithm

The bagging technique

Why are random forests better than regular decision trees?

Random forest training

Industry use case of random forests

Case study – I

About the data

Attribute information

Python code and step-by-step regression analysis

Case study conclusion

Summary

Quiz

## 10. Time Series Models in Machine Learning

Introduction

Structure

Objectives

Analysis methods in time series

Seasonal/periodic and cyclic patterns in time series

Seasonal patterns

Cyclic patterns

Moving average model

Walking through Table 10.1

ARIMA

How does it work?

Forecasting

Time series case study

About the data

Step by step Python Analysis for an ARIMA based time series forecasting model

Summary

Conclusion

Quiz

## 11. Demystifying Neural Networks

Introduction

Structure

Objectives

What are neural networks

Why neural networks/deep learning?

Working of the neural network

The mathematics associated with neural networks

*Types of activation functions in neural networks*

*Linear activation functions*

*Non-linear activation functions*

*Sigmoid/Logistic activation function*

*Tanh or Hyperbolic*

*ReLU (Rectified Linear Unit)*

*What is the vanishing gradient problem, and how does ReLU solve this problem?*

Forward propagation and Back propagation

*Forward propagation*

*Backward propagation*

Shallow and deep neural networks

A quick look at TensorFlow

Case study - I

*About the data*

*Attribute information*

*Python code and step-by-step analysis*

*Alternative method – Programming from the scratch*

*Case study summary*

Conclusion

Quiz

## **12. Recurrent Neural Networks (RNN)**

Introduction

Structure

Objectives

What are feed-forward networks?

What are Recurrent Neural Networks (RNNs)?

Applications of Recurrent Neural Network

*Application 1 – Image Captioning*

[Application 2 – Time series prediction](#)

[Application 3 – Text mining and sentiment analysis](#)

[Types of Recurrent Neural Network](#)

[One to one](#)

[Many to one](#)

[One to many](#)

[Many to many](#)

[Python code and step-by-step RNN using Keras](#)

[Application of RNN in real-time example](#)

[Conclusion](#)

[Quiz](#)

## [13. Convolutional Neural Networks](#)

[Introduction](#)

[Structure](#)

[Objective](#)

[The first killer app of deep learning](#)

[How do we represent images?](#)

[Convolutions](#)

[Application of filters across channels](#)

[Pooling](#)

[What is max pooling?](#)

[What is average pooling?](#)

[Step-by-step code walkthrough](#)

[Advanced architecture and techniques](#)

[Data augmentation](#)

[Practical notes](#)

[Transfer learning](#)

[Examples of advanced CNN task and architecture](#)

[DCGANs \(Deep Convolutional Generative Adversarial Networks\)](#)

[Object localization](#)

[Conclusion](#)

[Quiz](#)

[Image reference links](#)

## [14. Performance Metrics](#)

[Introduction](#)

Structure

Objective

What makes a model useful?

Classification

Formalizing concepts

What metrics should you use (rule of thumb guide)?

Classification problems

Regression

Most commonly used error metrics for regression

But now we notice a problem

Overfitting, underfitting, bias, and variance

Formalizing concepts

High bias (underfit)

Variance (overfit)

How to make your model more useful?

Efficiency

Example scenarios of model efficiency

Other practical considerations

Data needed

Parametric models versus non-parametric models

Transparency and interpretability

Robustness

Flexibility

Conclusion

Reference links:

## 15. Design Thinking for ML

Introduction

Structure

Objectives

Answering some important questions in the field of product

development

What is agile?

How well do estimates work in ML?

What is a design sprint?

Why is there a chapter on design for ML?

Introduction to design thinking

Why does design thinking matter?

Design thinking in select details

Questions during the process of design thinking

A quick peek into the challenges in current ML tools and technologies

Steps to follow in the design thinking process

Step 1 – Define

Step 2 – Ideate

Step 3 – Prototype

Step 4 – Test

Practical advice for ML prototyping

Conclusion

Quiz

References

Resources

## 16. Case Study for Machine Learning

Introduction

Structure

Objective

Machine Learning Case Study

Overview

About the data

Python code and step-by-step regression analysis

Case study summary

Conclusion

Citation used for a dataset

# CHAPTER 1

## Introduction to Python Programming

### Introduction

Python is one of the fastest-growing programming languages in today's world. It started as a scripting language and now continues to dominate the world of machine learning and data science. From automation to data analysis, Python is the preferred programming language for all the developers due to its use of ease and due to the abundant libraries that make up this language.

### Structure

- The origin story
- Python2 versus Python 3
- Python 3 installation
- Basics of Python programming
- Data types
- Flow control statements
- Functions
- Advanced Python programming
- Classes
- Exception handling
- Python sample programs

### Objective

This chapter will introduce the basics of Python programming to the readers. You will learn to install Python and its associated packages on your local system and run basic Python programs on your system. You will learn

about the different data types and functionalities that make Python a unique programming language.

## The origin story

Python 1.0 was released in 2004 in January. Initially, Python was designed to target the mobile operating system, and one of its main attractions was that it provided exception handling. One of the exciting features of Python 1.0 was its essential support for functional programming. In the beginning times, around the release of Python 1.2, it was widely used to promote computer programming, as it is a high-level language and more comfortable to learn as compared with other computer programming languages.

Python has been an open-source programming language from the very beginning, encouraging contributions from across the globe to diversify the features of the language itself. The two significant enhancements while moving from Version 1.0 to 2.0 in the year 2000 were as follows: provisioning a garbage collector and support for Unicode. Python version 3.0 is a relatively new version that was released in 2008. Currently, both versions 2.x and 3.x are being maintained simultaneously and can be used by the programmers.

**INFO: Developed by a Dutch programmer, Guido Van Rossum at the National Institute of Mathematics and Computer Science, the Python language is named after a TV show called “Monty Python’s Flying Circus” and not after the python snake.**

## Python 2 versus Python 3

Developed in 2000, Python 2.x became a universal language very soon. Due to its simplicity and ease of adoption, Python 2.x garnered a lot of community support and continues to still be prevalent in the community. There are several libraries from Python 3 that have been backported for Python 2 because of its continued usage in several companies. Although Python 3 has been released in the year 2008, many organizations still prefer to use Python 2 versions.

Python 2.7 is one of the most famous python versions that is recommended in the industry, but for all practical purposes, Python 3 will be the future of

the Python programming language. If you have already learned Python 2, it will not be that difficult to switch to Python 3, as the changes are minimal and easy to adopt. The core features of the language remain the same.

Python 3.7.3 is the latest version of Python and supports a `python2to3` utility that allows you to convert your Python 2.x code to Python 3.x seamlessly. It reads your Python 2.x code and applies a series of fixers to transform it into a valid 3.x code. We will be using Python 3 throughout this book, and mentions of Python everywhere will refer to Python 3.

**TIP:** If you are new to Python programming and are learning Python from scratch, it is highly recommended that you learn Python 3. Python 3.x offers several exciting and improvised features like “standard printing format,” “string is Unicode by default,” “division of integers returning float answers,” “addition of `as` keyword to import statements,” and many more.

## Python 3 installation

Python 3, for all the operating systems, can be downloaded and installed from the official [python.org](https://www.python.org) website: <https://www.python.org/downloads/release/python-373/>.

To check if you already have python installed on your system, you can use the following command on a terminal in a Linux/Mac operating system and a CMD window on a Windows operating system:

```
python --version
```

The preceding command will show you the version of Python installed on your system if it exists.

If your system already shows a Python installed (most systems come with Python 2.7 installed by default), you can install the latest Python 3 version on your system. Once the installation is complete, you can type the following command on the terminal to verify that Python 3 has been installed correctly:

```
python3 --version
```

The preceding command should display the version of Python 3 that you installed using the above given download link. You can keep both the versions of Python on your system if you wish to do so, as the commands for invoking scripts/interpreter using both the versions is different.

## **PIP – The package manager for Python**

PIP, a.k.a `pip`, is the package manager for Python, which is used to install all the Python libraries on your system. For all installed Python versions above 3.4, `pip` comes by default with Python.

PIP is a powerful feature that helps you install packages for Python, which in turn contains the python modules that you will need for your program/business logic. You can check for the `pip` version on your system by running the following command on your terminal:

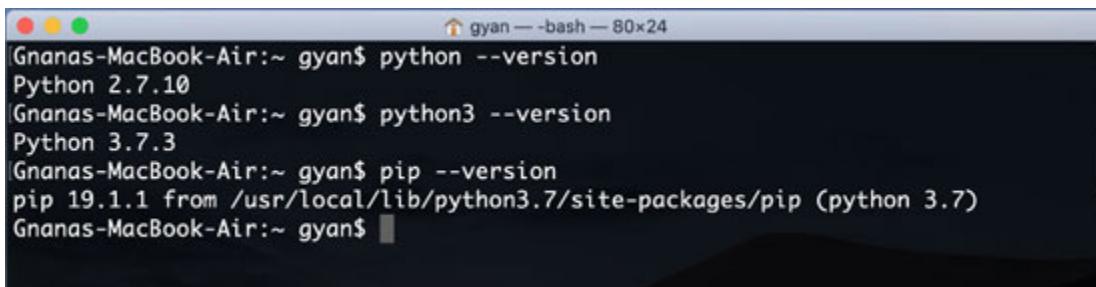
```
pip --version
```

The preceding command should give the `pip` version along with the Python version that it's installed with. In case you have two pips on your system, to check with you have `pip` associated with your Python 3 installation, you can also(optionally) type:

```
pip3 --version
```

The preceding command will show you the `pip` version for your Python 3 installation. In most of the cases, both the preceding commands may yield the same output if you have only one `pip` installed on your system.

If you do not have `pip` installed on your system, you can use the following link to download and install `pip` on your system:  
<https://pypi.org/project/pip/>:



```
gnan@Gnanas-MacBook-Air:~ gyan$ python --version
Python 2.7.10
gnan@Gnanas-MacBook-Air:~ gyan$ python3 --version
Python 3.7.3
gnan@Gnanas-MacBook-Air:~ gyan$ pip --version
pip 19.1.1 from /usr/local/lib/python3.7/site-packages/pip (python 3.7)
gnan@Gnanas-MacBook-Air:~ gyan$
```

**Figure 1.1: Verifying Python and pip installations**

## Installing Python packages

Python is an open-source language, and, therefore, most of its functionalities come in the form of modules, wherein each module acts as a Python code library that you can include in your code. These modules are bundles inside packages that need to be installed on your system to use the functionality provided by that module.

There are a few basic modules that come pre-installed with Python, but the rest of these need to be installed on your system. We will use the Python package manager, pip, to install the necessary packages. To install a package, you can type the following command on your terminal:

```
pip install <package_name>
```

Or the following command:

```
pip3 install <package_name>
```

Alternatively, to remove a package, you can use the following command:

```
pip uninstall <package_name>
```

Or the following command:

```
pip3 uninstall <package_name>
```

You can also have a look at the help menu provided by `pip install` to use some advanced options. The command to get the help menu is:

```
pip install --help
```

PIP is a beneficial tool that makes the management of the entire Python ecosystem easy and seamless. Python supports our applications by providing a plethora of in-built libraries and packages that can be installed in one click using the package manager.

**INFO: In case the python --version command does not reveal the python version installed, it could mean that the python path is not set in your system. You need to find the appropriate commands to set the python path for your operating system if it is not set by default.**

## Ways to run a Python program

The following are a few ways to run a Python program on your local system:

### Using the Python interpreter

Python is an interpreted language; that is, it is interpreted line by line without previously compiling it into a machine-readable language. The instructions are executed directly, and the execution doesn't go further if there's an error in any of the lines of the code.

The Python interpreter comes installed with the Python installation on your system. The interpreter allows you to run Python commands on your terminal and see the output immediately. Python interpreter is a great way to look at some python commands quickly on your screen and understand their functionality (for example, addition, modulus, subtraction, binary operators, and so on).

Typing the following command on your terminal will open the Python interpreter along with the version of Python that the interpreter supports:

**python**

Or the following command:

**python3**

The preceding command will open an interpreter that displays `>>>` for you to start writing your commands. If you want to exit the interpreter, you can use *Ctrl + D* or type `exit()` in the interpreter.

Here are some commands that you could try on the interpreter and see the output on your terminal:

```
>>> 2+3
>>> a = 8
>>> b = 9
>>> a+b
>>> a*b
>>> a\b
>>> a/b
```

```
>>> b = "Hello World"
>>> b
>>> c = b+" "+"From Interpreter"
>>> c
```

```
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.37.14)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> a = 8
>>> b = 9
>>> a+b
17
>>> a*b
72
>>> a/b
9.0
>>> a/b
0
>>> a^b
512
>>> b = "Hello World"
>>> b
'Hello World'
>>> c = b+" "+"From Interpreter"
>>> c
'Hello World From Interpreter'
>>> exit()
Gnanas-MacBook-Air:~ gyan$
```

*Figure 1.2: Interpreter o/p on the Terminal*

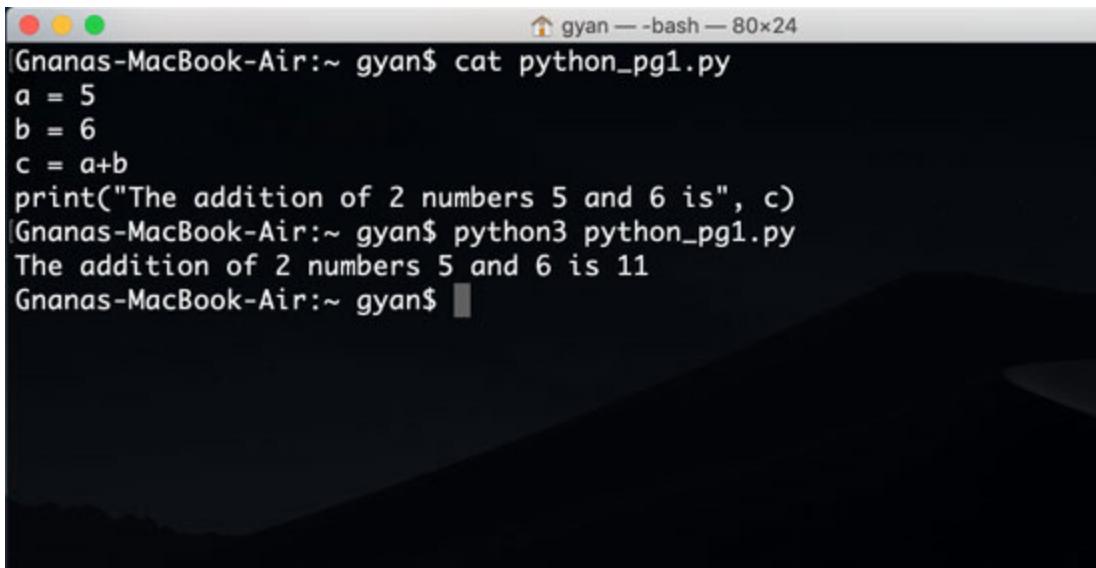
## Using a Text Editor

Python programs that contain more than five to six lines of code, are generally written in a text editor or in a vi editor that can be run on the Terminal. The Python program, also known as a Python script, can be run on the terminal. As Python is an interpreted language, the execution will stop and an error will be thrown with the line number and the related error information on the Terminal if the interpreter encounters an error at a particular line.

You can type the Python program in the text editor and then use the following command to run the Python script/program. The program needs to be saved with a .py extension.

The command to run a Python program is as follows:

```
python/python3 <pg_name> <parameters_if_required>
```



The screenshot shows a terminal window titled 'gyan — bash — 80x24' on a Mac OS X desktop. The window contains the following text:

```
Gnanas-MacBook-Air:~ gyan$ cat python_pg1.py
a = 5
b = 6
c = a+b
print("The addition of 2 numbers 5 and 6 is", c)
Gnanas-MacBook-Air:~ gyan$ python3 python_pg1.py
The addition of 2 numbers 5 and 6 is 11
Gnanas-MacBook-Air:~ gyan$
```

*Figure 1.3: Running a Python script on the Terminal*

## Basics of Python programming

In this section, we will discuss the basic properties, functionalities, and the behavior of Python programming. We will take a more in-depth look at the different data types and constructs supported by Python.

## Properties

Python programming language is inspired by several languages like C, C++, Unix shell, and so on, and the keywords used in the language are mostly English words. Python is dynamically typed and requires you to adhere to the syntax strictly.

The variables in Python are case sensitive; therefore, `obj` and `OBJ` are two different variables. Python also has help integrated at every step to understand more about a particular object or library. You can use the following command to understand more about an object:

```
help(obj)
```

The preceding command can also be typed in the interpreter to get help on specific libraries, objects, and many more.

If you would like to explore the directory structure of an object, you can type `dir(obj)` to get the structure of it.

## Data types

In Python programming, you do not have to mention the data type with each variable that you define, as it is dynamically typed. Python programming is heavily based on the concept of using objects, and, therefore, the data types are modeled as classes, and the variables of each data type are like the class instances or objects.

Python has several data types; here is a look into each data type and its example:

### None

The `None` variable is similar to the concept of having a null variable in other languages. If a variable is not assigned any values, it automatically assumes the value of `None`.

For example:

```
var_none = None

if var_none is None:
    print('The variable is type None')
else
    print('The variable is not type None')
```

### Numeric

Under the `Numeric` type, we have multiple subtypes like `int`, `float`, `bool`, and `complex` classes:

- **Integers/int:** It can be of any length and is only limited by the memory available for your program.
- **Floating-point numbers/float:** It indicates real numbers and can be accurate up to 15 decimal places. They always contain a decimal point that divides the integer and the fractional part. Floats can also be written using the scientific notation with `e` (For example, `1.2e4 = 1.2*10^4`).

- **Complex numbers:** They are of the format  $a + bj$ , where  $a$  is the real part and  $b$  is the complicated part. Here,  $a$  and  $b$  can be float values. The complex numbers are not commonly used in python language or programming.
- **Boolean/bool:** It implements true or false. It represents 1 for `true` and 0 for `false`.

```
Gnanas-MacBook-Air:~ gyan$ python3
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> b = 6.5
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> c = int(b)
>>> c
6
>>> type(c)
<class 'int'>
>>> d = complex(a,b)
>>> d
(5+6.5j)
>>> e = True
>>> type(e)
<class 'bool'>
>>>
```

*Figure 1.4: Numeric data types in Python*

## Converting from one type to another

We can easily convert between numeric variables using their data type keywords. Some of the example conversion statements are:

```
#Convert from float to int
a = 5.6
b = int(a) #b becomes 5

#Convert from int to float
a = 8
b = float(a) #b becomes 8.0
```

**INFO:** You can use the keyword `type()` to find out the type of the variable.

## List

Lists are comma-separated values, that are initialized using a square bracket. It generally contains an ordered sequence of items. All the items of the list need not be of the same type, and the lists are mutable.

An example code snippet for list declaration is as follows::

```
a = [1,2,3,4]
b = [1, 'a', 2, 'b']
c = ['Hello', 'World', 2]
d = [3, 2.2, 'Python']
```

## Accessing the elements of a list

Lists are similar to arrays in other programming languages, and the elements of a list can be accessed using the square brackets ("[]"). Python lists start with an index 0.

You can access the individual elements of a list or a group of elements using the following various syntaxes:

```
#Access individual elements
a = [1, 2, 3, 4]

#This will output 2
a[1]

#Access the first 2 elements, o/p - [1,2]
a[:2]

#Access the last two elements, o/p - [3,4]
a[2:]

#Access elements starting from index 1 to index 2, o/p - [2,3]
a[1:3]

a[1] = 5
#Mutable lists, output - [1,5,3,4]
```

a

```
[>>> a = [1, 2, 3, 4, 5]
[>>> a[2]
3
[>>> a[:3]
[1, 2, 3]
[>>> a[2:]
[3, 4, 5]
[>>> a[0]
1
[>>> a[4]
5
[>>> a[1:3]
[2, 3]
```

*Figure 1.5: Python lists*

## Tuple

Python tuples are similar to Python lists except for the fact that they are not mutable. Tuples can be used to store information like cryptographic keys, as you do not want anyone to change the data in the tuple.

A tuple can be declared using the round brackets (“()”). The elements of a tuple can also be of different data types.

An example code snippet for declaring a tuple is as follows::

```
a = (1, 2, 3)
#Output - (1,2,3)
a
#Output - 2
```

```
a[1]

#Generates error
a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## Sets

A Python set is similar to a list or a tuple but with some constraints as follows:

- The set cannot have duplicate elements.
- The individual elements in the set are not mutable, but the whole set is mutable at once (the whole set can be reassigned).
- There is no way to access an individual element; you can only loop through the set to access the elements.

Example code snippet for a set:

```
fruits1 = set(["Mango", "Apple", "Banana"])
fruits2 = set(["Grapes", "Mango"])

#Union of two sets
fruits3 = fruits1|fruits2

#print fruits3 - Output will be set(["Mango", "Apple",
#"Banana", "Grapes"]) - Final set has only one Mango
fruits3

#Adding items to a set
fruits1.add("Jackfruit")

#Removing items from a set
fruits1.discard("Mango")
```

## Dictionary

A Python Dictionary is a set of **key:value** pairs defined such that the keys are unique. We can initialize it using a pair of curly braces ("{}"). The

dictionaries can be used when you want to retrieve information in an optimized way.

Example code snippet for a dictionary:

```
#Declare a dictionary
a = {'fruit1':'mango', 'fruit2':'cherry'}
print(type(a))

#Print dictionary elements - prints mango and cherry
print("a['fruit1'] = ", a['fruit1']);
print("a['fruit2'] = ", a['fruit2']);

#Generates error
print("d['mango'] = ", d['mango']);
```

## Strings

In Python programming, strings are Unicode characters. Strings can be declared using single quotes (' ') or double quotes (" "). Strings are immutable, and the individual characters of a string can be accessed using square brackets ("[]").

An example string declaration and some essential string functions is as follows::

```
a = "Hello World"

#print the value of a | Output: Hello World
a

b = 'Hello World'

#print the value of b | Output: Hello World
b

#print the length of the string | Output: 11
len(a)

#print character from string | Output: H
a[0]

#print the lowercase of the string | Output: hello world
```

```
a.lower()  
  
#Print the uppercase of the string | Output: HELLO WORLD  
a.upper()
```

## String manipulation functions

Strings are quite easy to work within Python, as there are various string manipulation functions present in Python. Some of them are as follows::

### Concatenate

Two or more strings can be concatenated in python using the + operator.  
The string concatenation example is as follows:

```
a = "Hello "  
b = "World"  
c = a+b  
  
#c is now "Hello World"  
print(c)
```

### Replace

If you want to replace a part of your string, you can use the replace function in Python to do so.

The string replace example is as follows:

```
A = "Hey World"  
B = a.replace('y', 'llo')  
  
#Print b | Output: Hello World  
print(b)
```

### Slice

The process of slicing can help you remove some characters from your string and retain the rest of the string. You can use the ":" operator to perform slicing on your string.

The slice example code snippet is as follows::

```
a = "Message to slice"
b = a[4:16]

#print b | Output: age to slice
print(b)
```

## Split

The splitting of a string based on any character is known as a **delimiter**. A string that is split using a delimiter is stored as an array. You can use the `split` keyword to split a string.

The split example code snippet is as follows:

```
a = "Mango Fruit"
b = a.split(' ')

#print b | Output: ('mango', 'fruit')
print(b)
```

## Join

Two or more strings can be joined with a delimiter using the `join` keyword for strings in Python. An array of strings can be joined to create a single string in Python.

The join example code snippet is as follows:

```
a = ['apple', 'and', 'mango']
b = ' '.join(a)

#print b | Output: ("apple and mango")
print(b)
```

## Flow control statements

The flow control statements in Python predominantly refers to conditional statements and looping statements. These statements control the flow of your program based on certain logical conditions.

Here are examples of some flow control keywords and their usage in Python programming:

## If...Else statement

The `if...else` statement, also known as a conditional statement, is used to execute statements based on logical conditions that either return a `true` or `false` value.

The if...else example code snippets is as follows:

```
a = 10
b = 10

if(a == b)
    print(" a is equal to b")
else
    print("a is not equal to b")
```

## Looping statements

The Loops can be written to reduce excessive writing number of lines in your code. For loop and while loop are the two looping statements that we can use for looping in Python programming.

### for loop syntax

```
for var in arr { }
or
for var in range(x):
{ }
```

**Example code snippet – for loop:**

```
#Output: 1,2,3,4
arr = [1,2,3,4]
for x in arr:
    print(x)

#Output: 0,1,2,3,4,5,6,7,8,9
for i in range(10):
```

```
print(i)

#Output: m,a,n,g,o
for c in 'mango':
    print(c)
```

## while loop syntax

```
while var < int:
    print(var)
```

Example code snippet - while loop:

```
#Output: 1,2,3,4,5
i = 1

while i < 6:
    print(i)
    i++
```

**INFO:** range(x) will give us an array that has (0-x-1) values.

## Functions

A function is a critical piece of code in Python programming. A function can execute a bunch of statements; it can take a set of parameters and also be able to return a value. A function can only be executed when it is called from the Python script/program.

A function in Python can be defined using the keyword def.

The function declaration syntax is as follows:

```
def function_name( function_parameters ):
    function_body
```

Here are some example snippets of working with functions in python:

### **Example 1:**

```
#Output: Gnana, Lakshmi, Python
def funcPrint(name):
    print(name)
```

```
funcPrint("Gnana")
funcPrint("Lakshmi")
funcPrint("Python")
```

### Example 2:

```
def add(a,b):
    c = a+b
    return c

d = add(5,6)

#print d | Output: 11
print(d)
```

### Example 3:

```
#Output : 1,2,3,4
def printList(arr):
    for x in arr:
        print(x)

varArr = [1,2,3,4]
printList(varArr)
```

### Example 4:

```
#Output : 6
def loopFunc(arr):
    var = 0
    for i in arr:
        if(i%2 == 0):
            var += i
        else:
            continue
    return var

varArr = [1,2,3,4]
loopFunc(varArr)
```

## Advanced Python programming

In this section, we will explore some advanced python concepts like Python libraries, error handling, and classes.

## Libraries

Python libraries, also known as Python modules, are a group of functions or statements that provide some functionality that might be used frequently across different programs. The modules allow you to use certain functions repeatedly without having to re-write them for every individual program.

The libraries/modules can be used in your program by importing the library at the start of your program using the `import` keyword as follows:

```
import math
```

The preceding statement imports the `math` library and offers functionalities that are available in the `math` library.

You can alternatively import only certain functions from the library to save time by using the `from` keyword as follows:

```
from math import pi
```

Here is an example program of how you can use a library function:

```
from math import pi

#Calculating the area of a circle using pi
r = 5
a = r*r*pi

print('Area of the circle is', a)
```

You can also write your custom libraries that bundle together a group of functions and use them in your code freely by importing the module/library.

## Classes

A group of object-oriented elements bundled together can be written in a class. A class is like a template for creating objects. You can create classes in Python using the keyword `class`.

The syntax for declaring a `class` in Python is as follows:

```
Class className:

    <statement1>
    <statement2>
    .
    .
    .
    <statementN>
```

The example code snippets for classes in Python are as follows:

### Example 1:

```
#Output: apple
class Fruits:
    fruit1 = "mango"
    def printFruit():
        return "apple"

Fruits fruit;
isApple = fruit.printFruit();
print(isApple)
```

### Example 2:

```
class machineLearning:
```

```
"""
    This is the second example
"""

var = 1
def printHelloworld(self):
    print('Hello World')

#Output: 10
print(machineLearning.var)

# Output: <function machineLearning.printHelloworld at
# 0x00000ab453 >
print(machineLearning.func)

# Output: 'This is the second example'
```

```
print(machineLearning.__doc__)
```

### Example 3:

Creating a class object in Python:

```
class machineLearning:  
    """  
        This is the second example  
    """  
  
    var = 1  
  
    def printHelloworld(self):  
        print('Hello World')  
  
ml = machineLearning()  
  
# Output: 1  
print(ml.var)  
  
# Output: 'Hello World'  
ml.printHelloworld()
```

### Example 4:

Defining a constructor in Python:

```
class CNumber:  
    r = 1  
    i = 2  
  
    def __init__(self,a = 0,b = 0):  
        self.r = a  
        self.i = b  
  
    def printData(self):  
        print("{0}+{1}j".format(self.r,self.i))  
  
    # Object creation  
complex1 = CNumber(1,2)  
# Call printData() function  
# Output: 1+2j  
complex1.printData()  
  
# Create a second object
```

```

# Create another class element
complex2 = CNumber(7)
complex2.var = 50

# Output: (7, 0, 50)
print((complex2.r, complex2.i, complex2.var))

# but complex1 object doesn't have the new element
# AttributeError: 'CNumber' object has no attribute 'var'
complex1.var

```

## Exception handling

As Python is an interpreted language, the execution of the code stops at the line where the error is encountered, and an error message is displayed on the terminal/interpreter.

The exception handling needs to be written in the code to handle any exceptions that we might encounter while executing our Python program.

The following three keywords are used during exception handling:

- `try`: The `try` keyword lets you write specific lines of the code and test them for errors as one single block.
- `except`: The `except` keyword lets you handle the error.
- `finally`: The `finally` block lets you execute the code that you would like execute irrespective of the result of the `try` or `except` block.

The example code snippets for exception handling are as follows:

### **Example 1:**

```

# Output: An exception is generated because var is not defined
try:
    print(var)
except:
    print("An exception is generated because var is not defined")

```

### **Example 2:**

```

# Output: Error because var is not defined
# This statement will always be printed
try:

```

```

print(var)
except:
    print("Error because var is not defined")
finally:
    print("This statement will always be printed")

```

## Example Python programs

### Program 1 – Write a Python script to print all the prime numbers that are given an interval

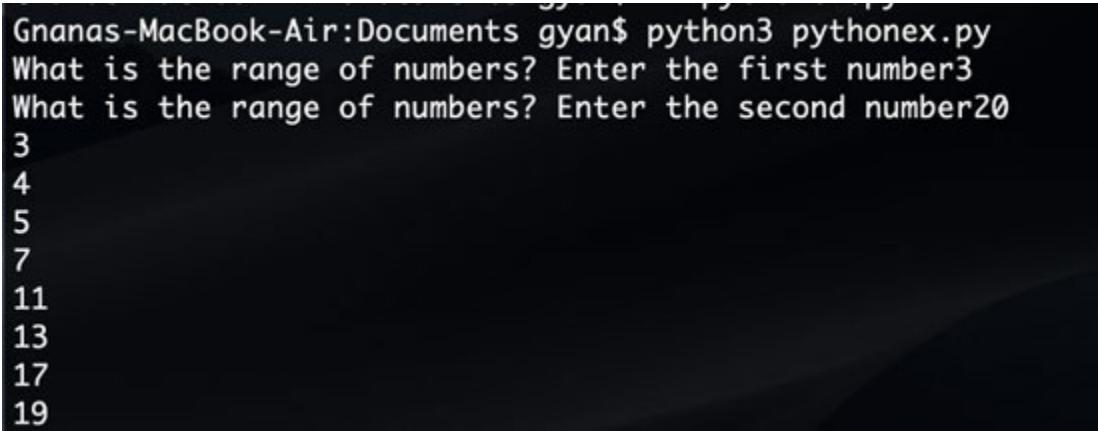
```

input1 = input("What is the range of numbers? Enter the first
number")
input2 = input("What is the range of numbers? Enter the second
number")

input1 = int(input1)
input2 = int(input2)

for i in range(input1,input2):
    if i > 1:
        for j in range(2,i//2):
            if(i%j == 0):
                break
        else:
            print(i)

```



```

Gnanas-MacBook-Air:Documents gyan$ python3 pythonex.py
What is the range of numbers? Enter the first number3
What is the range of numbers? Enter the second number20
3
4
5
7
11
13
17
19

```

*Figure 1.6: Python program to print all the prime numbers in a given interval*

## Program 2 – Print the Fibonacci series up to the given n-th term

```
num_terms = input("Enter the number of terms")
n = int(num_terms)
a = 0 #First Term
b = 1 #Second Term

i = 0

if(n <= 0):
    print("Please enter a valid number of terms")

if(n==1):
    print("Fibonacci series",a)

else:
    print("Fibonacci sequence upto",n,:")
    while i < n:
        print(a,end=' ')
        next = a + b
        #Update values
        a = b
        b = next
        i += 1
print("\n")
```

```
Gnanas-MacBook-Air:PythonBook gyan$ python3 fibbo.py
Enter the number of terms 10
Fibonacci sequence upto 10 :
0 1 1 2 3 5 8 13 21 34
```

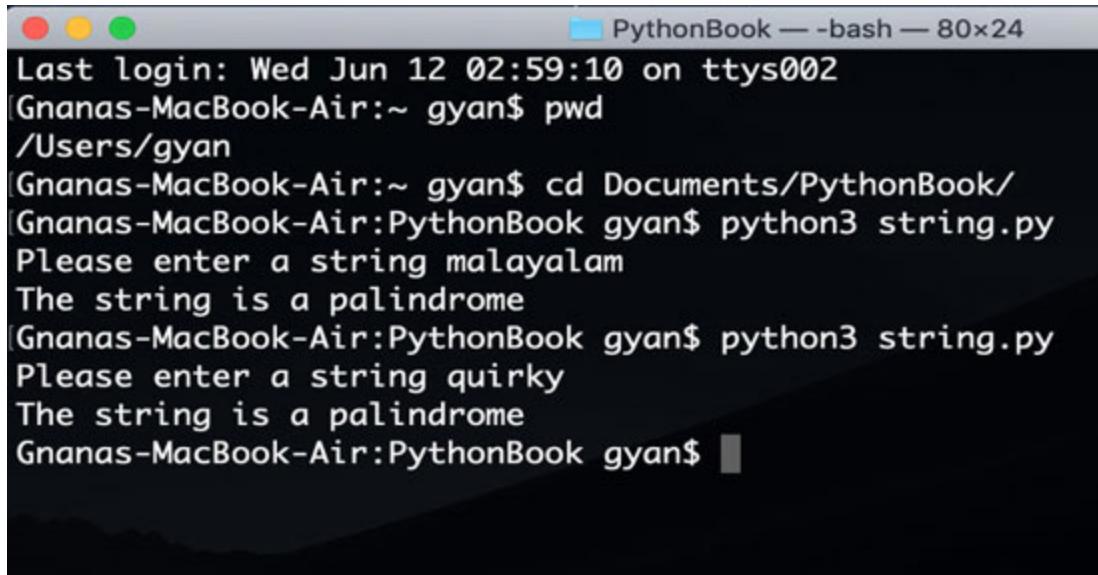
*Figure 1.7: Fibonacci series*

## Program 3 – Given a string as input, find out if it's a palindrome or not

You can use the keyword `input` to get the input from the user. To check for the palindrome, you can reverse the string using the slice (`::`) operator. If

the reversed string and the original string are equal, then the word is a palindrome.

```
str = input("Please enter a string")
if(str == str[::-1]):
    print("The string is a palindrome")
else:
    print("The string is not a palindrome")
```

A screenshot of a terminal window titled "PythonBook — bash — 80x24". The window shows a command-line session. It starts with system login information: "Last login: Wed Jun 12 02:59:10 on ttys002". The user then types "pwd" to show the current directory: "/Users/gyan". They change the directory to "Documents/PythonBook" using "cd". Finally, they run the Python script "string.py" with the command "python3 string.py". The script prompts for a string ("Please enter a string"), receives "malayalam" as input, and prints "The string is a palindrome". The user then runs the script again with the input "quirky", which also results in "The string is a palindrome". The session ends with the user's name "gyan\$".

```
Last login: Wed Jun 12 02:59:10 on ttys002
[Gnanas-MacBook-Air:~ gyan$ pwd
/Users/gyan
[Gnanas-MacBook-Air:~ gyan$ cd Documents/PythonBook/
[Gnanas-MacBook-Air:PythonBook gyan$ python3 string.py
Please enter a string malayalam
The string is a palindrome
[Gnanas-MacBook-Air:PythonBook gyan$ python3 string.py
Please enter a string quirky
The string is a palindrome
Gnanas-MacBook-Air:PythonBook gyan$ ]
```

Figure 1.8: Python program to find out if a string is a palindrome or not

## A quick python refresher quiz

1. What is the difference between Python lists and tuples?
2. What is the string manipulation function for finding a substring in a Python string?
3. Can we convert a Python float into an integer?
4. What is the “self” keyword used for in Python programming language?
5. How can you reverse a string in Python?

## CHAPTER 2

# Python for Machine Learning.(ML)

### Introduction

Python is one of the most popular languages being used in the field of data analytics and artificial intelligence. Over the years, Python language has come to support several thousands of artificial intelligence and data science related libraries.

Python has libraries to explore the data, perform data visualization using the existing data, normalize the data, use machine learning algorithms to derive insights, and predict results based on the algorithms.

### Structure

- Why Python?
- Python libraries for ML
- Introduction to Python Pandas
- Numpy and Scipy
- Matplotlib
- Scikit-learn
- Seaborn
- A quick introduction to Tensorflow, Keras, and PyTorch

### Objective

This chapter will help readers familiarise themselves with the different machine learning libraries supported by Python. Readers can try out the various examples mentioned in the chapter to get a hands-on understanding of how the libraries work.

## Why Python?

One of the other reasons that Python programming is used for is rapid prototyping. Rapid prototyping allows a user to validate an idea quickly and effectively. Since machine learning uses computationally intensive statistics and formulas in the background, due to the ease of python programming language, users can quickly build a small prototype and get an idea about their analysis methodologies and possible results of their algorithm.

Python is widely preferred in the field of machine learning due to the following advantages:

- Its data handling capacity is good
- Python is an extendable language and has support to interact with almost all the other languages and platforms
- Python is an open-source and hence, accessible to more people across the globe
- As a language, Python is quite easy to learn and user friendly

In this chapter, we will discuss several python libraries that provide support for **machine learning (ML)**. We will look at the various functions provided by these libraries and their usage in the field of ML.

## Python libraries for ML

Python provides several libraries for working with machine learning data. Each of these libraries provides different kinds of support to work with big datasets. Many of these libraries are used in the preprocessing stage, where your data is being converted to a machine-readable format.

We will present a brief introduction to these libraries and some of the most common functionalities offered by them in the following sections.

## Pandas

Python pandas are one of the most important libraries used to work with datasets in Python. Python pandas can be installed using the following command on your terminal.

Here is the installation command for Python pandas:

```
pip install pandas
```

Or the following command:

```
pip3 install pandas
```

Due to its extensive use throughout the code, you can observe that pandas are generally imported as pd since it's easier to use throughout the program. You can import pandas in your python code using the following statement:

```
import pandas as pd
```

## **Pandas DataFrames**

In the upcoming section, we will introduce you to a data structure in Python pandas known as DataFrames. DataFrames help to access the data much more comfortable and faster in Python.

### **Introduction**

Pandas DataFrames are one of the most commonly used data structures for storing the data. DataFrames let you store the observations in the form of rows and columns.

An example of how you can create DataFrames using pandas:

```
df_dict = {"fruits": ["Apple", "Mango", "Banana",
"Pomegranate"],
"vegetables": ["Carrot", "Zucchini", "Onion", "Tomato"],
"cereals": ["Corn", "Wheat", "Barley", "Rice"],
"price_fruits_per_kg": [900, 100, 60, 75] }

import pandas as pd
groceries = pd.DataFrame(df_dict)
print(groceries)
```

Here is the output of the program which creates a DataFrame:

	fruits	vegetables	cereals	price_fruits_per_kg
0	Apple	Carrot	Corn	900
1	Mango	Zucchini	Wheat	100
2	Banana	Onion	Barley	60
3	Pomegranate	Tomato	Rice	75

*Figure 2.1: Output of Pandas DataFrames creation*

As you can see in the above output, there are default indexes that have been assigned to the DataFrame. You can also assign your indexes to your DataFrames.

Here is an extension of the above program where you can assign your own keys:

```
df_dict = {"fruits": ["Apple", "Mango", "Banana",
"Pomegranate"],
"vegetables": ["Carrot", "Zucchini", "Onion", "Tomato"],
"cereals": ["Corn", "Wheat", "Barley", "Rice"],
"price_fruits_per_kg": [900, 100, 60, 75] }

import pandas as pd
groceries = pd.DataFrame(df_dict)
groceries.index = ["f1", "f2", "f3", "f4"]
print(groceries)
```

Here is the output of the DataFrames with self-defined indexes program:

	fruits	vegetables	cereals	price_fruits_per_kg
f1	Apple	Carrot	Corn	900
f2	Mango	Zucchini	Wheat	100
f3	Banana	Onion	Barley	60
f4	Pomegranate	Tomato	Rice	75

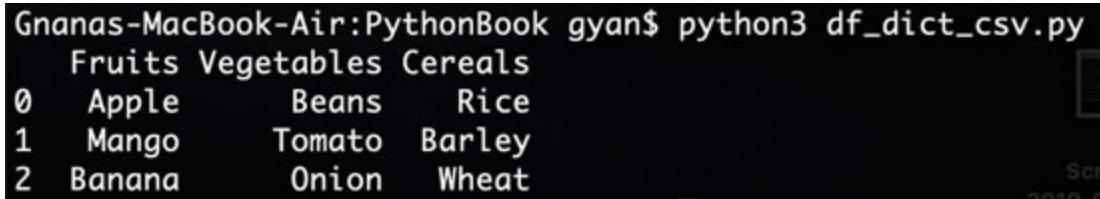
*Figure 2.2: DataFrames with self-defined indexes*

One of the most frequently used and most important commands to create a DataFrame in Python pandas is to read the CSV file and create a DataFrame using the CSV. The CSV file to be used is dict.csv

```
import pandas as pd
```

```
groceries = pd.read_csv('dict.csv')
print(groceries)
```

Here is the output of the CSV data exported program:



A terminal window titled 'Gnanas-MacBook-Air:PythonBook gyan\$' displays the command 'python3 df\_dict\_csv.py'. Below the command, the data is printed in a tabular format:

	Fruits	Vegetables	Cereals
0	Apple	Beans	Rice
1	Mango	Tomato	Barley
2	Banana	Onion	Wheat

*Figure 2.3: CSV data exported in pandas*

## Indexing DataFrames

You can use the square braces ([]) or the dot(.) operator to index the columns in a DataFrame.

You can use single square brackets to print the column as a `pandas.Series` object, or you can use two square brackets to print the columns as a `pandas.DataFrame` object.

Dataset to be used - `dict.csv`

Here is an example of how you can index the columns:

```
import pandas as pd

df = pd.read_csv('dict.csv')

print(df.Vegetables)

#Print the Vegetables column as pandas Series
print(df['Vegetables'])

#Print the Vegetables column as pandas DataFrame
print(df[['Vegetables']])

#Index both Cereals and Vegetables column as DataFrame
print(df[['Cereals', 'Vegetables']])

print("The first element of column Cereal is ")
#Index the first observation of a column
print(df['Cereals'][0])
```

Here is the output of the pandas indexing program:

```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 df_dict_index.py
 0    Beans
 1   Tomato
 2   Onion
Name: Vegetables, dtype: object
 0    Beans
 1   Tomato
 2   Onion
Name: Vegetables, dtype: object
  Vegetables
 0    Beans
 1   Tomato
 2   Onion
  Cereals  Vegetables
 0   Rice      Beans
 1  Barley     Tomato
 2  Wheat      Onion
The first element of column Cereal is
Rice
```

*Figure 2.4: Python Pandas Indexing*

## Pandas loc and iloc

In Python pandas, `loc` and `iloc` can also be used for indexing the columns and the individual observations. To use `loc`, we need to use the labels, whereas `iloc` uses integers to index.

The use of `loc` and `iloc` is becoming more and more common in python pandas and is being increasingly used everywhere.

`loc` and `iloc` use the indexes to refer to the DataFrame elements.

Here is an example of how we can use `loc` and `iloc` in a DataFrame:

```
import pandas as pd

df = pd.read_csv('dict.csv')

df.index = ['0', '1', '2']

#Indexing the first row
print(df.loc[['0']])
```

```
#Indexing the first two row elements using iloc  
print(df.iloc[0][0:2])
```

Here is the output for the Python `loc` and `iloc` example:

```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 df_dict_loc_iloc.py  
    Fruits Vegetables Cereals  
0   Apple      Beans     Rice  
Fruits        Apple  
Vegetables    Beans
```

*Figure 2.5: Python pandas loc and iloc*

## Numpy

NumPy stands for **Numerical Python**, which can be used for performing all the mathematical and logical operations on multi-dimensional arrays in Python. NumPy arrays are the primary data structure of the library `numpy`. They are similar to lists in Python.

The `numpy` library needs to be installed to use the functions supported by this library. You can use the following command to install the `numpy` library:

```
pip install numpy
```

or the following command:

```
pip3 install numpy
```

`numpy` is imported in the code using the following syntax:

```
import numpy as np
```

## Numpy array syntax and parameters

Numpy arrays can also be defined with certain parameters. Here is the syntax to declare a numpy array:

```
numpy.array(object, dtype=None, copy=True, order='K',  
subok=False, ndmin=0)
```

Important parameters:

- `dtype`: `dtype` is used to mention the type of array that you want to declare. It can be an integer, complex, a float, and so on.

- `ndmin`: Specifies the minimum number of dimensions that the resulting array should have.

The other parameters are not used often and hence, have not been explored deeply.

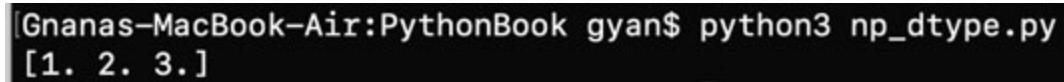
Here is an example of how a numpy array can be declared using the `dtype` parameter:

```
import numpy as np

arr = np.array([1,2,3], dtype=float)

#print array - output is in the form of float numbers
print(arr)
```

Here is the output for the numpy array example:



```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 np_dtype.py
[1. 2. 3.]
```

*Figure 2.6: Numpy array example*

## Numpy ndarray creation

The `ndarray` is one of the most commonly used objects in numpy, which is an n-dimensional array. Unlike Python lists, which can have different elements, ndarrays represent a collection of objects of the same type.

Example code snippet to create and use numpy arrays:

```
import numpy as np

#Create lists in python
fruit_cost = [20, 60, 90, 40, 50]
vegetable_cost = [10, 20, 30, 40, 50]

#Convert the lists into numpy arrays
fruit_np = np.array(fruit_cost)
vegetable_np = np.array(vegetable_cost)

#print the array type of the object
print(fruit_np)
print(type(fruit_np))
```

Here is the output for the numpy ndarray example:

```
Gnanas-MacBook-Air:PythonBook gyan$ python3 np_example.py
[20 60 90 40 50]
<class 'numpy.ndarray'>
```

*Figure 2.7: Numpy ndarray example*

## Subsetting numpy arrays

Numpy arrays can easily be subsetted using square brackets ([]). You can use the higher than (>) or less than(<) symbol to subset the arrays.

Here is an example of array subsetting:

```
import numpy as np

arr = np.array([1,2,3,4,5,6,7,8,9,10])

#Subset the elements greater than 5
print(arr[arr > 5])
```

Here is the output for the numpy array subset example:

```
Gnanas-MacBook-Air:PythonBook gyan$ python3 np_subset.py
[ 6  7  8  9 10]
```

*Figure 2.8: Numpy array subset example*

## SciPy

SciPy stands for **Scientific Python** and offers various functionalities that help in scientific computation. Scipy is built on top of NumPy and operates on numpy arrays.

SciPy complements numpy in a way that contains a fully-featured version of linear algebra, whereas numpy contains just the basic version. Scipy has added data science functionalities over numpy.

The `scipy` library needs to be installed to use the functions supported by this library. You can use the following command to install the `scipy` library:

```
pip install scipy
```

Or the following command:

```
pip3 install scipy
```

SciPy functions can be imported using the following syntax.

where io is a scipy module being imported as sio:

```
from scipy import io as sio
```

## Functions of SciPy

SciPy has a package named `special`, which has numerous functions like exponential, cubic root, log related functions, permutations and combinations, and many more. Here are some of the functions that `scipy.special` provides:

- **Exponential function:**

Exponential function: `exp10(a/arr)`

The exponential function is used to find the result for 10 to the power of given parameters; the parameters can be a single-digit or an array.

Example code snippet:

```
from scipy.special import exp10

val = exp10([1,10])

#Output : [1.e+01 1.e+10]
print(val)
```

- **Cubic root function:**

Cubic root function: `cbrt(a)`

The cubic root function is used to find the cubic root of a given number.

Example code snippet:

```
from scipy.special import cbrt

num = cbrt(27)

#Output : 3.0
print(num)
```

- **Permutations and combinations:**

Permutations function: `perm(a,b)`

The permutations function is used to find the permutations of  $a$  things in  $b$  different ways.

Example code snippet:

```
from scipy.special import perm

a = 4
b = 2
permutation = perm(4,2)

#Output: 12.0
print(permutation)
```

- Combinations function: `comb(a,b)`

The combinations function is used to find the combinations of  $a$  things taken  $b$  at a time.

Example code snippet:

```
from scipy.special import comb

a = 4
b = 2
combination = comb(4,2)

#Output: 6.0
print(combination)
```

## Linear algebra with SciPy

Linear algebra is an implementation of BLAS and ATLAS LAPACK libraries. The input to a linear algebra function is a two-dimensional array, and the output is also a two-dimensional array.

**Package name:** `linalg`

Example code snippets using the linear algebra package:

Calculating the determinant of a 2-D matrix:

```
from scipy import linalg
import numpy as np
```

```
#define square matrix
arr = np.array([ [1,2], [3,4] ])

#calculate the determinant
determinant = linalg.det(arr)

#print(determinant)
```

Calculating the inverse of a 2-D matrix:

```
from scipy import linalg
import numpy as np

#define square matrix
arr = np.array([ [6,7], [4,5] ])

#calculate the inverse
inverse = linalg.inv(arr)

#print(inverse)
```

## Matplotlib

Matplotlib is a library that's used for plotting graphs to gain insights into your data. `Matplotlib.pyplot` is a package that is used to plot 2-D graphs in Python.

The `matplotlib` library needs to be installed to use the functions supported by this library. You can use the following command to install the `matplotlib` library:

```
pip install matplotlib
```

Or the following command:

```
pip3 install matplotlib
```

Matplotlib functions can be imported using the following syntax where `#pyplot` is the `matplotlib` module being imported as `plt`:

```
from matplotlib import pyplot as plt
```

## Python matplotlib – Different plots

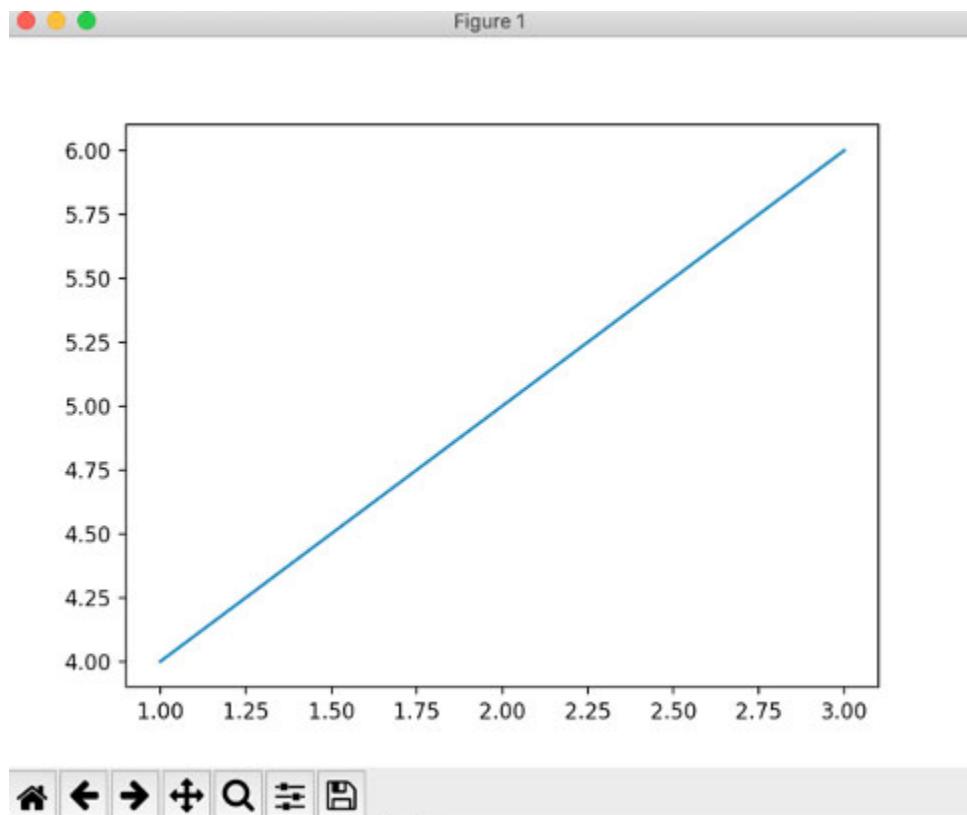
Example code snippet to plot a generic graph using `pyplot.matplotlib`:

```
from matplotlib import pyplot as plt

#Creating the plot
plt.plot([1,2,3],[4,5,6])

#printing the created plot
plt.show()
```

Here is a simple linear plot created using matplotlib:



*Figure 2.9: Simple plot created by matplotlib*

Here are some plots and their descriptions that can be created using matplotlib to analyze our dataset.

### Bar graph

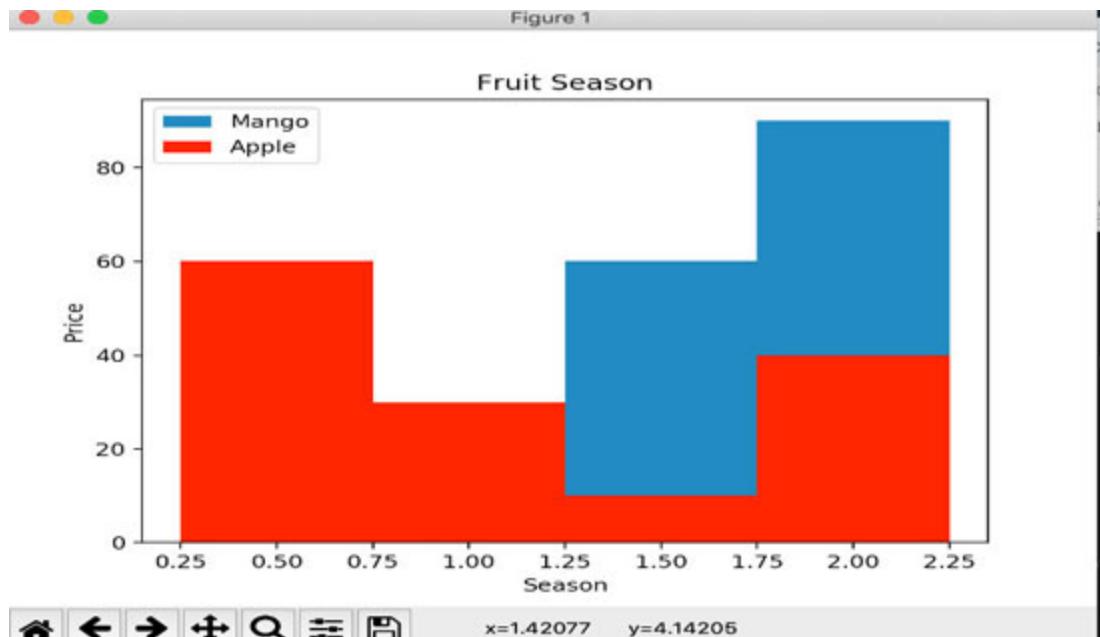
A bar graph uses bars to compare data between different categories. It is also a great tool to observe changes over time. You can use horizontal bars or vertical bars to represent your data.

Here is the example code for plotting a bar graph:

```
from matplotlib import pyplot as plt

plt.bar([0.5,1.0,1.5,2.0],[10,30,60,90],
label="Mango",width=.5)
plt.bar([0.5,1.0,1.5,2.0],[60,30,10,40],
label="Apple", color='r',width=.5)
plt.legend()
plt.xlabel('Season')
plt.ylabel('Price')
plt.title('Fruit Season')
plt.show()
```

Here is the example of a bar graph plotted using matplotlib:



*Figure 2.10: Pyplot Bar Graph*

## Histogram

A histogram focuses on a single entity and shows its distribution. The values in a histogram are generally split into intervals to plot the

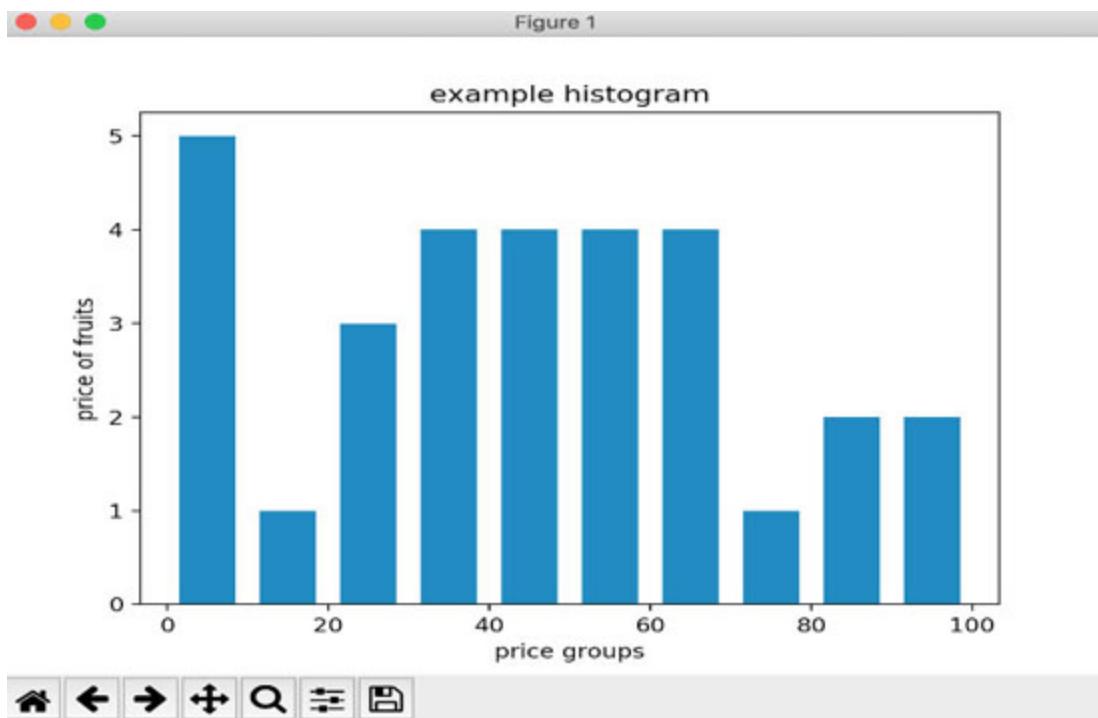
distribution.

Here is the example code for plotting a histogram:

```
from matplotlib import pyplot as plt

price_season =
[1,10,2,20,3,30,4,45,5,65,43,54,57,87,65,34,85,37,56,75,98,99,2
6,29,35,42,46,53,67,68]
bins = [0,10,20,30,40,50,60,70,80,90,100]
plt.hist(price_season, bins, histtype='bar', rwidth=0.7)
plt.xlabel('price groups')
plt.ylabel('price of fruits')
plt.title('example histogram')
plt.show()
```

Here is an example of a histogram plotted using matplotlib:



*Figure 2.11: Pyplot Histogram*

## Scatter plot

A scatter us the opportunity to compare the distribution of more than one variable. It helps us find relationships between variables and understand

how one variable affects the other.

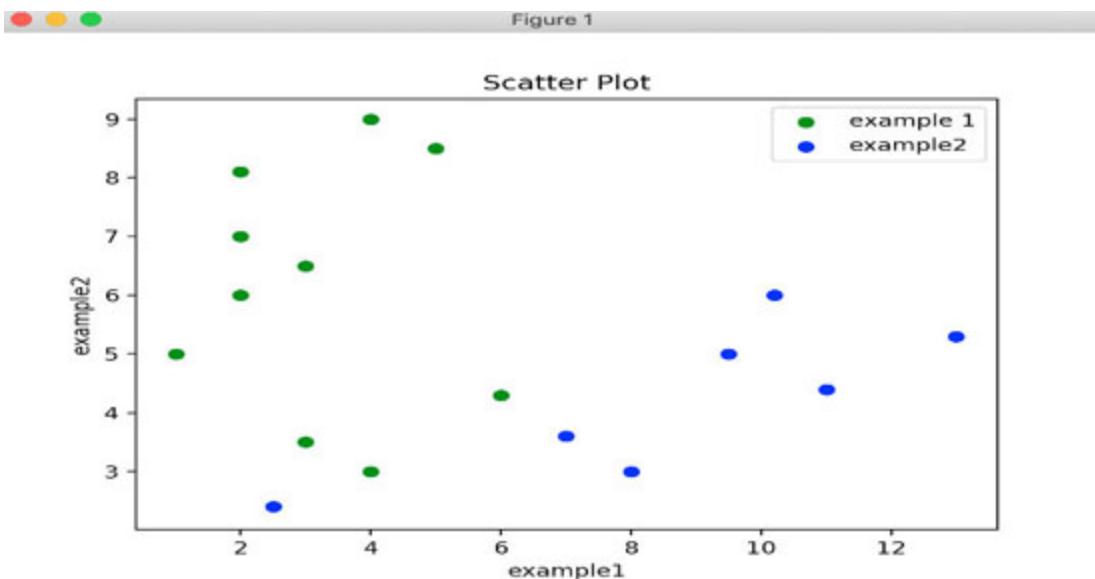
Here is the example code for plotting a scatter plot:

```
import matplotlib.pyplot as plt
x = [2,1,3,4,2,5,4,3,6,2]
y = [6,5,6.5,3,7,8.5,9,3.5,4.3,8.1]

x1=[8,2.5,7,9.5,11,10.2,13]
y1=[3,2.4,3.6,5,4.4,6,5.3]

plt.scatter(x,y, label='example 1',color='g')
plt.scatter(x1,y1,label='example2',color='b')
plt.xlabel('example1')
plt.ylabel('example2')
plt.title('Scatter Plot')
plt.legend()
plt.show()
```

Here is an example of a scatter plot plotted using matplotlib:



*Figure 2.12: Matplotlib scatter plot*

## Pie chart

A pie chart gives us a visual insight into what percentage different categories occupy when compared to the overall percentage. It gives us an insight into how different categories of data are distributed in the dataset.

Here is the example code for plotting a pie chart:

```
import matplotlib.pyplot as plt

breakfast = [1,2,3,4,5]

Fruits = [6,13,8,12,9]
Cereal = [1,4,5,2,6]
Meat = [6,10,8,4,3]
Eggs = [3,7,2,6,12]
slices = [6,3,4,12]

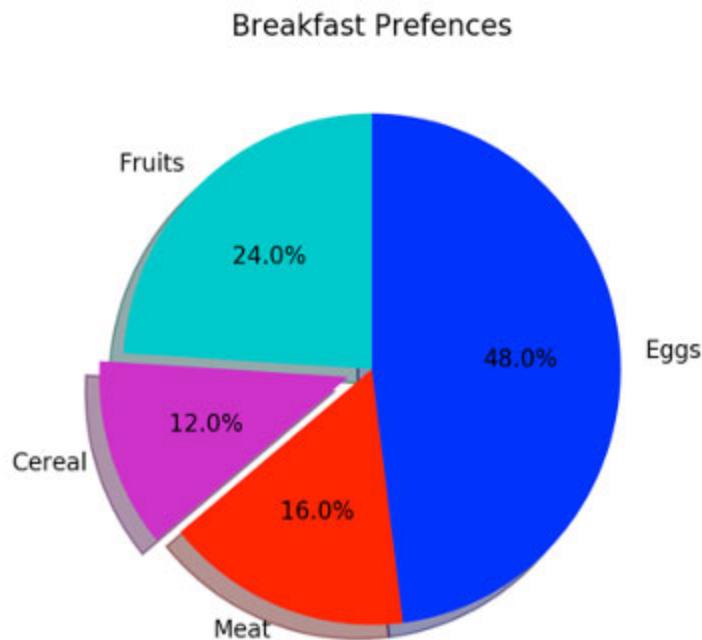
breakast_options = ['Fruits', 'Cereal', 'Meat', 'Eggs']
cols = ['c', 'm', 'r', 'b']

plt.pie(slices,
        labels=breakast_options,
        colors=cols,
        startangle=90,
        shadow= True,
        explode=(0,0.1,0,0),
        autopct='%1.1f%%')

plt.title('Breakfast Preferences')
plt.show()
```

Here is an example of a pie chart plotted using matplotlib:

Figure 1



*Figure 2.13: Matplotlib pie chart*

## Scikit-learn

Scikit-learn is a library that not only offers several machine learning functions, but it also contains some built-in datasets which can be used for exploration and analysis. It's an open-source library and supports numerous algorithms like KNN, Random Forest, SVM, XGBoost, and so on. It is built on top of NumPy and is used frequently in industries and by developers.

The scikit-learn library needs to be installed to use the functions supported by this library. You can use the following command to install the scikit-learn library:

```
pip install sklearn
```

Or the following command:

```
pip3 install sklearn
```

sklearn functions can be imported using the following syntax where `#train_test_split` is the `sklearn` function being imported from the module `sklearn.model_selection`:

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

## Scikit-learn – Datasets

You can import the existing datasets provided by `scikit-learn` to explore some machine learning algorithms on those datasets.

### **Example code for using datasets:**

We import the datasets module from `sklearn`, using which we can load the dataset of our choice. Each dataset contains some data and metadata about that data. The data is present in the `.data` part:

```
from sklearn import datasets
digits = datasets.load_digits()
print(digits.data)
```

Here is the output of importing the `digits` dataset from the `sklearn` library:

```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 scikit.py
 [[ 0.  0.  5. ... 0.  0.  0.]
 [ 0.  0.  0. ... 10. 0.  0.]
 [ 0.  0.  0. ... 16. 9.  0.]
 ...
 [ 0.  0.  1. ... 6.  0.  0.]
 [ 0.  0.  2. ... 12. 0.  0.]
 [ 0.  0.  10. ... 12. 1.  0.]]
```

*Figure 2.14: Output of the digits dataset*

## Scikit-learn – Principal component analysis

Scikit-learn can be used for principal component analysis, where we try to find out which features are the most important in our dataset. Features, also known as attributes, are the deciding factors for your machine learning model predictions. Features describe the datasets as a whole, and hence, it is essential to select the right features for your modeling.

**Principal Component Analysis (PCA)** helps you identify the top significant features in your feature list.

Here is an example code snippet that shows the workings of PCA:

```
from sklearn import datasets
from sklearn.decomposition import PCA

digits = datasets.load_digits()

random_pca = PCA(n_components=2, svd_solver='randomized')

#Randomized PCA performs better when there are more number of
dimensions
rpca_model = random_pca.fit_transform(digits.data)

# Comparing with a regular PCA
pca = PCA(n_components=2)

# Fit-transform will transform the data to the model
pca_model = pca.fit_transform(digits.data)

# Printing the randomized and normal pca data
print(rpca_model)
print(pca_model)
```

Here is an example of the output that shows the workings of PCA:

```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 scikit.py
[[ -1.25946653  21.27488137]
 [  7.95761224 -20.76870363]
 [  6.99192291 -9.9559829 ]
 ...
 [ 10.80128338 -6.96025218]
 [ -4.87209783  12.42394959]
 [ -0.344391    6.36555434]]
[[ -1.25946768  21.27488478]
 [  7.95761238 -20.76869311]
 [  6.99192327 -9.95599143]
 ...
 [ 10.80128312 -6.96025292]
 [ -4.87210015  12.42395339]
 [ -0.34439047  6.36554378]]
```

*Figure 2.15: PCA Randomized versus normal on large datasets*

## Scikit-learn – Pre-processing

The preprocessing module from scikit-learn offers several functionalities like encoding the data to different formats, splitting the data into training and test sets, and many more.

Pre-processing is essential as data needs to be in a mathematical format for the machine learning algorithm to process it. There are also several other functions offered by the preprocessing module that are extremely useful for machine learning.

Here is an example code snippet of some preprocessing functionalities:

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier

#Define some data with labels
data = [ [0,1], [1,2], [2,3], [3,4] ]
label = [9,8,2,7]

label = LabelBinarizer().fit_transform(label)

#Using an SVM Gamma Classifier
svc_classifier =
OneVsRestClassifier(estimator=SVC(gamma='scale',
random_state=0))

#Creates binary labels
print(svc_classifier.fit(data, label).predict(data))
```

Here is the output of `LabelBinarizer`:

```
Gnanas-MacBook-Air:PythonBook gyan$ python3 preprocessing.py
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

*Figure 2.16: LabelBinarizer on an array example*

**Example 2: Creating labels for categorical data (data that is non-numeric):**

```
from sklearn import preprocessing
```

```

data = ['Apple', 'Mango', 'Banana', 'Chickoo', 'Jackfruit']
labels = ['T','F','F','T','T']

#Creating an object that represents the label encoder
encoder_label = preprocessing.LabelEncoder()

data_encoded = encoder_label.fit_transform(data)
labels_encoded = encoder_label.fit_transform(labels)

print(data_encoded)
print(labels_encoded)

```

Here is the output of `LabelEncoder`:

```

Gnanas-MacBook-Air:PythonBook gyan$ python3 preprocessing_2.py
[0 4 1 2 3]
[1 0 0 1 1]

```

*Figure 2.17: LabelEncoder from sklearn preprocessing*

## Seaborn

Seaborn is also a visualization library built on top of `matplotlib`. It is used to visualize the various patterns in the data using different kinds of graphical representations of the given dataset.

Seaborn is complementary to `matplotlib` and provides a high-level interface to visualize data.

The `seaborn` library needs to be installed to use the functions supported by this library. You can use the following command to install the `seaborn` library:

```
pip install seaborn
```

Or the following command

```
pip3 install seaborn
```

Seaborn is generally imported using the following syntax:

```
import seaborn as sns
```

## Seaborn – Different plots

Seaborn is used to drawing different kinds of plots. Here are some example plots that can be drawn using the seaborn Python library:

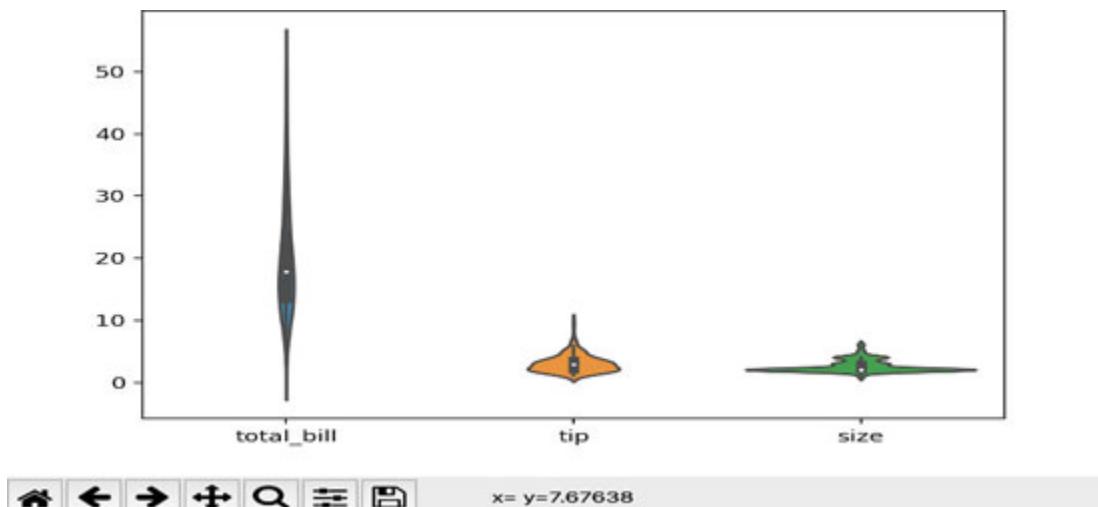
```
import matplotlib.pyplot as plt
import seaborn as sns

# Load the "tips" dataset
df = sns.load_dataset("tips")

# Create violinplot using seaborn
sns.violinplot(data=df)

# Show the plot
plt.show()
```

Here is a sample plot plotted using seaborn:



*Figure 2.18: Seaborn example*

## Seaborn: Scatter Plots

A scatter us to compare the distribution of more than one variable. It helps us find relationships between variables and understand how one variable affects the other. Seaborn offers simple functions to create scatter plots

Here is the example code for plotting a scatter plot using seaborn:

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

```

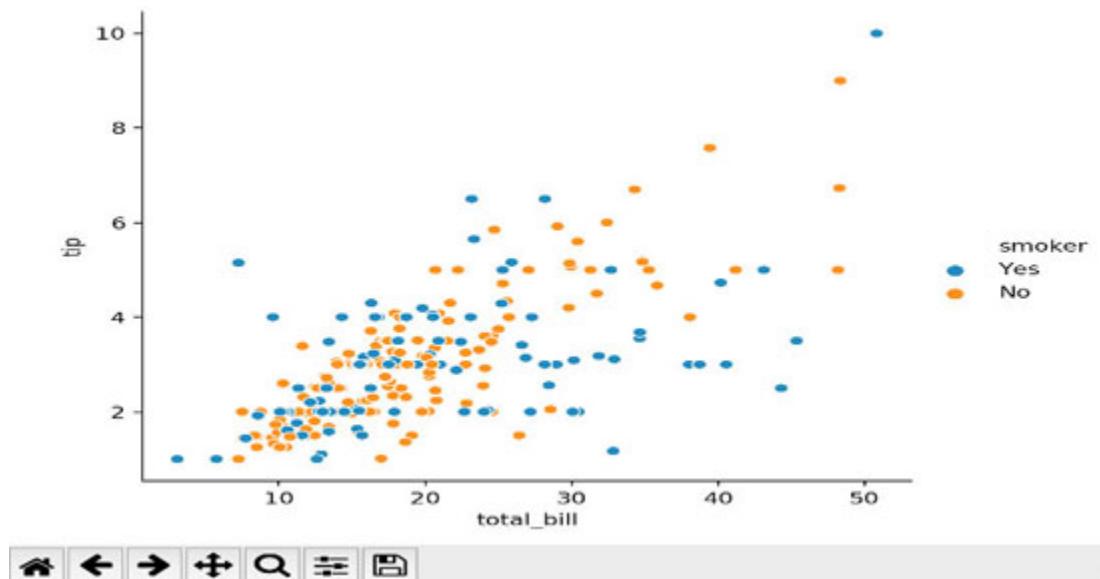
# Load the "tips" dataset
df = sns.load_dataset("tips")

#Creating a scatter plot with hue(colour) to one of the
variables
sns.relplot(x="total_bill", y="tip", hue="smoker", data=df);

# Show the plot
plt.show()

```

Here is the output for plotting a scatter plot using seaborn:



*Figure 2.19: Scatter plot in Seaborn*

## Tensorflow and Keras

TensorFlow is an open-source library for machine learning research. TensorFlow offers Keras APIs, which help in creating deep learning models.

Deep learning uses underlying neural networks that mimic the neurons in our brains. We will study more about deep learning in our upcoming chapters.

The tensorflow library needs to be installed to use the functions supported by tensorflow and the Keras APIs. You can use the following command to install the tensorflow library:

```
pip install tensorflow
```

Or the following command:

```
pip3 install tensorflow
```

Tensorflow is generally imported using the following syntax:

```
import tensorflow as tf
```

Here is an example snippet of how `tensorflow` and Keras APIs can be used for machine learning:

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist
(data_train, label_train), (data_test, label_test) =
mnist.load_data()
data_train, data_test = data_train / 255.0, data_test / 255.0

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(512, activation=tf.nn.relu),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(data_train, label_train, epochs=2)
model.evaluate(data_test, label_test)
```

Here is the output of a tensorflow Keras API example:

```
Gnanas-MacBook-Air:PythonBook gyan$ python3 tensorflow_ex.py
WARNING: Logging before flag parsing goes to stderr.
W0621 16:14:44.438549 4559463872 deprecation.py:506] From /usr/local/lib/python3
.7/site-packages/tensorflow/python/ops/init_ops.py:1251: calling VarianceScaling
.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and wil
l be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the c
onstructor
2019-06-21 16:14:44.858881: I tensorflow/core/platform/cpu_feature_guard.cc:142]
Your CPU supports instructions that this TensorFlow binary was not compiled to
use: AVX2 FMA
Epoch 1/2
60000/60000 [=====] - 7s 116us/sample - loss: 0.2197 -
acc: 0.9347
Epoch 2/2
60000/60000 [=====] - 6s 102us/sample - loss: 0.0958 -
acc: 0.9702
10000/10000 [=====] - 1s 59us/sample - loss: 0.0755 - a
cc: 0.9750
```

*Figure 2.20: TensorFlow Keras API example using MNIST*

## PyTorch

As machine learning became more and more compute-intensive, GPUs on the system started coming into the picture as they also offer power that can be used for computation. PyTorch is a library that offers the same functionality as numpy but can leverage the GPUs in the system.

It is predominantly used for deep learning and offers excellent flexibility and speed.

Tensors used in PyTorch are similar to the numpy ndarrays that we have studied before.

The PyTorch library needs two installations to use the functions supported by PyTorch - `torch`, and `torchvision`. You can use the following command to install the libraries:

```
pip install torch
pip install torchvision
```

Or the following command:

```
pip3 install torch
pip install torchvision
```

The `torch` is generally imported using the following syntax:

```
import torch
```

Here is an example code snippet of how the PyTorch library:

```
from __future__ import print_function
import torch

#Construct a matrix using torch
matrix = torch.empty(4,2)
print(matrix)

#Construct a random matrix using torch
matrix_rand = torch.rand(4,2)
print(matrix_rand)
```

Here is the output snippet of how the PyTorch library works:

```
[Gnanas-MacBook-Air:PythonBook gyan$ python3 torch_ex.py
tensor([[1.1210e-44, -0.0000e+00],
       [0.0000e+00, 0.0000e+00],
       [0.0000e+00, 1.4013e-45],
       [0.0000e+00, 0.0000e+00]])
tensor([[0.1724,  0.1182],
       [0.5985,  0.3070],
       [0.4469,  0.6544],
       [0.8180,  0.9236]])
```

*Figure 2.21: PyTorch example*

## Conclusion

Python offers a plethora of libraries that can work magic with datasets and provide seamless execution of machine learning algorithms. Python, being a modular language, allows us to write our libraries as well for Machine Learning, which could be a slightly tweaked version of the original algorithms.

Python is one of the best languages for any individual who is at any stage in their machine learning journey as it is a very developer-friendly and accessible language to work with.

## A quick refresher quiz

1. What function is used in pandas to calculate the number of categorical values for each categorical attribute (Hint: count the different values of a categorical variable)
2. Which kind of graphs are ‘bins’ used in?
3. What is the function present in scikit used to split the training data?
4. Which module is generally imported as ‘tf’ in Python?
5. What is a violinplot in seaborn?

## CHAPTER 3

# Introduction to Machine Learning (ML)

### Introduction

**Artificial Intelligence (AI)**, as the world understands today, is the science behind creating intelligent systems that can complement humans in their ways of thinking. **Machine Learning (ML)** is a subset of AI, where machines are trained to learn from data and information provided to them.

Machine learning is not just restricted to providing recommendations or predicting values; it's also about simulating the way processes work in a human brain. This new-age technology is also being termed as Deep Learning.

### Structure

- The origin story
- Machine learning data
- Branches of machine learning
- Applications of machine learning
- Exploratory data analysis
- Predictive modeling
- Machine learning case study

### Objectives

This chapter aims to introduce the readers to the basics of machine learning and the different concepts involved in machine learning. The readers will be able to visualize the path of machine learning, starting from the origin story

to the various industrial applications of machine learning prevalent in today's times.

The chapter ends with a small quiz, which will serve as a refresher for the readers containing questions from the chapter as well as questions that the readers will have to research on, to get the correct answers.

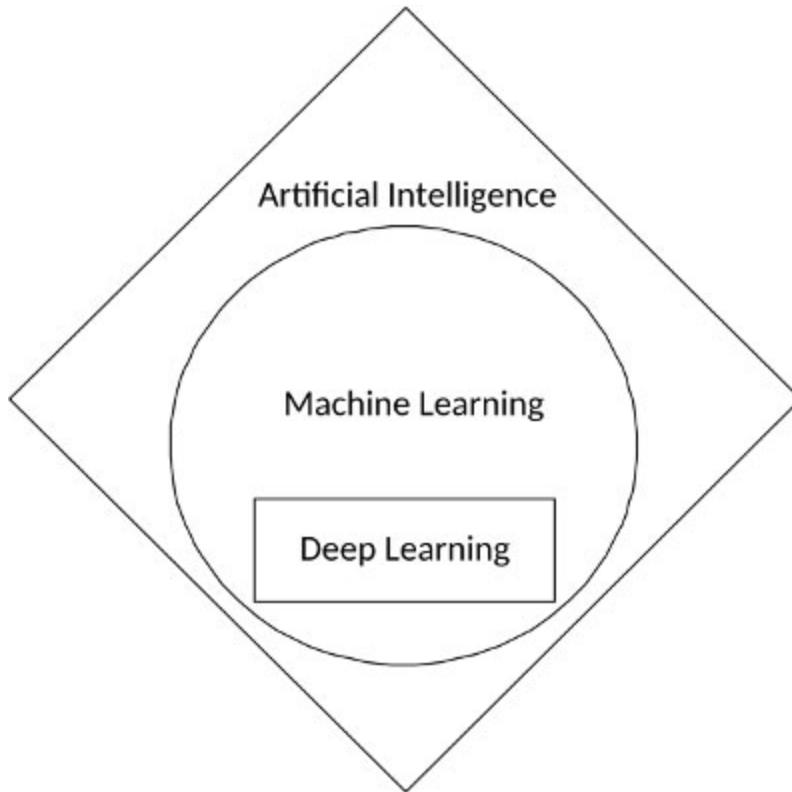
## **The origin story**

The origin story that we would like to introduce is set in 1950 when Alan Turing designed the Turing tests to test a machine's real learning ability. Turing had proposed that a human judge evaluate the natural language-based conversations between two machines, and the channel of communication was restricted to a text-only channel such as a keyboard or a screen.

From text-only conversations, we have moved to an era of voice-first devices where there are Siris, Alexas, and Google Homes, which can render speech-to-text translations and make human life much more comfortable.

## **The big picture**

Before deep-diving into machine learning, it is essential to understand the various terminologies associated with machine learning. Here is an image that depicts the fundamental machine learning terms:



*Figure 3.1: Differentiating between AI and ML*

[Figure 3.1](#) gives us a clear idea of the various terminologies associated with machine learning as a whole. Natural Language Processing also falls under the AI umbrella and uses ML but is a whole different topic on its own:

- **Artificial Intelligence** is the bigger umbrella under which several topics explore the processes or techniques that can be used to create an intelligent machine.
- **Machine Learning (ML)** is a branch that uses statistical mathematics and related algorithms to achieve the same goal.
- **Deep Learning** is a sub-branch of machine learning, which we will explore further in this chapter. Deep learning deals with creating ML models that mimic the functionality of our brain, that is, the way neurons are connected and function together to draw inferences from the presented information.

For all machine learning related work, data is one of the essential pivotal points using which we can build our ML models. Data acts as the single

most significant entity in our whole process of machine learning and needs to be handled very carefully.

There is a vast amount of data that is being collected every minute, every day, to derive insights on user patterns, customer behavior, performance numbers for various products/websites across the globe. In the following chapters, we will understand these different use cases for ML the different ways to work with these use cases.

## A brief glimpse into machine learning data

One of the most important elements of machine learning is the data. Data drives the insights that we get from the machine learning algorithms. In this section, we will look at the different types of data and a glimpse into the datasets that we commonly use in ML.

### Types of data

As we already know that data is the bread and butter of machine learning, we need to understand the different elements that describe the data. Data used in machine learning is of different types:

- **Customer data:** Data collected on your platform with the help of customers using your product/platform.
- **Feedback data:** Reviews/feedback collected from customers on your product/framework/platform, and so on.
- **Performance data:** Data that represents the performance of a product/employee overtime.
- **Open-source data:** Open source data can be in the form of public datasets, data crawled from the web, and so on.
- **Internal company data:** This can be internal reports, data collected from the company's products/services, or systems. It could also be of the first three types of data mentioned.

A typical CSV format file is read in python and stored as a DataFrame to perform data cleaning, exploratory data analysis, and data modeling. Data might also need to be cleaned manually/programmatically before being able to read it as a DataFrame using python pandas.

## A glimpse into the dataset

An example of how a dataset might look like:

The following dataset gives us an estimate of house prices for different types of houses in a particular location:

Index	No. of Bedrooms	Area(sq. ft.)	Location	Furnished	Prices(\$)
1	3	1800.23	Downtown	Yes	939874
2	2	1000.10	Midtown	No	456781
3	3	1600.87	Downtown	No	789823

*Table 3.1: An example dataset*

The above-given example dataset has three rows and six columns. Here are some terminologies associated with the dataset:

- **Row 1:** The top row generally comprises of the various column names in your dataset. These column names can be indicative of the kind of data you have in your dataset.
- **The index column:** The index column represents the index of the various rows in your dataset. An index column can be assigned to your dataset using pandas if it isn't present already.
- **Rows:** The rows of readings, also known as observations, describe the data that you have collected.
- **Columns:** The columns, also known as features or attributes, are the most significant parts of your data. The features are the ones that will help us build the models, give us insight into how the models perform, and in general, give us an idea about relationships in the dataset.

## Variables in a dataset

There are different types of variables in a dataset. Each feature falls under one of these kinds of variables. It is essential to understand the different kinds of variables as each type of variable requires a different treatment to convert it into a machine-understandable format.

## Quantitative variables

Quantitative variables in a dataset can be categorized as discrete or continuous. Quantitative variables are necessarily numeric variables:

- **Discrete variables:** Discrete variables are numeric variables that are finite and can be counted/slotted between two intervals. A discrete variable will always be numeric. An example would be the number of customer complaints, the number of reviews for a product, and so on.
- **Continuous variables:** Continuous variables can be numeric, or they can be present in the form of date/time. Continuous variables can take infinite values and do not have discrete values. For example, the marks obtained by a student (in float), the house prices of houses in a district, and so on.

## Categorical variables

Categorical variables are non-numeric variables, and as the name suggests, those variables indicate a category that the feature/attribute falls into. One of the characteristics of categorical variables is that they have no particular order and have finite categories that they can be classified into.

Examples of categorical variables can be: ‘whether a student passed or failed in an exam’, ‘the location of the house’ etc.

In the above-given dataset (Refer [Table 3.1](#)), here is the classification of the variables present:

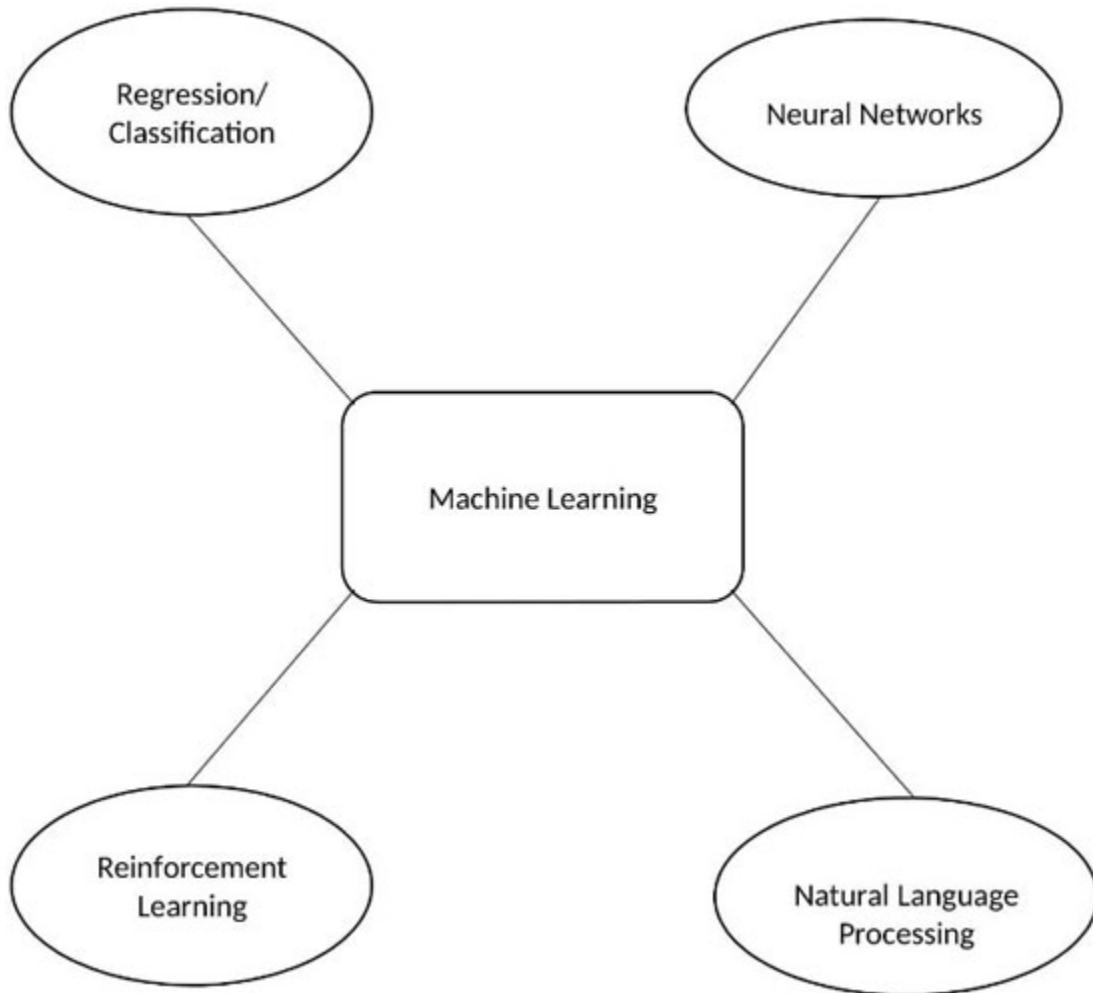
Quantitative	Categorical
No. of Bedrooms	Location
Area(sq. ft.)	Furnished
Prices(\$)	

*Table 3.2: Classification of variables into quantitative and categorical*

## Branches of machine learning

Machine learning has several branches or several applications spread across various industries. Here is a broad classification of Machine Learning and specific examples associated with each classification to understand the usage better.

Here is an image that depicts the branches of machine learning:



*Figure 3.2: Branches of machine learning*

## **Regression/Classification**

Regression and classification are two different methodologies that can help derive insights from data. In this section, we will dive deeper into understanding what these methodologies look like.

### **Classification**

Classification refers to an action where the final output is classified into one or the other classes; that is when the variable to be predicted is categorical and not continuous. Binary classification is a type of classification where

the variable to be predicted is Boolean, that is, True/False, Yes/No, 1/0, and so on.

Multiclass classification is a classification where the variable to be predicted is predicted to fall in different classes.

## Examples for classification

**Binary classification:** Predicting if a customer will purchase a product on your platform(Yes/No).

**Multiclass classification:** Predicting if a customer will buy a product from a collection of products(finite). In this example, the collection of products (a finite number) is the number of classes.

## Regression

Regression is performed when the variable to be predicted a real (or continuous) variable. Continuous variables here refer to a quantitative variable; they can be either discrete or continuous. Several great Machine Learning (ML) algorithms are regression algorithms. Linear Regression, Logistic Regression, and many more are some of the basic ML algorithms which act as the foundational pillars for machine learning.

## Examples for regression

Predicting the house prices of houses in a location, predicting the sales numbers of an organization for the next quarter, and so on.

## Neural networks

Neural networks are a sub-branch of machine learning, which is also known as deep learning. As the name suggests, the machine learning models are created using a neuron model that mimics the neurons and the networks in our brain.

The neural networks are known to perform deep learning since the features are taken through several layers before predicting the output variable. The inner layers in neural networks are also known as **hidden layers**.

## Examples for neural networks

Image processing using **Convolutional Neural Networks (CNN)**.

## Reinforcement learning

Reinforcement learning is a sub-branch of Machine Learning (ML) which is aimed to maximize rewards/notion of rewards in a given situation. It is generally used to find out the best path to take given various paths in a situation.

### Example of reinforcement learning

The best way to win an award while using slot machines at casinos.

## Natural language processing

Natural language processing falls under AI and is one of the most talked-about technologies in the field of ML and AI today. Natural language processing encompasses speech-to-text, text-to-speech, and other methodologies where a machine is taught to speak the natural language of humans.

Natural language is used in all of the voice-first devices like Amazon Alexa, Google Home, Microsoft Cortana, and many more.

### Examples of natural language processing

Chatbots used in websites for answering common queries.

## Applications of machine learning

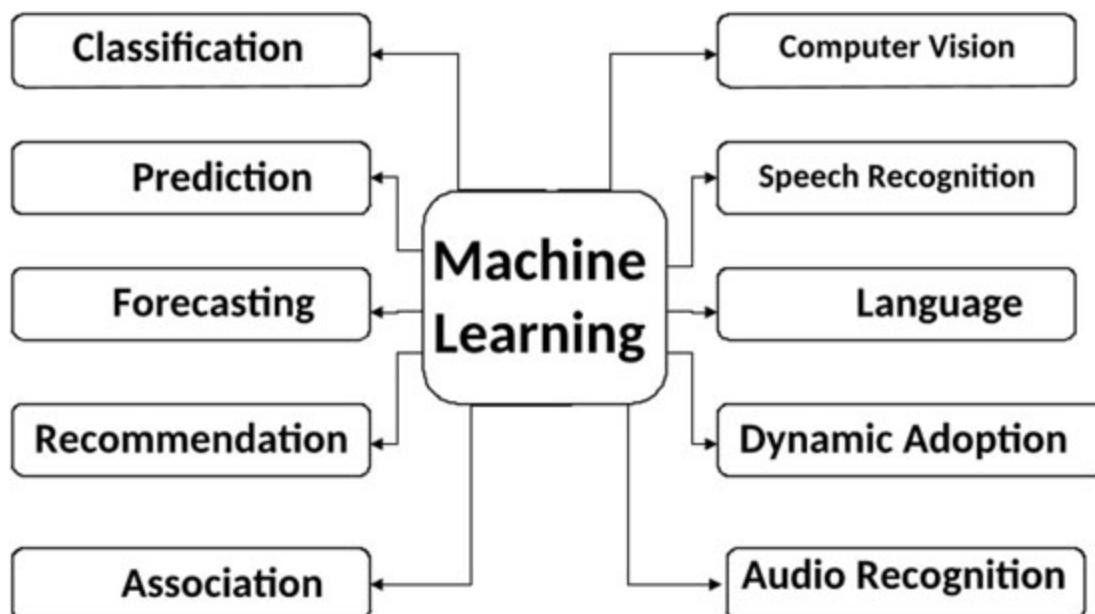
**Machine learning (ML)** has several applications that spawn across industries. We can see some of the examples from the above-stated branches of machine learning. In this section, we will explore some everyday use cases that use different types of ML algorithms.

*Figure 3.3* gives us an insight into the various use cases of machine learning; here are some examples associated with each of these use cases:

- **Classification:** Being able to classify an image of Apple versus Banana

- **Prediction:** Being able to predict the sales of a particular product for the next quarter based on past data.
- **Forecasting:** Forecasting is similar to prediction but associated with actions like forecasting weather or natural calamities and many more.
- **Recommendation:** Providing recommendations to the user on an e-commerce website based on previous usage patterns.
- **Association:** Associating another product that might go along with a product that the user has already picked.
- **Computer vision:** Computer vision is predominantly related to image processing, where we can read from images and infer information. For example, a machine being able to read the handwriting from an image.
- **Speech recognition:** Voice to text conversion like chatbots.
- **Language:** Translation of one language to another uses natural language processing methods.
- **Dynamic adoption:** Automation is an example of dynamic adoption where machine learning can help efficiently automate processes.
- **Audio recognition:** Recognising gun-shots fired in a radius of 4 km.

Here is an image that gives us a sneak peek into the various applications of machine learning:



*Figure 3.3: Applications of machine learning*

## Exploratory data analysis

**Exploratory Data Analysis (EDA)** is an essential part of machine learning. It helps us to gain overall insights into the dataset that we are working with. EDA involves cleaning the data, visualizing the data, and getting to know the essential features that dictate the behavior of the models.

Here are the different operations involved in EDA:

- **Summary statistics:** Statistics such as Mean and Median values can be used to understand how the data is distributed mathematically. Statistics such as Minimum and Maximum values will help you discover the range of your data.

You can understand the trends amongst the various attributes/features using the summary results.

- **Plotting:** Provides a good overview of the data distribution. It gives us a visual medium to understand more about the dataset; it's the shape and the relationship between the various attributes present in the dataset.
- **Correlation analysis:** Correlation analysis is one of the most critical steps in EDA. Certain features are predictive of other values in a data set, and correlation analysis helps us discover those relationships. It is instrumental in eliminating unnecessary features in the beginning.

Not all conclusions can be drawn based on correlation analysis. Correlation analysis needs to be controlled in a fashion such that human intervention should be necessary to draw the final inferences using correlation analysis. It is also known as **controlled correlation** and is being widely practiced in the Machine Learning industry.

- **Feature importance:** Feature importance is a step that will decide the accuracy and effectiveness of your machine learning model. Based on your dataset, you may have to reduce/increase the number of features to model your data effectively.

Using various methods, you can determine as to which features will impact you the most during data analysis and retain those while eliminating the ones which are unnecessary or redundant.

## Why feature selection/generation?

Feature selection/generation is applicable in situations when data has a very high dimensionality or when it has a very low dimensionality.

In the case of data with high dimensionality, models fail to function effectively because:

- Training time increases exponentially with the increase in the number of features.
- With an increasing number of features, models have an increased risk of overfitting.

Feature selection helps avoid these problems by reducing the number of features and selecting the ones that are important for modeling.

In case of data with lower dimensionality, models fail to function effectively because:

- There is a lack of features to understand the way the data is distributed
- Due to the inadequacy of information, there is an underfitting of the models.

Feature generation helps solve the above issues by generating new features from existing features, thereby giving more insight into how the data is distributed.

## Overfitting and underfitting

Overfitting and underfitting are common occurrences in machine learning. They are generally the reason why a machine learning model does not perform so well. In this section, we will explore these two occurrences and take a look at the desired output graph as well.

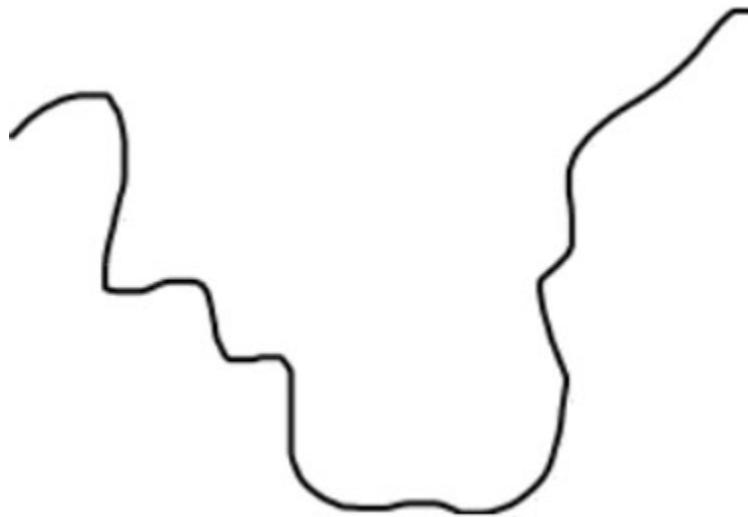
### Overfitting

Overfitting is a common phenomenon in machine learning when the data dimensions are enormous. As the term indicates, overfitting happens when the machine learning algorithm can predict the output variable with very high accuracy for the training dataset.

Overfitting could also mean that the devised model is highly complex and needs to undergo more validation tests using cross-validation data to

prevent overfitting. Feature selection also helps to avoid overfitting as lesser dimensions would mean that your model is not too complicated.

Here is an example of how an overfitted dataset curve would look like:



*Figure 3.4: An overfitted machine learning curve*

[Figure 3.4](#) shows us a machine learning model that has overfitted for the given dataset; that is, it predicts all the output variables correctly, and all the observations lie on the graph. It also shows us that in the case of overfitting, one cannot generalize the model for data other than the given training data, which defeats the entire purpose of creating ML models.

Overfitting may seem like a positive sign while modeling on your training data, and that is why it is essential to perform rigorous cross-validation tests before declaring that your machine learning model is robust and generic.

## Underfitting

Underfitting implies that your model did not fare well on the training dataset itself and hence, is quite simple. The main goal of machine learning is to capture the trends or patterns in the given data, but an underfitted model fails to achieve that goal.

Underfitting can also be a result of insufficient data to be able to model; that is, there is not enough data present to draw any insights from it. In this case, the only resolution is to wait for more data to be collected. The other reason for underfitting could be the use of a straightforward model.

Here is an example of how to underfit dataset curve would look like:



*Figure 3.5: An underfit machine learning model*

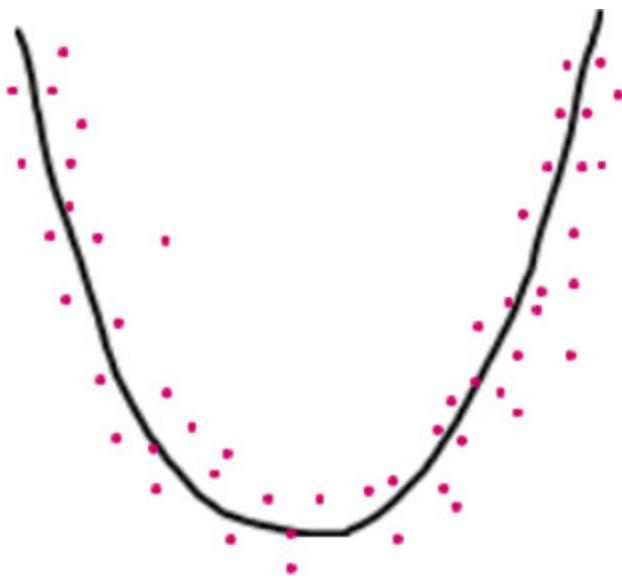
[Figure 3.5](#) shows us an example of a dataset that has been under fitted. You can see that the algorithm uses a linear model, that is, a straight line, which is unable to cover the area where the data is scattered.

Underfitting is not a very common problem these days since there are a lot of in-built machine learning algorithm libraries present in python, which you can use to experiment on your data.

## Desired output

The desired output or machine learning model is the one that fits the training data with a high accuracy leaving room for some outliers as well. It would mean that the model is generic and can also be extended to new and unknown data.

Here is an example of how the desired output graph of a model could look like:



*Figure 3.6: A model close to the desired output*

[Figure 3.6](#) depicts a model that fits the existing data as much as possible and is not focused on fitting every observation.

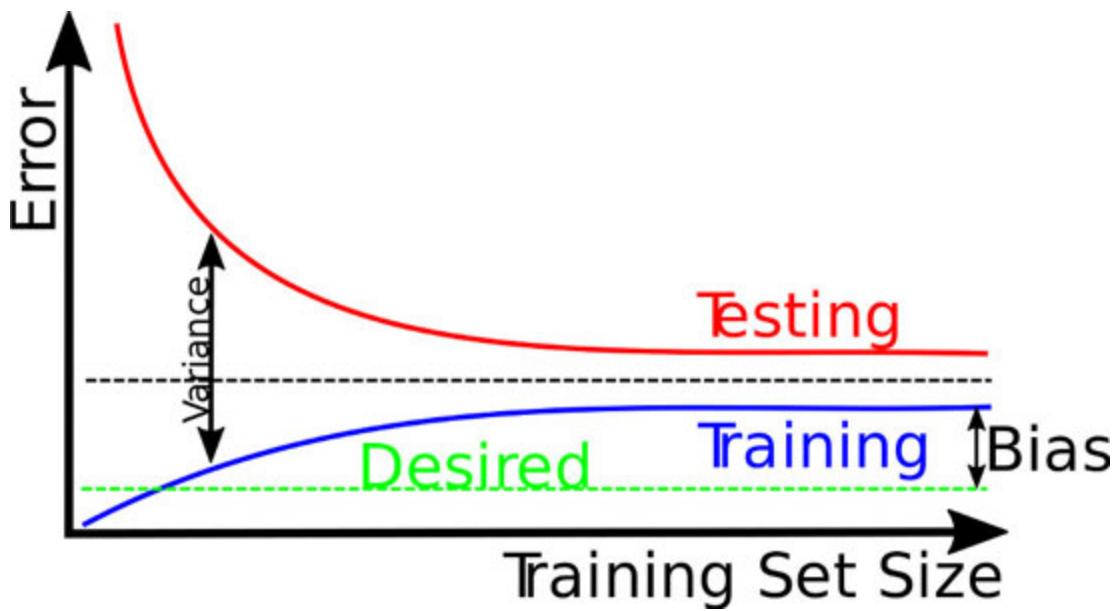
These kinds of models are ideal and can be extended to more extensive readings that can be fed into the model to predict variables for unknown user data.

## Variance and Bias

Every machine learning model has Variance errors and Bias errors. These are some of the errors that can be reduced by understanding what variance is and what is bias in a machine learning context.

By analyzing your training and test data errors for variance and bias, you can improve the accuracy of your model and positively influence your machine learning algorithms.

Here is an image that depicts the variance and bias for a randomly chosen ML model:



*Figure 3.7: Variance and Bias for a model*

As you can see in [Figure 3.7](#), as the training set size increases, the variance becomes lesser, whereas the bias is not affected by it. The above graph is a way to visualize the training and testing error with an increase in the training data size.

## Variance

Every model is trained on a given set of data, also known as training data. Variance refers to the variability of model production for the given dataset. It is the estimate of the change in our target function if the training data is changed. A target function is created for a given dataset (training data) based on the machine learning model built for the data; therefore, the algorithm will have some variance.

The ideal scenario will be if the variance is not too high since that would imply that the model is more generic and not very specific to the given dataset. It would mean that the algorithm can create a consistent mapping between the input and output variables.

## Bias

Bias leads to high error in both the training and test datasets. Bias tries to simplify the model as much as it can, which leads to lower predictive

performance on relatively complex problems.

Parametric functions generally tend to have high bias and are less flexible when it comes to learning.

Lesser bias implies that there have been fewer assumptions made about the target function itself, whereas a higher bias means that there have been more assumptions made about the target function. Linear and logistic regression models have shown to display higher bias values in general.

**NOTE:** For every model, there is a trade-off between minimizing bias and variance.

**Parametric functions are a way to represent functions that have one-dimensional input and a multi-dimensional output.**

## Predictive modeling

In this section, we will give you a brief glimpse into the process of predictive modeling, which is a critical step in machine learning. Predictive modeling is the process of building machine learning models to predict specific outcomes based on your dataset.

Predictive modeling uses two significant concepts to predict/forecast outcomes:

- **Data mining:** Data mining is the process of analyzing large databases to obtain new information from them. It essentially means that data mining is used to derive patterns from raw data that is given to us.

Data mining is a critical process considering the amount of data that is being collected in today's world is massive and is not used directly.

- **Probability:** Probability, as we all know, is the process of quantifying the likelihood of an event happening. Probability always results in a number between 0 and 1.

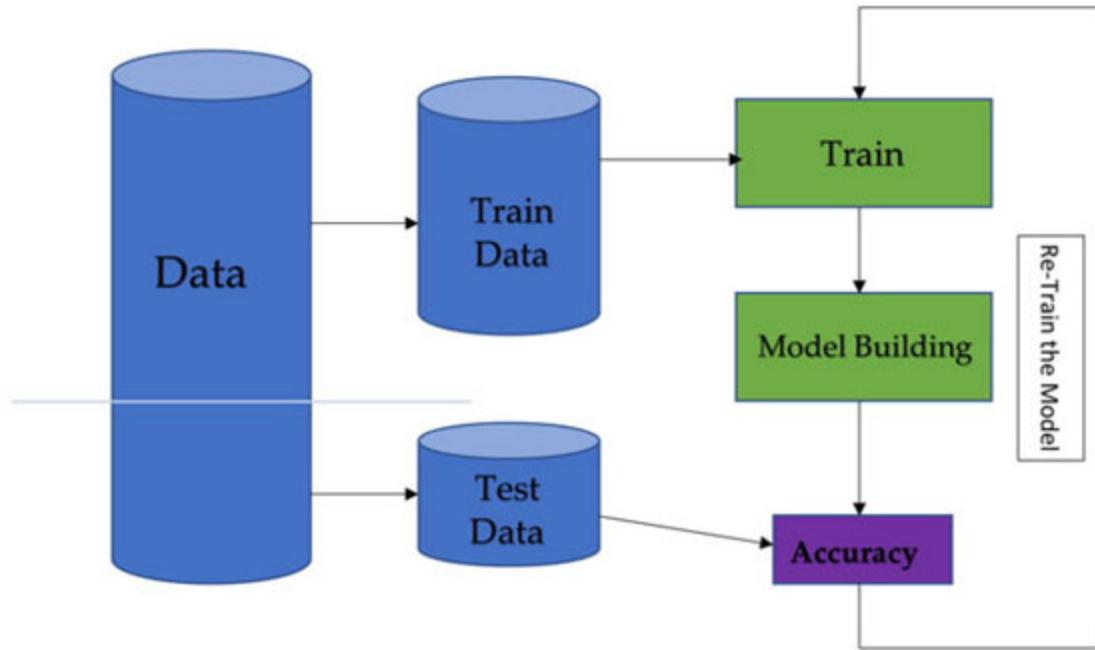
In this case, the mathematics behind probability helps us identify these patterns and their corresponding outcomes with specific probabilities.

Most of the machine learning algorithms predict the outcomes with a certain amount of probability associated with them, which also gives us an idea of how the model is making decisions.

Predictive modeling uses the above two core concepts to build statistical models and predict outcomes with the help of the statistical machine learning models created during the process.

**Regularization is a method that can be used to improve your predictive models.**

Here is an image that gives us an overall idea of what a predictive modeling cycle looks like:



*Figure 3.8: Predictive modeling cycle*

Predictive modeling is a continuous cycle unless the predictions are accurate to our liking.

The predictive modeling cycle is represented in [Figure 3.8](#). In this section, we will discuss the various steps involved in predictive modeling.

## Steps that take place in predictive modeling

### Step 1 – Collecting data

Data collection is the first and foremost step in predictive modeling. You need to define your problem statement and ensure that you have sufficient

data to predict your desired outcome. Many of the models end up predicting weakly due to insufficient data.

Data collection can refer to collecting any type of data (you can refer to different types from the **Types of Data** section). It can be your company data, open-source data, and so on, but there needs to be a sufficient amount of data to perform analysis.

Data can be labeled or unlabelled based on which the learning methods will differ. We will learn more about this in our next chapter on Supervised and *Unsupervised Learning*.

Data can also be in a dirty format, that is, not in a very row and column aligned format. In that case, data needs to be cleaned, and unwanted values should be removed. Initial steps of the Exploratory Data Analysis need to be performed such that the redundant features are removed. Relevant feature/selection generation has happened, and the data is in a machine-readable format.

Now the data is ready to be taken to *Step 2*.

## **Step 2 – Splitting the data into training and test data**

The given dataset is split into training and test dataset for model building and verification purposes. We can also split the data into training, cross-validation data, and test datasets.

### **What is the training data?**

Training data is a significant portion of our dataset using which the machine learning models are built. The training data is used to derive insights and data patterns that will help us predict specific outcomes.

Training data generally comprises of 70% of more of the given dataset.

### **What is cross-validation data?**

Cross-validation data is a small sample of data taken from our dataset to validate the machine learning model that has been designed using the training data. Cross-validation is an essential step while building a model since it tells you as to how the results of your analysis will generalize at a larger scale.

Cross-validation is also known as out of sample testing, and the data generally comprises of 10% of the dataset.

### **What is the test data?**

Test data is the data that is used to determine the robustness of your model. In the test data, the variable to be predicted is hidden, and the data is passed to our ML model, which then generates the required predictions. The generated predictions are compared against the actual labels that were associated with the data.

Test data generally comprises 30% of the original dataset, but the proportion can vary based on the different kinds of tests that you would like to perform.

### **Step 3 – Creating a machine learning model**

Based on the type of data, labeled or unlabelled, and based on the types of labeling (binary, multiclass, continuous), we can create machine learning models using several available algorithms in the form of Python libraries and functions.

The models are created using the training data provided and are validated using the test data. Several parameters define the accuracy of the machine learning model, including the percentage of correctly predicted outcomes on the test data.

The accuracy metrics are used to understand as to how good the model is and if the model needs to be re-trained again by considering a different set of features/attributes, and so on.

### **Key elements of machine learning**

There are three key elements of machine learning, each of them equally important to get accurate outcomes from our machine learning model. Here are the three key elements one must consider while building a machine learning model.

### **Representation**

How can we represent the data in the best possible format? There are several ways to represent the collected information, in the form of decision trees, in the form of neural networks, support vector machines, another kind of model, and more.

## **Evaluation**

The hypotheses to calculate the performance of your model. You can use several accuracy parameters like accuracy, precision and recall, f1-score, probabilities, entropy, **RSS (residual sum of squares)**, and so on.

## **Optimization**

Once the machine learning model is built, based on the evaluation, it needs to be optimized to get the best results. There are several types of optimizations, like combinatorial optimization, convex optimization, and constrained optimization. You can use any one of these techniques based on your machine learning model.

## **Machine learning in practice**

In this section, we will explore a case study that will take us through the process of building a Hotel Recommendation Engine.

## **Travel website case study**

Several online travel agencies have gained massive success in recent times due to the additional options on their website to book hotels, cabs, and many more, apart from just booking flights. These online agencies help you build a customized itinerary for your entire trip, starting from flight booking to hotel booking and local transportation.

In this post, we will take you through the steps of creating a recommendation engine and how machine learning plays a vital role while recommending hotels to visitors on an online platform.

We will not write the python code but just walk through the process of how a data science problem needs to be approached.

## Data

As we saw in the above section, it is essential to collect and represent data in the right format. The data that you need to collect needs to have exhaustive information about user behavior. Here is an example of a dataset that can be found on Kaggle (dataset link: <https://www.kaggle.com/c/expedia-hotel-recommendations/data>).

The above dataset gives us an insight into the users that visit the Expedia website for travel bookings.

Once you take a look at the training data (the file named train.csv), you should be able to see that most of the fields are numeric. Since the number of records (observations) is high in number, you can sample the observations, that is, select a smaller chunk to perform analysis depending on your system's storage and memory space.

## Exploratory data analysis

As we all know that **Exploratory Data Analysis (EDA)** is one of the most critical steps in machine learning. If you observe the data, you will be able to see that the variable to be predicted is named `hotel_cluster`.

You can plot the distribution of the `hotel_cluster` variable and observe if the data is skewed. It is essential because this will help you understand if your training model will be biased or not. You can use the `matplotlib` or the `seaborn` libraries to plot the distribution.

## Feature engineering

The next step is feature engineering, which includes converting the features that are in other formats to a numerical format. The date columns in this dataset need to be converted into another format as they cannot be used directly.

The next step is feature selection. You can conduct **Principal Component Analysis (PCA)** on the dataset and find out the features that are of importance. We can also perform correlation analysis and eliminate the highly correlated features.

## Building the machine learning model

At this point, you have all the features you need to build the machine learning model. You can use the in-built Python libraries to train the data using several algorithms. You can try several algorithms like Naive-Bayes, Multi-class Logistic Regression, SVM, Random Forest, and many more, to finalize the model that fits best with your data.

In this dataset, the training data and test data have already been separated and are into different CSV files, so you do not have to split the data into training and test datasets.

To finalize the model that works the best, you can calculate several parameters like accuracy, precision, recall, f1-score, and so on, and build a confusion matrix to get the numbers for your model.

## Conclusion

The above process is quite generic and can be applied to all the datasets that need to be analyzed. Machine learning in practice, is a series of feature engineering, building the model, analyzing the accuracy, and then re-iterating the process. The best machine learning models are the ones that use the best features available to build the model.

## A quick research quiz

1. What is PCA? Is it a deterministic algorithm?
2. How does deep learning differ from conventional machine learning?
3. What is the role of Hadoop and Hive in data analysis?
4. What are Type-1 and Type-2 errors?
5. What are the scatter plots used for machine learning?

## CHAPTER 4

# Supervised and Unsupervised Learning in Python

### Introduction

Machine learning encompasses several forms of learning based on the data that is being analyzed and the outcome that the user wants from the data. Supervised and unsupervised learning are two different learning methodologies to learn from the data provided to us. As the term indicates, supervised learning needs a supervisor or a guide from where it can learn.

On the contrary, unsupervised learning is performed in situations where there is no supervisor or guiding information available.

Python has been one of the pioneers in the field of machine learning as it offers several built-in libraries that can make the life of a machine learning programmer quite easy.

The majority of the industry prefers to use supervised learning as it is closer to giving the desired information that we need. It is easier to infer patterns and outcomes from a supervised learning dataset as compared to a dataset where unsupervised learning needs to be performed.

In this chapter, you will learn more about the supervised and unsupervised learning techniques and their use cases in the industry.

### Structure

- Difference between supervised and unsupervised learning
- Role of Python in machine learning
- Deep-dive into supervised learning with examples
- Deep-dive into unsupervised learning with examples
- Conclusion

- Quiz

## Objectives

The primary objective of this chapter is to understand the difference between supervised and unsupervised learning and their usage in different contexts. In a scenario where the reader has to apply either of the techniques, this chapter will also give them an insight into what are the different sub-techniques that they can use for each of these learning types.

## Difference between supervised learning and unsupervised learning

The main difference between supervised learning and unsupervised learning is that, in the case of supervised learning, we have prior knowledge about the behavior of the dataset.

In the case of unsupervised learning, there is no prior data about the labels that define the behavior of the data, and new labels/classes are created for the data from which we can learn.

## Supervised learning

Supervised learning is a form of machine learning where the dataset is labeled; that is, each observation has a specific class associated with it. The goal of supervised learning is to establish a function, which can describe the relationship between the given inputs (features present in our dataset) and the output variable in the best possible manner.

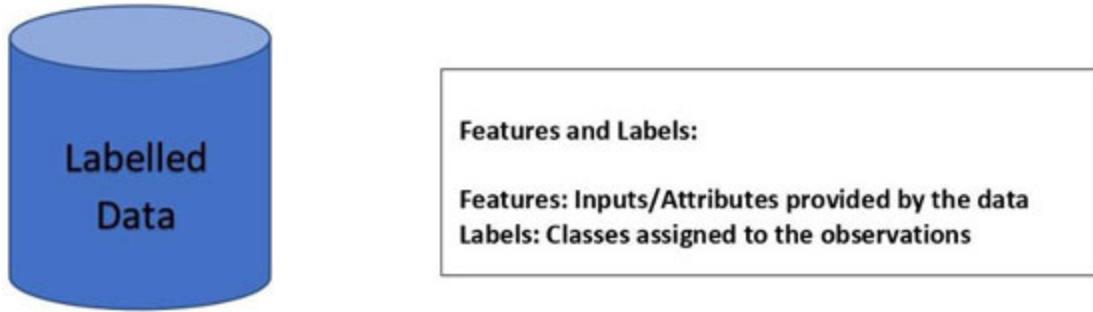
Typically, if  $x$  is our input variable and  $y$  is our output variable, a supervised learning equation would look something like this:

$$y = f(x)$$

Where  $f()$  is the function that gives us the relationship between the input variable/variables ( $x$ ) and the output variable ( $y$ ).

As you can see in [Figure 4.1](#), supervised learning algorithms work on data that is already labeled. Humans/programs label the data before analyzing it. In this case, we can directly learn as to how the features map to the labels.

Here is an image that describes supervised learning:



*Figure 4.1: Supervised learning with labeled data*

## Unsupervised learning

Unsupervised learning is a form of machine learning where the dataset is unlabelled; that is, there are no classes associated with the observations initially. The goal of unsupervised learning is to explore the possible relationships within the given observations and group them into similar clusters.

Typically, if  $x$  is our input, we use several exploratory algorithms to figure out a function  $f()$ , which can create clusters and map these inputs to an appropriate cluster.

As you can see in [Figure 4.2](#), in an unsupervised learning model, we do not know what the label is. The algorithms are used to first group similar observations into one cluster and post that, we learn how the clusters map to different labels.

Here is an image that describes unsupervised learning:



*Figure 4.2: Unsupervised learning with unlabelled data*

In the next section, we will talk about why Python is the most suitable language when it comes to machine learning and what does it have in store for all the machine learning enthusiasts.

## The role of Python in machine learning

Python is one of the most versatile languages that we have in the current industry. With its simple constructs and interpretable nature, Python has become one of the most widely used languages for modern-day programmers.

But why Python for **Artificial Intelligence (AI)**? AI projects vary in nature from the traditional software projects because they need in-depth research implemented most seamlessly. Python gives us access to frameworks and machine learning libraries that can be directly used in our code, abstracting the user from complex mathematics.

For supervised and unsupervised machine learning, Python offers several libraries like PyTorch, Tensorflow, Keras, and Scikit-Learn. Libraries like Numpy, Pandas, SciPy, Seaborn, Matplotlib, and many more, help work with the data analysis and visualization, and we have OpenCV and NLTK for Computer Vision and NLP respectively.

Python is one of the most natural choices for developers as it is platform-independent as well. To work with emerging technology like machine learning, it is essential to choose platform-independent languages as they don't bind you to certain Operating Systems.

Overall, Python offers a great community for machine learning developers and is an asset for the programmers when it comes to debugging issues or seeking libraries to perform certain operations.

In the upcoming sections, we will deep-dive into both supervised and unsupervised learning with real-time examples.

## Deep-dive into supervised Learning with examples

The two main kinds of supervised learning are classification and regression. As we have already seen in the previous chapter, classification is involved when the variable to be predicted a categorical variable; that is, it has a

defined set of values. Regression, on the other hand, is the algorithm where the variable to be predicted is a continuous set of values.

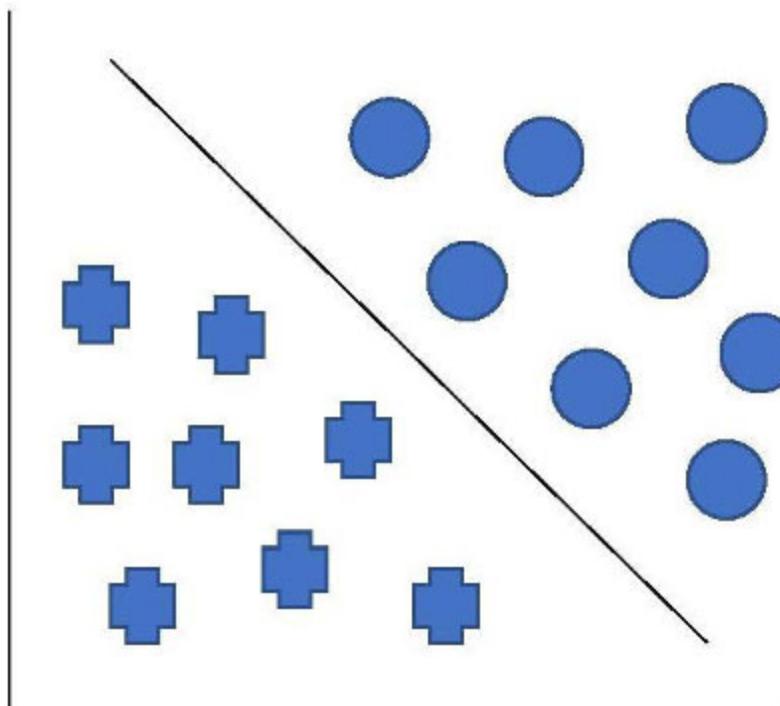
## Classification

Classification in supervised learning is when you can assign specific labels to your observations. An example of classification would be to predict if the symptoms of a person will result in breast cancer or not, based on previous data collected on symptoms and their expected output(Yes/No).

Figure 4.2 is a visual example of how classification can classify two separate classes using a linear function. In this figure, we have two classes, one of which is represented by the Solid Plus (+) and the other is represented by a circle (o).

The two classes are separated by a linear function, that is, a line which states that some of the observations fall under the (+) category, and some of the observations fall under the (o) category, and each of them has certain characteristics.

Using this labeling, we can predict the future labels for unknown data by assessing the features and applying the following linear function to the features to find the relationship between the labels and the given data:



*Figure 4.3: An example of classification*

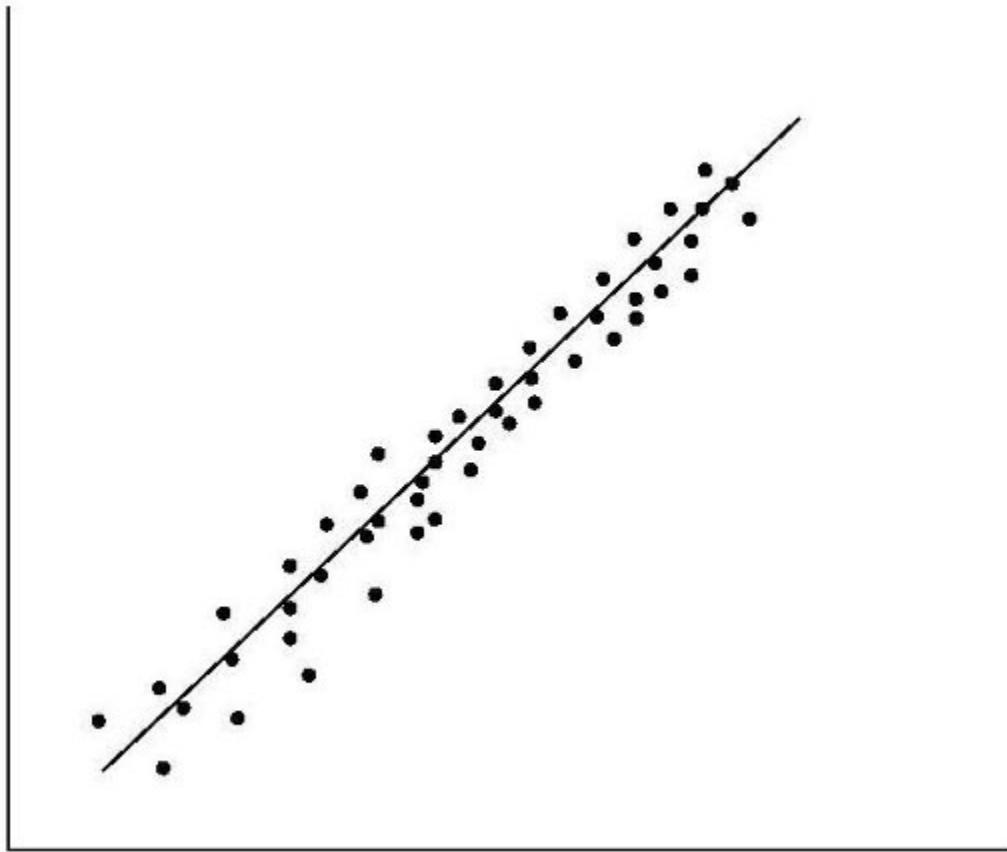
In the next section, we will get a glimpse into what regression methodology looks like.

## Regression

Regression is an algorithm used in supervised learning when your labels are continuous variables, also known as real-valued labels. An example of regression would be, when you may need to predict the percent of bad cells (cancerous cells) in a person's body by analyzing a scanned image. In this case, the percentage could be any real value, like 85.6%.

Figure 4.4 is an example of how regression can be used to predict an outcome using a linear line (linear regression). In this figure, we have values that are spread across, and the line is a function that is trying to fit these output values for the given input features/attributes.

The linear function that is shown in the figure is a function that describes the relationship between the inputs and the outputs (which are continuous). Using this labeling, we can predict the outcome of the unknown data observations that are given to our model in real-time. Based on the accuracy of the function chosen (which is generally determined by several parameters, post which the most accurate model is chosen), the model will be able to predict real-valued outputs for the unknown inputs most accurately:



*Figure 4.4: An example of regression*

In the upcoming section, we will look at examples of both supervised learning and unsupervised learning methodologies to understand their differences and implementations in real-time.

## **Supervised learning example 1– Classification of fruits**

In this section, we will take you through an example that will help you understand supervised learning clearly. It is a typical example and is often used for supervised classification.

### **Problem statement**

We will classify a basket of fruits using supervised learning.

### **Data required**

To classify a basket of fruits, we need first to create a database that will have fruits and their features:

Colour	Shape	Fruit
Yellow	Curved	Banana
Green	Round	Apple
Red	Round	Apple
Black	Tiny	Grapes
Green	Tiny	Grapes

*Table 4.1: Data for supervised learning*

The above table is an example of a dataset that contains some information about the fruits. There are three columns: **Colour**, **Shape**, and **Fruit**. The **Colour** and **Shape** columns are used as features, whereas the **Fruit** column is the label in this case.

The above table, in simple terms, states the following:

- An object in the basket which is curved in shape and is **Yellow** is labeled as a Banana.
- An object in the basket which is round in shape and **Green** or **Red** will be classified as Apple.
- An object that is tiny and Black or **Green** will be classified as Grapes.

**The table defined above is not a real table but has been defined just, for example, purposes.**

## Features

The features that we will consider for this example are as follows:

- Colour of the fruit
- The shape of the fruit

We can also have more general features describing a fruit like it's "Sweetness Level," "Origin of the fruit," and so on, but in this context, we can assume that we have an image of fruit with us and based on the shape

and color data that has been extracted from the image, we will classify the fruit.

## Supervised learning

Our upcoming image is a clear example of how a labeled dataset would look like, with several features, each of which is labeled. In the following image, you can see a mix of fruits, each of which is different but can be identified (labeled) based on their color, structure, and knowledge of their taste.

Here is an image of mixed fruits, all of which can be distinctly identified:



*Figure 4.5: Mix of fruits – labeled data*

Now that we have a dataset that is labeled - in our case, the fruits are known based on color and structure; we can perform supervised learning algorithms on this dataset.

The supervised learning algorithm will use the given dataset and classify the fruits present in the basket into one of the labels that are present in the dataset.

In this case, the machine learns from the data provided by us and then classifies the fruits present in the basket. Therefore, in our case, the fruit basket is the test dataset.

### Accuracy and next steps

Based on the number of correct predictions of the fruits present in the fruit basket, we will be able to understand the accuracy of our supervised algorithm. We can re-train our model with a different algorithm or request for more training data in case the accuracy is not up to the mark.

## Supervised learning example 2 – Predicting quarterly attrition rate of women in an organization

In this section, we will look at a supervised learning example, where we will try to look at the different contributing factors that can predict the quarterly attrition rate of women in an organization.

### Problem statement

We will try to predict the quarterly attrition rate of women in an organization.

### Data required

To predict the quarterly attrition rate of women in an organization, we will create a database that will have features and a label that will state if the churn was **Yes** or **No**:

Name of the Employee	Age of the Employee	Location of the Employee	Experience	Role	Churn
----------------------	---------------------	--------------------------	------------	------	-------

A	32	Bangalore	7	Developer	Yes
B	28	Bangalore	4	Admin	No
C	30	Mumbai	5	Developer	No
D	45	Pune	12	Director	Yes
E	38	Mumbai	11	Developer	Yes

**Table 4.2:** Table representing the features and the quarterly attrition of female employees in an organization

## Features

The following are our features based on which we can perform supervised learning:

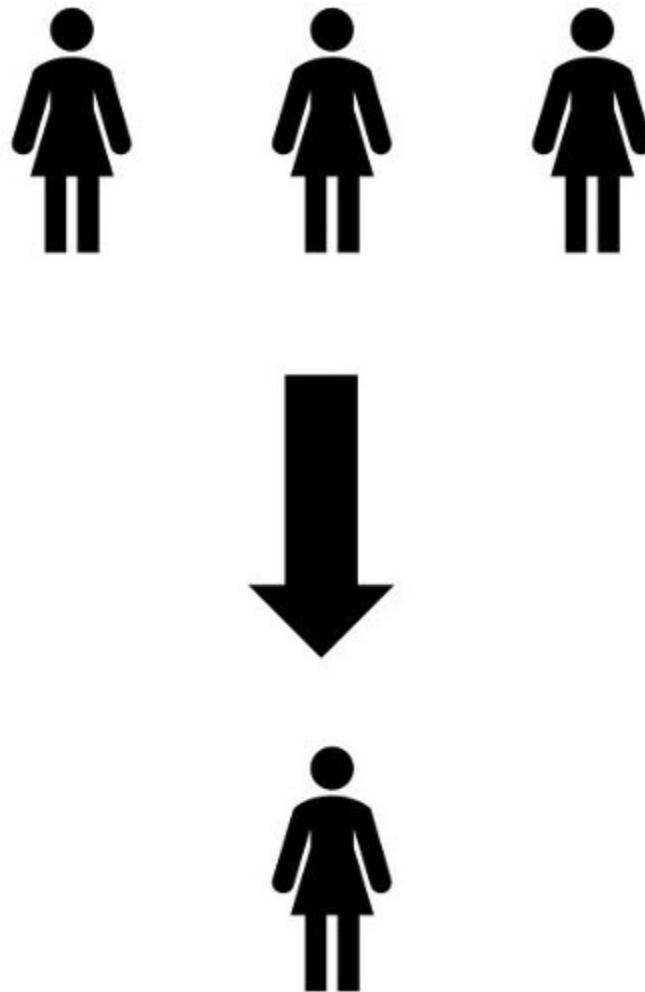
- Demographic details (Name, Age, Location)
- Number of years of experience
- Role in the organization

We can also have more general features that are factors for an employee's attrition rate like "Salary Earned," "Number of promotions received in the past year," "Remote workers - Yes/No," and many more, but in this context, we can assume that we have these couple of features given to us, based on which we will be able to predict for new female employees if they will churn in this quarter or not.

## Supervised learning

Using the above information that we have about the problem, we will look at how supervised learning is performed in the following section.

Here is an image that visually describes the attrition rate of women in an organization:



*Figure 4.6: Women's Attrition rate in an organization*

Now that we have a dataset that is labeled - in our case, the churn(Yes/No) possibility of women is known to us along with other features like demographic details, number of years of experience, and role in the organization.

The supervised learning algorithm will use the given dataset and predict if a woman will leave the organization or not based on the given features.

In this case, the machine learns from the data provided by us and then predicts the churn(Yes/No) for any test data, which is the data provided by the same organization when they want to understand how their quarterly attrition rates for women will look like for a particular quarter.

## Accuracy and next steps

Based on the number of correct predictions of the attrition rate for a new quarter, we will be able to understand the accuracy of our supervised algorithm. We can re-train our model with a different algorithm or request for more training data in case the accuracy is not up to the mark.

## Deep-dive into unsupervised Learning with examples

While supervised learning is one of the most commonly used learning methodologies, there is increasingly more focus given to unsupervised learning since a vast amount of unlabelled data is difficult to segregate and label manually or by machines.

Unsupervised algorithms are being researched continuously and improvised as they have immense potential to change the way machine learning is being used in the industry. In this section, we will be looking at four unsupervised learning techniques and how they work.

### Clustering

Clustering, as the name suggests, is a grouping of points. Points having similar features get grouped into one cluster. It is based on the fact that data points having similar properties should be in one group and should have different properties from data points in other groups.

The similarity between the data points is calculated by using a distance metric based on a type of feature variable set.

Clustering is a simple algorithm that kisses valuable insights into our data. It has several applications in the industry, for example:

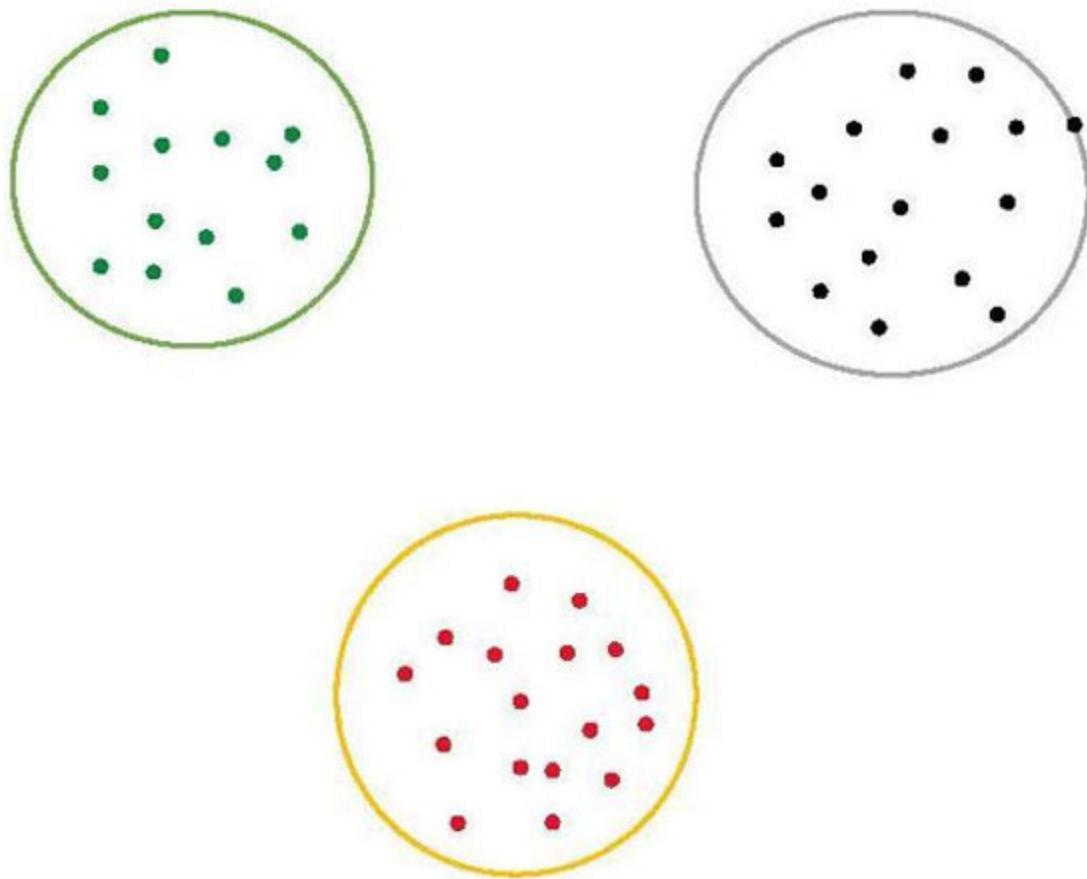
- Biology, a grouping of species
- Medical imaging
- Recommendation systems

Here are some of the most famous clustering algorithms used currently in the industry:

- K-means
- Agglomerative hierarchical clustering

- DBSCAN

The following is an image that visually represents clustering. The three different clusters represented in the image correspond to three different colors - yellow, green, and blue, each of which is symbolic to a label in the dataset. Here is the image representation:



*Figure 4.7: Clustering data into three different groups*

Our next section will dive deeper into Autoencoders, which are very helpful when it comes to handling large amounts of data.

## Autoencoders

Autoencoders are used in machine learning when the data that we have is too large to handle.

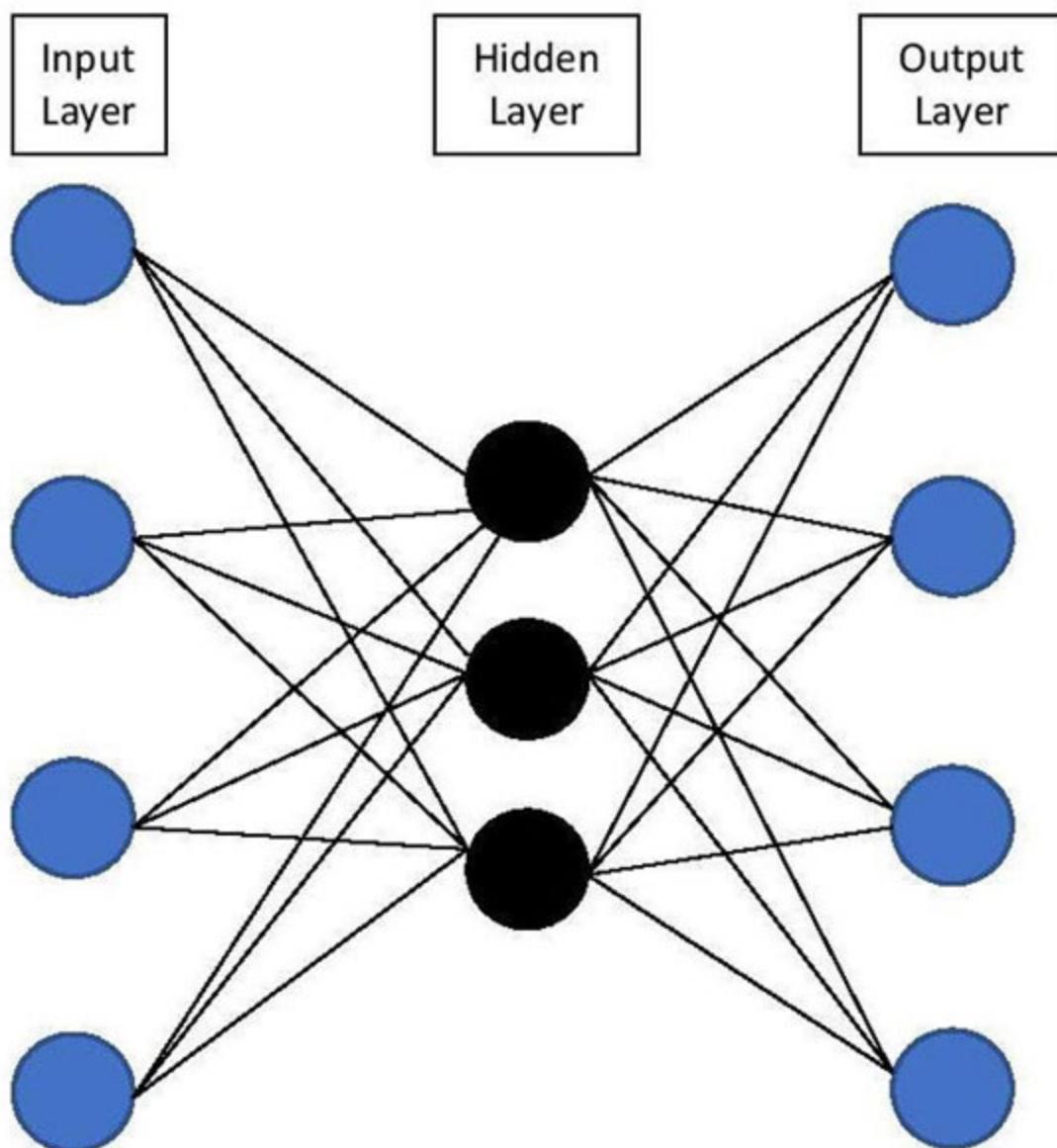
For example, imagine a situation where you need to store one trillion fingerprints into the database so that you can access those fingerprints at a

later stage for various authentication purposes.

It is where autoencoders come into the picture. They can encode the data and store it into the database by reducing the size of the data. They encode the data such that the data represent the same information, but the size is reduced.

It requires training a neural network to predict its output. Generally, the middle layer of such a network does not have too many layers; in fact, the neural network is dense at the input stage and the output stage.

The idea is to train our neural network to learn the compressed features:



*Figure 4.8: Autoencoders*

## Features separation techniques

Since features are the most critical elements in machine learning, we need to be able to break down the features and look at the features independently.

The two most popular techniques for feature separation are:

- **PCA (Principal Component Analysis):** It is a statistical procedure that finds linear combinations that hold the most variance and information on your data.
- **SVD (Singular Value Decomposition):** It factorizes your data into a product of much smaller matrices.

The techniques mentioned above are used to isolate and analyze the feature vectors. Given an opportunity to analyze these vectors independently, we can choose which one's are the best for our model. Dimensionality reduction is a sort of unsupervised activity since there isn't any pre-existing supervision that goes into this process.

## Expectation maximization algorithms

The expectation-maximization algorithm can be used for those variables which are not directly available to us but are derived from the existing features/variables. The algorithm itself is used to find (local) maximum likelihood parameters of a statistical model where the equations cannot be solved directly.

The algorithms are described in the following steps:

1. Choose a set of starting parameters for a given set of incomplete data
2. Expectations step (E-step): Using the available observed data, guess the values of the missing data of the dataset
3. Maximization step(M-step): Maximise the data guessed in the second step and update the values of the parameters.
4. Repeat Step 2 and Step 3 until the convergence of the value.

The essence of this algorithm is to use the observations' available values to estimate the missing data and then update the parameter values using the estimated data.

## Unsupervised learning example 1 – Dividing pictures of people based on their facial pattern

In this section, we will look at an example of unsupervised learning, which involves classifying or dividing the various pictures of people based on their facial patterns.

### Problem statement

We will try to classify the pictures of individuals into the same pile without having information on who is on which picture.

### Data required

Our data will be a dataset of pictures which will have several pictures of several individuals. The pictures can be in JPG or PNG format and will be converted to a matrix format to use it for machine learning. It generally means converting the pixels of the picture into a matrix format.

Our data is essentially a matrix of pixels of different pictures.

### Features

Our features based on which we can perform unsupervised learning will be:

- Pixel values of the picture
- Colour schema of the picture (which pixel corresponds to which color)

Our significant feature here will be the facial features, which will be used for categorizing the pictures of the same individuals together and differentiating it from other individuals. In general, other features like the 'colour of the clothes,' 'shape of the hand' and many more, can also be considered as features, but logically, they may not give us the differentiation that we want to classify them as similar to different.

To achieve the pixels and the color schemas, we will have to encode the images and reduce the original matrices to an encoded matrix to store them effectively. The several hidden features based on which the algorithm will be able to classify will be the outline of the face (in case of face recognition), the shape of the face, and other facial features.

## Unsupervised learning

The following image gives us a glimpse of how the facial features of an individual are captured:



*Figure 4.9: Classification of images based on facial features*

We have an unlabeled dataset; that is, we do not have labels on each picture. Our unsupervised algorithm will use the given dataset and cluster the given photos in different clusters, that is, put each individual's picture into their cluster.

As we have seen above, there are several clustering algorithms that we can use for this purpose, like K-Means, Fuzzy Clustering, and so on. These algorithms will create clusters based on matching features in a given dataset and club the ones with similar features together.

## Accuracy and next steps

Unsupervised algorithms let the machine learn from the given data and create classes of its own, which can be verified by us based on the number of correct groupings made by the algorithm. We can fine-tune the algorithm

by changing some of its parameters, use another algorithm, or get more data to increase the accuracy.

## Unsupervised learning example 2 – Creating groups for your customer based on customer usage patterns

In this section, we will look at an example of unsupervised learning, which involves creating groups for customers based on their usage patterns.

### Problem statement

We are a pizza company (ABC). We are collecting statistics on the buying patterns of the customer like how many pizzas do customers buy each day, which day of the week hits the maximum amount of profit, which toppings do customers prefer on their pizza etc.

Based on the above statistics, ABC wants to create some discount coupons to increase sales and attract more customers. We want to use unsupervised learning and create three groups of customers and create custom coupons for each group/segment.

### Data required

Our data is the data collected by our company ABC over some time about customer preferences. It has statistical information about the most preferred toppings, frequently ordered type of pizza (cheese burst, thin-crust, and so on), day of the week that sees the maximum amount of profit.

It is a matrix with rows and columns but no labels associated with them:

Preferred toppings	Customers' favourite pizza type	Average toppings per pizza	Average pizzas per week
Jalapeno	Cheese Burst	6	340
Olives	Thin Crust	8	560
Pepperoni	Cheese Burst	4	230

*Table 4.3: Store-wise data collected for pizzas*

## Features

Our features based on which we can perform unsupervised learning will be:

- Preferred toppings
- Customer's favourite pizza type
- Average toppings per pizza
- Average pizzas per week

In this case, we can even have more features based on the amount of data collected in that store. As we have two numerical features already available in this example (Average toppings per pizza, Average pizzas per week), we would use these for demonstrating how the unsupervised algorithm will work.

## Unsupervised learning

We have an unlabelled dataset; that is, we do not have groups pre-allocated to each observation. Our unsupervised algorithm will use the given dataset and cluster the given observations into different groups based on the given features and their values in the dataset:



*Figure 4.10: Creating discount coupons for pizza consumers*

## **How it works with K-Means**

Since the number of groups that we have chosen is 3, initially, we will choose any three data points from our observations, and we call them as centroids. Then, we assign the different data points to the centroids they are closest to.

Now that we have a group of observations clubbed together, we will re-calculate the centroid for this group. Once the new centroids are calculated, we will re-assign the points to the closest centroid around to create new groups.

The algorithm will repeat until the centroids converge, that is, the centroids and groups stop changing even if we repeat the steps. Once these three groups are finalized, we can create discount coupons for each group.

The above algorithm is completely unsupervised as there were no groups present in the beginning but have been created based on similarities in their characteristics.

## **Accuracy and next steps**

Unsupervised algorithms let the machine learn from the given data and create groups/classes of its own, which can be verified by us based on the number of correct groupings made by the algorithm. Although, in this case, we do not have any source of truth and, therefore, cannot determine the accuracy of these groups.

There is a concept of silhouette scores that help us validate if the points in the groups are closed or not. You can explore this concept in depth if unsupervised learning interests you.

## **Conclusion**

The various sections and practical examples in this chapter have given us an overall understanding of both supervised and unsupervised learning methodologies. This chapter should help you decide when to use supervised and when to use an unsupervised learning algorithm while working on a machine learning problem.

Both supervised, and unsupervised learning algorithms have their importance in the industry. Supervised learning is relatively easy to understand since labels can give you insights on how your data is categorized. However, in this book, we have focused on supervised learning techniques to get you started hands-on with machine learning.

Our next chapter will talk about the most basic technique in supervised learning - Linear Regression. Regression analysis is the base for several machine learning problems, and linear regression will give you a fundamental understanding of how a machine learning algorithm functions.

## A quick quiz

1. What does the xgboost algorithm do?
2. What are some big data frameworks used for machine learning?
3. What is the drawback of using K-Means for unsupervised learning?
4. An automated vehicle is an example of supervised learning or unsupervised learning?
5. What is Ensemble Learning?
6. What are the two most popular techniques for feature separation?
7. What is the primary difference between regression and classification?

# CHAPTER 5

## Linear Regression - A Hands-on Guide

### Introduction

Linear regression is used to find out the linear relationship between the target variable and the features in the dataset. The features in the dataset are also known as **independent variables**, and the target variable is also known as the **dependent variable**. There are two subtypes under Linear Regression, Simple Linear Regression, and Multiple Linear Regression.

If we have one independent variable and one target variable, then it's known as a single linear regression. If we have multiple independent variables and one target variable, then it's known as multiple linear regression.

Linear regression was the very first type of regression analysis that became quite popular when it came to practical applications. It is because linear models are easy to fit as compared to non-linear models. In this chapter, we will explore Linear Regression and its implementation in the industry.

### Structure

- What is linear regression
- Statistics in regression analysis
- Simple linear regression
- Case study simple linear regression - I
- Case study simple linear regression - II
- Multiple linear regression
- Case study multiple linear regression – I
- Case study multiple linear regression – II
- Conclusion
- Quiz

## Objectives

- Understand the concepts involved in linear regression—both simple and multiple linear regression
- Learn to develop an end-to-end solution for a linear regression problem using Python
- Learn different techniques to calculate the accuracy of a linear regression problem

## What is the linear regression?

The idea of regression is to determine two things:

- Do the dependent variable/variables predict the independent variable with the right amount of accuracy?
- Which of the dependent variables are best fitted to predict the independent variable?

The linear regression is a statistical model that follows the formula of a straight line:

$$Y = mX + c$$

$X$  is the dependent variable here, and  $Y$  is the independent variable. The variable  $m$  is the slope of the line, and  $c$  is the  $Y$ -intercept, and these two variables are used as variables used to optimize the linear relationship equation.

The same equation for multiple dependent variables would look like this:

$$Y = m(X_1 + X_2 + X_3 + X_4 + X_5 + \dots) + c$$

Where  $X_1, X_2, X_3, X_4, X_5$ , and so on are the independent variables/features present in the dataset.

## Statistics in regression analysis

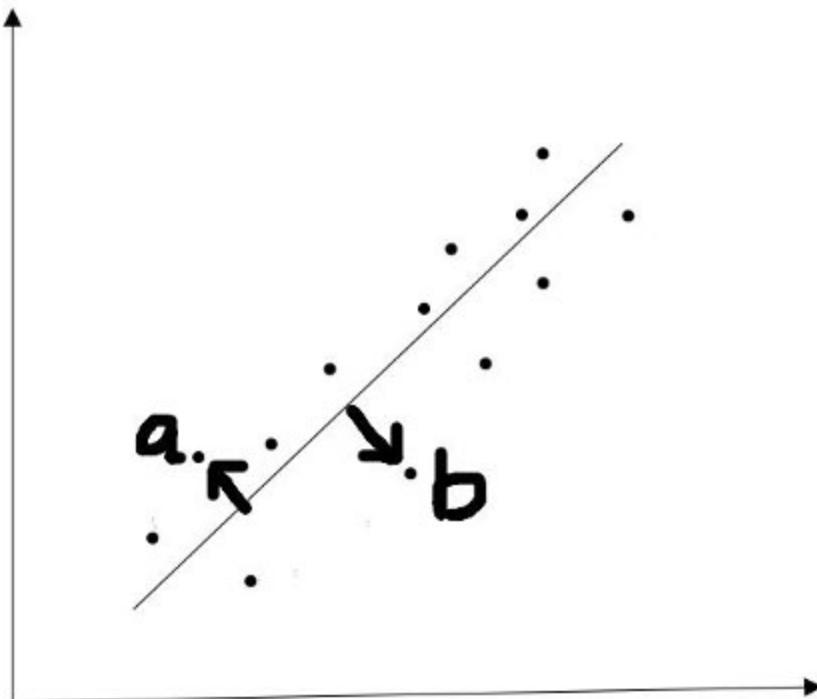
As we are starting with the fundamental machine learning algorithm, linear regression, it is highly essential to understand the statistics behind this algorithm. The statistics that we will be looking at in the upcoming session

form the basic principles of machine learning and are used across several algorithms to determine the efficiency of the algorithm.

## Residual sum of squares

Residual sum of squares, also called **Sum of Squared Residuals (SSR)** or **Sum of Squared Errors (SSE)** is the sum of squares of the residuals. Residuals are the measure of distance from the data points to the line, which can be harmful to positive depending on whether the data point lies above the line or below the line.

**Residual sum of squares (RSS)** are used to find out the extent of discrepancy between the data and the estimated model built to predict the result the amount of variation in the target variable that the model could not explain:



*Figure 5.1: Residual sum of squares for regression*

As you can see in the above [Figure 5.1](#), the point a is at a distance above the line, whereas point b is at a certain distance below the line. The distances, therefore, can be both negative and positive.

Our main aim in linear regression is to obtain a line to which the distance of our various data points is minimum. To get a positive value for the distance,

we will square the residuals/distances of the points from the line and minimize their sum, to get the best fit line for our dataset.

The equation for a linear regression line is:

$$Y = mX + c$$

$Y$  can be categorized as **Yoriginal** and **Ypredicted**. Yoriginal consists of the original labels assigned to our training dataset and Ypredicted is the set of labels predicted by our model for the training dataset:

$$RSS = \sum(Y_{predicted} - Y_{original})^2$$

RSS is the sum of the differences between the predicted target variable and the original target variable. The coefficients of our line (the slope  $m$  and intercept  $c$ ) are calculated in such a way that the RSS is minimum. The smaller the value of RSS, the better our model fits the dataset.

## R<sup>2</sup> score/ R<sup>2</sup> score

R<sup>2</sup> or R-Squared, also known as the coefficient of determination, is the percentage of variation in the target variable that can be explained by our regression model. Its formula is given by:

$$R^2 = \frac{\text{Variance explained by the model}}{\text{Total variance}}$$

The value of R<sup>2</sup> can be anything between 0 to, that is, 0% to 100%. Zero percent would mean that the model is unable to explain any variation in the target variable, and a hundred percent would mean that the model is successfully able to explain all the variation in the target variable.

Usually, the higher the R<sup>2</sup> score, the better your model's performance is, as compared to a model with a lower R<sup>2</sup> score for your dataset. But it is also essential to understand that not all models have a high R<sup>2</sup> score. Regression models with low R<sup>2</sup> scores can also be great models if the coefficients are statistically significant.

It is essential to understand that the R<sup>2</sup> score is not the only metric to determine the accuracy of your model; it is also necessary to check the residual plots.

## Root Mean Squared Error (RMSE)

Root Mean Squared Error (RMSE) represents the standard deviation of the residuals. RMSE tells you how concentrated or spreads out the data points are around the regression line. The formula for RMSE is given by:

$$RMSE = \sqrt{Mean(Y_{predicted} - Y_{original})^2}$$

A smaller value of RMSE would generally mean that the points are much closer to the predicted line than the larger values of RMSE. There could be some exceptions; for example, if the RMSE value is zero, then all the points lie on the predicted line, which can lead to overfitting the data for the given dataset.

## Simple linear regression

Simple linear regression is a form of linear regression where there is one dependent variable and one independent variable. It is the simplest form of a linear regression which is represented by the equation of a line as follows:

$$Y = mX + c$$

Where  $X$  is the independent variable and  $Y$  is the dependent variable.

Some of the other terminologies associated with  $X$  and  $Y$  are predictor variables, feature variables, and so on (for  $X$ ) and target variable, criterion variable, and so on (for  $Y$ ).

Regression analysis can be used for several purposes, most importantly, to understand the relationship between the dependent and the independent variables, predicting the value/score of a dependent variable, and many more.

Linear regression is aimed at finding the best fit line for all the data points that are present in the data set. The best-fitting line is known as the regression line and is calculated by minimizing the sum of squares of residuals.

We have two case studies that we will be working on for simple linear regression. Simple linear regression has been one of the earliest known machine learning algorithms and has been used extensively in the field of machine learning/artificial intelligence.

The case studies presented to us will give us a glimpse of the dataset that we are working with, possible **Exploratory Data Analysis (EDA)** methods

on the dataset, and building a model for the given dataset. The EDA steps are example steps, and you could do a deep-dive on each of these steps to understand more and learn more for yourself.

There are also some example plots for each of the datasets, but I highly encourage you to build various types of plots and understand the different trends and patterns between the two variables.

In the upcoming sections, we will look at case studies on simple linear regression to understand the real-time implementation in the industry.

## **Case study – I**

This case study will take us through the various steps that are involved in building a linear regression model. In this section, we will be focusing on datasets that have one independent variable and one dependent variable which fall under the category of simple linear regression.

We will go through the process of cleaning the data (if required), performing EDA followed by model building and checking for accuracy of the built model.

## **About the data**

The dataset that we are using for this case study has been collected from an open-source website which provides several types of data sample for regression analysis, time series analysis, and many more. The data describes the auto insurance claims files in Sweden in thousands of Swedish Kronor for geographical regions.

This dataset has two columns—the dependent variable is the 'Number of Claims' and the independent variable is the 'Total Payment,' that is, the total payment that is likely to be paid based on the number of claims files. Our problem statement involves predicting the total payment to be paid, given the number of claims in Sweden for data collected over some time.

## **Python code and step-by-step regression analysis**

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```

import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score,
accuracy_score

```

2. Read the data (which is stored in the CSV format) using `python-pandas` and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `AutoInsuranceSweden.csv`

```
df = pd.read_csv("Datasets/AutoInsuranceSweden.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame.

```
df.head()
```

Here is the output for `df.head()`:

	<b>Number of Claims</b>	<b>Total payment</b>
<b>0</b>	108	392.5
<b>1</b>	19	46.2
<b>2</b>	13	15.7
<b>3</b>	124	422.2
<b>4</b>	40	119.4

*Figure 5.2: Output of `df.head()`*

In the next step, we will analyze the data given to us.

4. Analyze the data by looking at the data types, and if the column's observations are numerical, you can use the describe function to understand more about the data. You could also store the column names in an array to avoid typing them in the future if the column names are lengthy:

```
df.dtypes
```

**Output:**

```
Number of Claims      int64
Total payment        float64
dtype: object
df.describe()
```

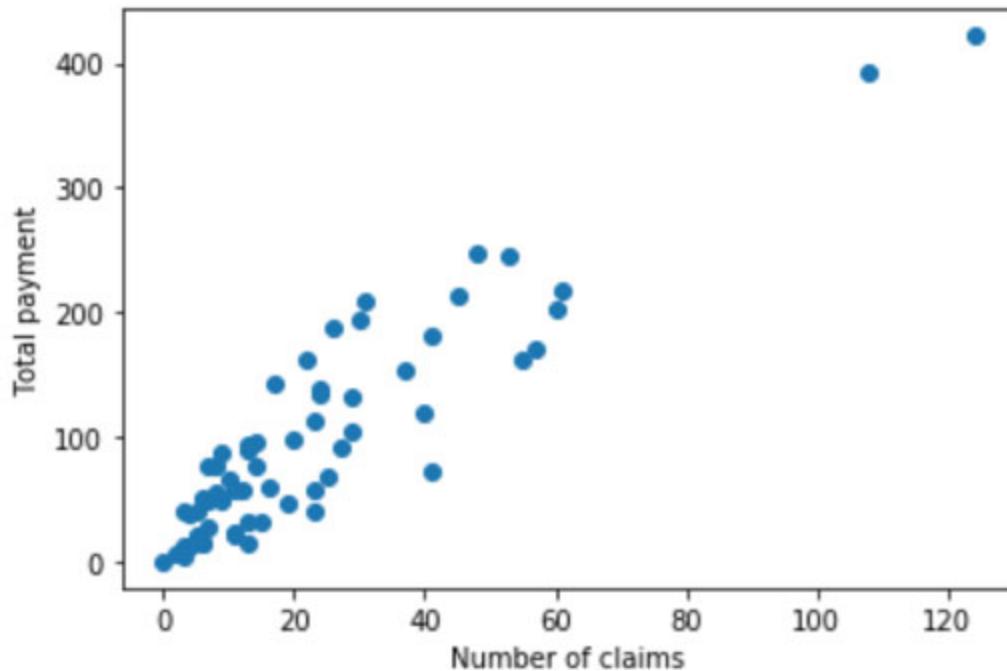
Here is the output of `df.describe()`:

	Number of Claims	Total payment
<b>count</b>	63.000000	63.000000
<b>mean</b>	22.904762	98.187302
<b>std</b>	23.351946	87.327553
<b>min</b>	0.000000	0.000000
<b>25%</b>	7.500000	38.850000
<b>50%</b>	14.000000	73.400000
<b>75%</b>	29.000000	140.000000
<b>max</b>	124.000000	422.200000

*Figure 5.3: Output of df.describe()*

- In the next step, we will visualize the data by creating various plots.
5. Create plots to understand more about the distribution. Here, we will create a scatter plot to find out the distribution of data, the relationship between Column 1 and Column 2.

```
Text(0, 0.5, 'Total payment')
```



*Figure 5.4: Scatter plot between the number of claims and total payment*

6. Separate the dependent and the independent variables and store them in newly defined variables. It is because the models will need to know as to which is a target variable and which is/are the dependent variables. We will also use the `standardize` function to standardize/normalize the column values:

```
x = df['Number of Claims']
y = df['Total payment']

# standardize the data attributes
standardized_x = preprocessing.scale(X)

# standardize the target attribute
standardized_y = preprocessing.scale(Y)
```

7. We used the `test_train_split` function from the `preprocessing` module to split our data into the training and test dataset. The `test_size=0.25` indicates that 25% of the data will be treated as test data, and the rest of it as the training data. The parameter `random_state` is a seed used by the random generator in to pick the observations that will fall under the test and training datasets:

```
x_train, x_test, y_train, y_test =  
train_test_split(standardized_X, standardized_Y,  
test_size=0.25, random_state=42)
```

8. We need to re-shape the data when it comes to simple linear regression since there is only one dependent and one independent variable, whereas the `model.fit` and `model.predict` functions always expect a 2-D array:

```
x_train = x_train.reshape(-1,1)  
y_train = y_train.reshape(-1,1)  
x_test = x_test.reshape(-1,1)
```

9. Now we will use the `LinearRegression` function from the `sklearn` library to initialize our model. Post initialization, we will fit the training data into the model and then predict the results using the fit model.

We will store the predictions in a variable known as `y_predicted`:

```
# Model initialization  
regression_model = LinearRegression()  
# Fit the data(train the model)  
regression_model.fit(x_train, y_train)  
# Predict  
y_predicted = regression_model.predict(x_test)
```

10. We will evaluate the built model using the following accuracy parameters:

- Root mean squared error
- R2 score

We will also find and print the values of the slope and intercept parameters which are the deciding parameters in the equation of a line (Remember  $Y = mx + c$  where  $m$  is the slope and  $c$  is the intercept)

```
# model evaluation  
rmse = mean_squared_error(y_test, y_predicted)  
r2 = r2_score(y_test, y_predicted)  
  
# printing values
```

```
print('Slope:', regression_model.coef_)
print('Intercept:', regression_model.intercept_)
print('Root mean squared error: ', rmse)
print('R2 score: ', r2)
```

### Output:

```
Slope: [[0.92016023]]
Intercept: [-0.03941964]
Root mean squared error: 0.16426914499408302
R2 score: 0.8480051329208176
```

## Conclusion

In this case study, we have shown you as to what a sample output might look like for a simple linear regression problem. The R2 Score, in this case, 0.84, which means that our model can explain the variations in the dataset with a success rate of 84%. Our Root Mean Squared Error (RMSE) is also relatively low, which shows that most of the data points are lying nearby our linear regression line.

The case studies used in this book are just an example, and the real-time datasets might be much more extensive than the datasets used in our example case studies. The case studies aim to give you an idea of how to look at various machine learning problems with different perspectives.

## Case study – II

This case study will take us through the various steps that are involved in building a linear regression model. In this section, we will be focusing on datasets that have one independent variable and one dependent variable which fall under the category of simple linear regression.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been collected from an open-source website which provides several types of data sample for regression analysis, time series analysis, and many more. The data describes the estimated years of the initial occupations and the end of the occupation of **Prehistoric Puebloans**.

This dataset has two columns—the dependent variable is the 'Estimated year of initial occupation,' and the independent variable is the 'Estimated year of the end of the occupation.' Our problem statement involves predicting the estimated year of the end of the occupation of Puebloans, given the estimated year of their start of the occupation.

## Python code and step-by-step regression analysis

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score,
accuracy_score
```

2. Read the data (which is stored in the CSV format) using `python-pandas` and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `Prehistoric.csv`

```
df = pd.read_csv("Datasets/Prehistoric.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the output of `df.head()`:

	Estimated Year of Initial Occupation	Estimated Year of End of Occupation
0	1000	1050
1	1125	1150
2	1087	1213
3	1070	1275
4	1100	1300

*Figure 5.5: Output of df.head()*

In the next step, we will understand how to analyze the data given to us.

4. Analyze the data by looking at the data types, and if the column's observations are numerical, you can use the describe function to understand more about the data. You could also store the column names in an array to avoid typing them in the future since the column names are lengthy:

```
arr_columns = df.columns
print(arr_columns[0])
O/P: Estimated Year of Initial Occupation
print(arr_columns[1])
O/P:Estimated Year of End of Occupation
df.dtypes
```

### Output:

```
Estimated Year of Initial Occupation      int64
Estimated Year of End of Occupation      int64
dtype: object
```

```
df.describe()
```

Here is the output of `df.describe()`:

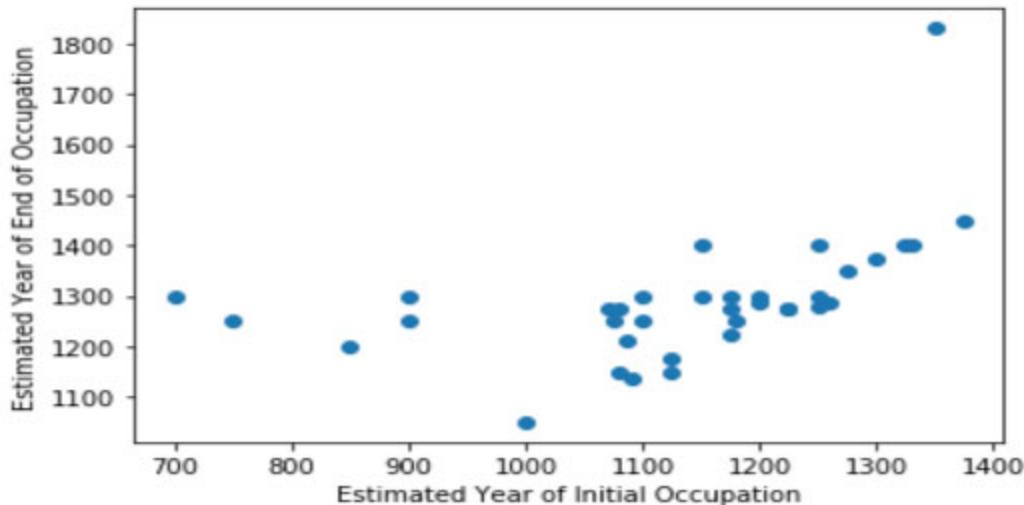
	Estimated Year of Initial Occupation	Estimated Year of End of Occupation
count	36.000000	36.000000
mean	1134.777778	1291.055556
std	158.583211	124.058038
min	700.000000	1050.000000
25%	1080.000000	1250.000000
50%	1162.500000	1277.500000
75%	1250.000000	1300.000000
max	1375.000000	1830.000000

*Figure 5.6: Output of df.describe()*

In the next step, we will visualize the data and gain some insights from it.

5. Create plots to understand more about the distribution. Here, we will create a scatter plot to find out the distribution of data, the relationship between Column 1 and Column 2:

```
plt.scatter(df['Estimated Year of Initial Occupation'],
            df['Estimated Year of End of Occupation'])
plt.xlabel('Estimated Year of Initial Occupation')
plt.ylabel('Estimated Year of End of Occupation')
Text(0, 0.5, 'Estimated Year of End of Occupation')
```



*Figure 5.7: Scatter plot between Estimated Year of Initial Occupation and End of Occupation*

6. Separate the dependent and the independent variables and store them in newly defined variables. It is because the models will need to know

as to which is a target variable and which is/are the dependent variables. We will also use the `standardize` function to standardize/normalize the column values:

```
x = df['Estimated Year of Initial  
Occupation']      #Dependent Variable  
y = df['Estimated Year of End of  
Occupation']      #Independent Variable  
  
# standardize the data attributes  
standardized_x = preprocessing.scale(x)  
  
# standardize the target attribute  
standardized_y = preprocessing.scale(y)
```

7. We used the `test_train_split` function from the `preprocessing` module to split our data into the training and test dataset. The `test_size=0.33` indicates that 33% of the data will be treated and test data, and the rest of it as the training data. The parameter `random_state` is a seed used by the random generator to pick the observations that will fall under the test and training datasets:

```
x_train, x_test, y_train, y_test =  
train_test_split(standardized_x, standardized_y,  
test_size=0.33, random_state=42)
```

8. We need to re-shape the data when it comes to simple linear regression since there is only one dependent and one independent variable, whereas the `model.fit` and `model.predict` functions always expect a 2-D array:

```
x_train = x_train.reshape(-1,1)  
y_train = y_train.reshape(-1,1)  
x_test = x_test.reshape(-1,1)
```

9. Now we will use the `LinearRegression` function from the `sklearn` library to initialize our model. Post initialization, we will fit the training data into the model and then predict the results using the `fit` model.

We will store the predictions in a variable known as `y_predicted`:

```
# Model initialization
```

```

regression_model = LinearRegression()
# Fit the data(train the model)
regression_model.fit(X_train, y_train)
# Predict
y_predicted = regression_model.predict(X_test)

```

10. We will evaluate the built model using the following accuracy parameters:

- Root mean squared error
- R2 score

We will also find the values of the slope and intercept parameters which are the deciding parameters in the equation of a line (Remember  $Y = mx + c$  where  $m$  is the slope and  $c$  is the intercept):

```

# model evaluation
rmse = mean_squared_error(y_test, y_predicted)
r2 = r2_score(y_test, y_predicted)

# Slope and Intercept Values
Slope = regression_model.coef_
Intercept = regression_model.intercept_

```

## Conclusion

The above case study shows us how we can use simple linear regression to predict the target variable based on just one given independent variable. It is essential to understand that **Exploratory Data Analysis or EDA** plays an essential role in Regression analysis as it takes you through the various steps of cleaning and preprocessing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data, the attribute strength, and so on, to revamp your model.

In the next section, we will take a closer look at multiple linear regression and the case studies associated with it.

## Multiple linear regression

Multiple linear regression is a form of linear regression where there are one dependent variable and several independent variables. It is one of the most frequently used forms of a linear regression which is represented by the equation of a line as follows:

$$Y = m_1X_1 + m_2X_2 + m_3X_3 + m_4X_4 + m_5X_5 + \dots + c$$

Also written as:

$$Y = m_1X_1 + m_2X_2 + m_3X_3 + m_4X_4 + m_5X_5 + \dots + c$$

Where  $X_1, X_2, X_3, X_4, X_5$  are the independent variables, and  $Y$  is the dependent variable. In this case,  $m_1, m_2, m_3, m_4, m_5$  are the coefficients of the independent variables that need to be calculated using our model. Some of the other terminologies associated with  $X$  and  $Y$  are explanatory variables (for  $X$ ) and the response variable (for  $Y$ ).

Regression analysis can be used for several purposes, most importantly, to understand the relationship between the dependent and the independent variables, predicting the value/score of a dependent variable, and so on. Multiple linear regression models extend the simple model to fit in more explanatory variables.

Linear regression is aimed at finding the best fit line for all the data points that are present in the data set. The best-fitting line is known as the regression line and is calculated by minimizing the sum of squares of residuals.

We have two case studies that we will be working on for multiple linear regression.

The case studies presented to us will give us a glimpse of the dataset that we are working with, possible EDA methods on the dataset, and building a model for the given dataset. The EDA steps are example steps, and you could do a deep-dive on each of these steps to understand more and learn more for yourself.

There are also some example plots for each of the datasets, but I highly encourage you to build various types of plots and understand the different trends and patterns between the independent and the dependent variables.

In the upcoming sections, we will look at case studies on multiple linear regression to understand the real-time implementation in the industry.

## Case study – I

This case study will take us through the various steps that are involved in building a multiple linear regression model. In this section, we will be focusing on datasets that have one independent variable and one dependent variable which fall under the category of multiple linear regression.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been collected from an open-source website which provides several types of data sample for regression analysis, time series analysis, and many more. The data describes the health scenario and death rate of residents in an area. It gives us an understanding of death rates based on doctor availability, hospital availability, per capita income of the households, and so on.

This dataset has various attributes - the dependent variables are the various factors affecting Death rate, and the independent variable is Death rate per thousand residents. Our problem statement involves predicting the Death rate per 1000 residents, given the dependent attributes.

## Python code and step-by-step regression analysis

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

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

2. Step 2: Read the data (which is stored in the CSV format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `Health.csv`.

```
df = pd.read_csv("Datasets/MLR/Health.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the output of `df.head()`:

	doctor availability per 100,000 residents	hospital availability per 100,000 residents	annual per capita income in thousands of dollars	population density people per square mile	Death rate per 1000 residents
0	78	284	9.1	109	8.0
1	68	433	8.7	144	9.3
2	70	739	7.2	113	7.5
3	96	1792	8.9	97	8.9
4	74	477	8.3	206	10.2

*Figure 5.8: Output of `df.head()`*

In the next step, we will analyze the data given to us.

4. Analyze the data by looking at the data types, and if the column's observations are numerical, you can use the `describe` function to understand more about the data. You could also store the column names in an array to avoid typing them in the future if the column names are lengthy.

Also printing out the column names to view them.

```
df.columns
```

**Output:**

```
Index(['doctor availability per 100,000 residents',  
       'hospital availability per 100,000 residents',  
       'annual per capita income in thousands of dollars',  
       'population density people per square mile',
```

```
'Death rate per 1000 residents'],
dtype='object')
```

```
df.dtypes
```

### Output:

```
doctor availability per 100,000 residents           int64
hospital availability per 100,000 residents          int64
annual per capita income in thousands of dollars    float64
population density people per square mile          int64
Death rate per 1000 residents                        float64
dtype: object
```

```
df.describe()
```

Here is the output of `df.describe()`:

	doctor availability per 100,000 residents	hospital availability per 100,000 residents	annual per capita income in thousands of dollars	population density people per square mile	Death rate per 1000 residents
count	53.000000	53.000000	53.000000	53.000000	53.000000
mean	116.094340	589.792453	9.435849	110.641509	9.30566
std	37.886604	332.618305	1.075442	47.179728	1.66253
min	60.000000	190.000000	7.200000	35.000000	3.60000
25%	82.000000	353.000000	8.800000	80.000000	8.30000
50%	114.000000	525.000000	9.500000	103.000000	9.40000
75%	134.000000	686.000000	10.300000	129.000000	10.30000
max	238.000000	1792.000000	13.000000	292.000000	12.80000

*Figure 5.9: Output of df.describe()*

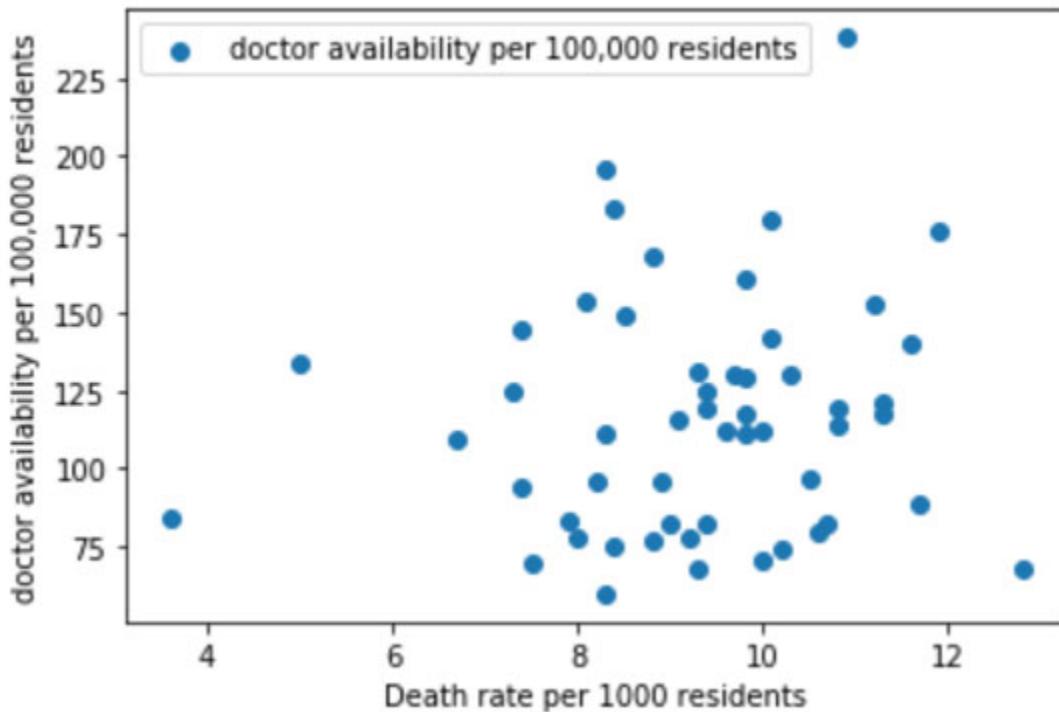
In the next section, we will look at how to visualize the data by creating various plots.

5. Create plots to understand more about the distribution. Here, we will create several scatter plots to find out the relationship between the independent variable and the several dependent variables.

The following code snippet creates a scatter plot between the `Death rate per 1000 residents` and `doctor availability per 100,000 residents`:

```
plt.scatter('Death rate per 1000 residents', 'doctor availability per 100,000 residents', data=df)
plt.xlabel('Death rate per 1000 residents')
plt.ylabel('doctor availability per 100,000 residents')
```

```
<matplotlib.legend.Legend at 0x105c27da0>
```

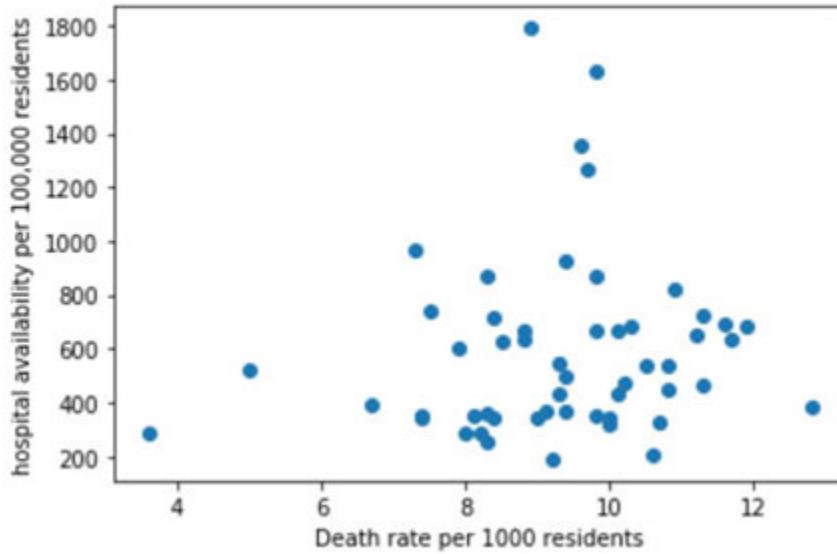


*Figure 5.10: Scatter plot between the Death rate per 1000 residents and doctor availability per 100,000 residents*

The following code snippet creates a scatter plot between the Death rate per 1000 residents and hospital availability per 100,000 residents:

```
plt.scatter( 'Death rate per 1000 residents', 'hospital availability per 100,000 residents', data=df)
plt.xlabel('Death rate per 1000 residents')
plt.ylabel('hospital availability per 100,000 residents')
```

```
Text(0, 0.5, 'hospital availability per 100,000 residents')
```

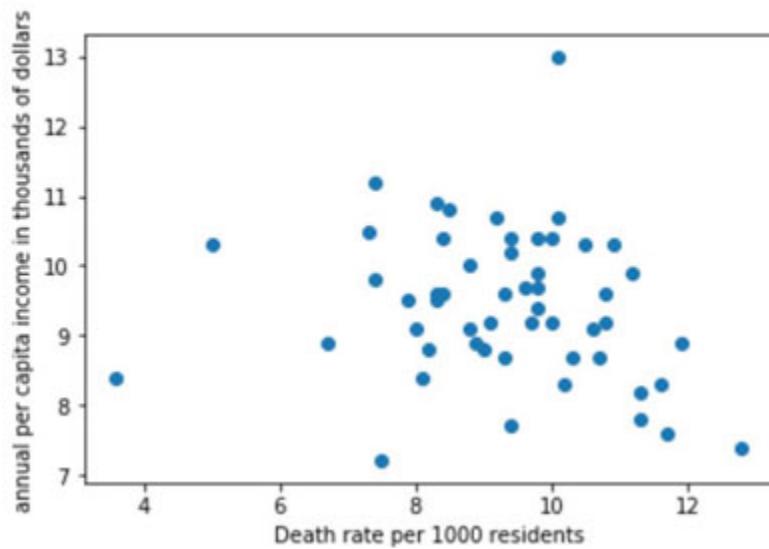


**Figure 5.11:** Scatter plot between the Death rate per 1000 residents and hospital availability per 100,000 residents

The following code snippet creates a scatter plot between the Death rate per 1000 residents and annual per capita income in thousands of dollars:

```
plt.scatter('Death rate per 1000 residents', 'annual per capita income in thousands of dollars', data=df)
plt.xlabel('Death rate per 1000 residents')
plt.ylabel('annual per capita income in thousands of dollars')
```

```
Text(0, 0.5, 'annual per capita income in thousands of dollars')
```

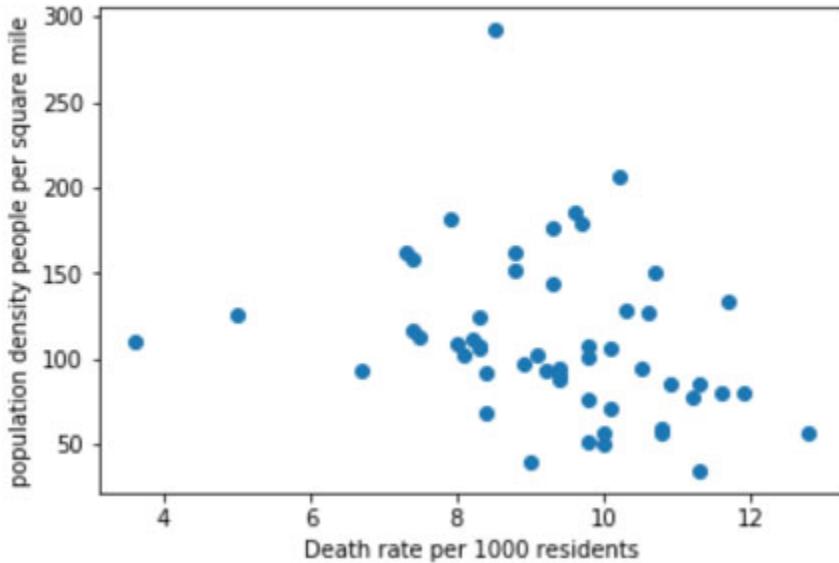


**Figure 5.12:** Scatter plot between the Death rate per 1000 residents and annual per capita income in thousands of dollars

The following code snippet creates a scatter plot between the Death rate per 1000 residents and population density people per square mile:

```
plt.scatter( 'Death rate per 1000 residents', 'population density people per square mile', data=df)
plt.xlabel('Death rate per 1000 residents')
plt.ylabel('population density people per square mile')
```

```
Text(0, 0.5, 'population density people per square mile')
```



**Figure 5.13:** Scatter plot between the Death rate per 1000 residents and population density people per square mile

In the next step, we will take a look at how to normalize the data.

6. Separate the dependent and the independent variables and store them in newly defined variables. It is because the models will need to know as to which is a target variable and which is/are the dependent variables. We will also use the `standardize` function to standardize/normalize the column values:

```
Y = df['Death rate per 1000 residents']
X = df.drop('Death rate per 1000 residents', axis=1)

# standardize the data attributes
standardized_X = preprocessing.scale(X)

# standardize the target attribute
standardized_Y = preprocessing.scale(Y)
```

7. We used the `test_train_split` function from the `preprocessing` module to split our data into the training and test dataset. The `test_size=0.20` indicates that 20% of the data will be treated and test data, and the rest of it as the training data. The parameter `random_state` is a seed used by the random generator to pick the observations that will fall under the test and training datasets:

```
x_train, x_test, y_train, y_test =  
train_test_split(standardized_X, standardized_Y,  
test_size=0.20, random_state=42)
```

8. We need to re-shape the data when it comes to simple linear regression since there is only one dependent and one independent variable, whereas the `model.fit` and `model.predict` functions always expect a 2-D array:

```
y_train = y_train.reshape(-1,1)
```

9. Now we will use the `LinearRegression` function from the `sklearn` library to initialize our model. Post initialization, we will fit the training data into the model and then predict the results using the `fit` model.

We will store the predictions in a variable known as `y_predicted`:

```
# Model initialization  
regression_model = LinearRegression()  
# Fit the data(train the model)  
regression_model.fit(x_train, y_train)  
# Predict  
y_predicted = regression_model.predict(x_test)
```

10. We will evaluate the built model using the following accuracy parameters:

- Root mean squared error
- R2 score

We will also find the values of the slope and intercept parameters which are the deciding parameters in the equation of a line (remember  $Y = mx + c$  where m is the slope and c is the intercept):

```
# model evaluation  
rmse = mean_squared_error(y_test, y_predicted)  
r2 = r2_score(y_test, y_predicted)  
  
# Slope and Intercept Values  
Slope = regression_model.coef_  
Intercept = regression_model.intercept_
```

## Conclusion

The above case study takes us through how the different attributes in multiple linear regression problems help create an end-to-end machine learning model to predict the target variable for us.

You can see that the various scatter plots help us understand the behavior of the independent variable to the dependent variables. In the case of multiple attributes, we need to visualize the distribution of various attributes to the target variable.

You can look at the intercept and the slope value to construct your linear equation. Looking at the R<sup>2</sup> score and RMSE values will help you understand as to how accurately your model can predict the target variable.

## Case study – II

This case study will take us through the various steps that are involved in building a multiple linear regression model. In this section, we will be focusing on datasets that have one independent variable and multiple dependent variables that fall under the category of multiple linear regression.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been collected from an open-source website which provides several types of data sample for regression analysis, time series analysis, and so on.

This dataset has several columns—the independent variable is the `avghouseholdszie`, and the independent variables are the `Total Payment`, that is, the total payment that is likely to be paid based on the number of claims files. Our problem statement involves predicting the total payment to be paid, given the number of claims in Sweden for data collected over some time.

## Python code and step-by-step regression analysis

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score,
accuracy_score
```

2. Read the data (which is stored in the CSV format) using `python-pandas` and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `avg-household-size.csv`

```
df = pd.read_csv("Datasets/MLR/avg-household-size.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the first image describing the output of `df.head()`:

Out[18]:	<bound method NDFrame.head of	statefips	countyfips	avghouseholdszie	\
0	2	13	2.43		
1	2	16	3.59		
2	2	20	2.77		
3	2	50	3.86		
4	2	60	2.50		
5	2	68	2.34		
6	2	70	3.56		
7	2	90	2.69		
8	2	100	2.12		
9	2	105	2.26		
10	2	110	2.62		
11	2	122	2.57		
12	2	130	2.55		
13	2	150	2.98		
14	2	158	4.65		

*Figure 5.14: df.head() output part - I*

Here is the second image describing the output of `df.head()`:

```

      geography
0          Aleutians East Borough, Alaska
1          Aleutians West Census Area, Alaska
2          Anchorage Municipality, Alaska
3          Bethel Census Area, Alaska
4          Bristol Bay Borough, Alaska
5          Denali Borough, Alaska
6          Dillingham Census Area, Alaska
7          Fairbanks North Star Borough, Alaska
8          Haines Borough, Alaska
9          Hoonah-Angoon Census Area, Alaska
10         Juneau City and Borough, Alaska
11         Kenai Peninsula Borough, Alaska
12         Ketchikan Gateway Borough, Alaska
13         Kodiak Island Borough, Alaska
14         Kusilvak Census Area, Alaska

```

*Figure 5.15: df.head() output part – II*

In the next step, we will analyze the data given to us.

- Analyze the data by looking at the data types, and if the column's observations are numerical, you can use the describe function to understand more about the data. You could also store the column names in an array to avoid typing them in the future if the column names are lengthy.

Also printing out the column names, to view them:

```
df.columns
```

#### Output:

```
Index(['statefips', 'countyfips', 'avghouseholdsiz
e', 'geography'], dtype='object')
```

```
df.dtypes
```

#### Output:

statefips	int64
countyfips	int64
avghouseholdsize	float64
geography	object

5. As we can see in the dtypes, the geography column is of type object. If we look closely at the column, it has two comma-separated string values. We can separate the geography variable into two variables, both of which would be of the object type, and we can use encoders to encode the object variables in the later stages.

In the following code snippet, we will split the geography column and create two new columns `Area` and `Country` out of this column.

```
new = df['geography'].str.split(", ", n = 1, expand = True)
df['Area'] = new[0]
df['Country'] = new[1]
df.drop(columns = ["geography"], inplace = True)
```

To understand the behavior of categorical variables, we have a command named `value_counts()` that gives us the distribution of categorical variables (In our case, `Area` and `Country` variables in the dataset):

```
df['Area'].value_counts()
```

Here is an output image of `df['Area'].value_counts()`:

Washington County	30
Jefferson County	25
Franklin County	24
Lincoln County	23
Jackson County	23
Madison County	19
Montgomery County	18
Clay County	18
Marion County	17
Union County	17
Monroe County	17
Wayne County	16
Warren County	14
Greene County	14
	..

*Figure 5.16: Output of `df['Area'].value_counts()`*

```
df['Country'].value_counts()
```

Here is an output image of `df['Country'].value_counts()`:

Texas	254
Georgia	159
Virginia	133
Kentucky	120
Missouri	115
Kansas	105
Illinois	102
North Carolina	100
Iowa	99
Tennessee	95
Nebraska	93
Indiana	92
Ohio	88
Minnesota	87
Michigan	83

*Figure 5.17: Output of df['Country'].value\_counts()*

In the next step, we will encode the categorical attributes present in our dataset.

6. In the following code snippet, we will use a label encoder and encode the categorical values to numerical labels:

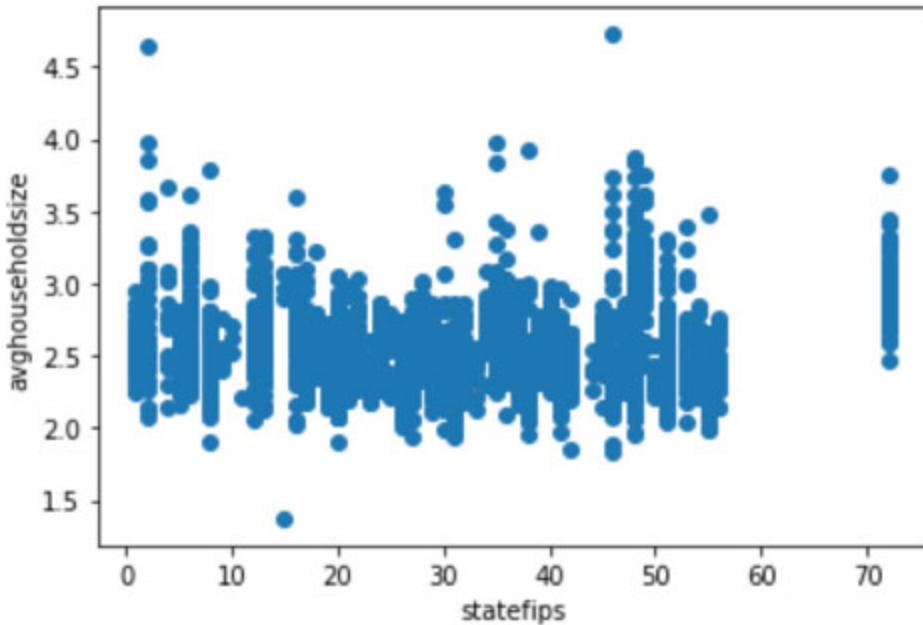
```
le = LabelEncoder()
df["Area"] = le.fit_transform(df["Area"])
```

7. Create plots to understand more about the distribution. Here, we will create scatter plots to find out the relationship between the independent variable and one of the dependent variables.

The following code snippet creates a scatter plot between statefips and avghouseholdszie:

```
plt.scatter('statefips', 'avghouseholdszie', data=df)
plt.xlabel('statefips')
plt.ylabel('avghouseholdszie')
```

```
Text(0, 0.5, 'avghouseholdszie')
```



**Figure 5.18:** Scatter plot between State fips and avghouseholdszie

In the next section, we will separate the dependent and independent variables present in the dataset.

8. Separate the dependent and the independent variables and store them in newly defined variables. It is because the models will need to know as to which is a target variable and which is/are the dependent variables.

```
y = df['avghouseholdszie']
x = df.drop('avghouseholdszie', axis=1)
```

9. We used the `test_train_split` function from the preprocessing module to split our data into the training and test dataset. The `test_size=0.33` indicates that 33% of the data will be treated as test data, and the rest of it as the training data. The parameter `random_state` is a seed used by the random generator to pick the observations that will fall under the test and training datasets:

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.33, random_state=42)
```

10. Now we will use the `LinearRegression` function from the `sklearn` library to initialize our model. Post initialization, we will fit the

training data into the model and then predict the results using the fit model.

We will store the predictions in a variable known as `y_predicted`:

```
# Model initialization
regression_model = LinearRegression()
# Fit the data(train the model)
regression_model.fit(X_train, y_train)
# Predict
y_predicted = regression_model.predict(X_test)
```

11. We will evaluate the built model using the following accuracy parameters:

- Root mean squared error
- R2 score

We will also find the values of the slope and intercept parameters which are the deciding parameters in the equation of a line (Remember  $Y = mx + c$  where  $m$  is the slope and  $c$  is the intercept):

```
# model evaluation
rmse = mean_squared_error(y_test, y_predicted)
r2 = r2_score(y_test, y_predicted)

# Slope and Intercept Values
Slope = regression_model.coef_
Intercept = regression_model.intercept_
```

## Conclusion

The above case study takes us through how the different attributes in multiple linear regression problems help create an end-to-end machine learning model to predict the target variable for us.

You can see that the various scatter plots help us understand the behavior of the independent variable concerning the dependent variables. In the case of multiple attributes, we need to visualize the distribution of various attributes to the target variable.

You can look at the intercept and the slope value to construct your linear equation. Looking at the R<sup>2</sup> score and RMSE values will help you understand as to how accurately your model can predict the target variable.

In the next chapter, we will look into another fundamental regression analysis technique known as **Logistic Regression**. Logistic regression is quite popular in the industry for customer-facing organizations where insights from the customers are quite valuable to design products/services by the organization.

## **Quiz – What did you learn about linear regression**

1. The formula for a regression equation is  $Y = 4X+3$ . What would be the predicted Y score for a person score 2 on X?
2. What is the importance of correlation coefficient (R) in linear regression?
3. What is the range of R<sup>2</sup>?
4. Which type of regression is used extensively in econometrics?

# CHAPTER 6

## Logistic Regression – An Introduction

### Introduction

In the last chapter, we studied linear regression, which is a type of simple regression and uses the equation of a line to build a machine learning model for us. We saw that simple linear regression has a dependent variable and one independent variable, whereas multiple linear regression has one dependent variable and several independent variables.

Linear regression was based on a straight-line equation, but logistic regression is a type of regression that is used when the output is binary, that is, a Yes/No.

### Structure

- Logistic regression with an example
- Inner workings of logistic regression
- Logistic regression equation
- Gradient descent
- Case study logistic regression – I
- Case study logistic regression – II
- Conclusion
- Quiz

### Objectives

- Understand the working of logistic regression and gradient descent
- Practice hands-on Python coding for logistic regression related problem statements

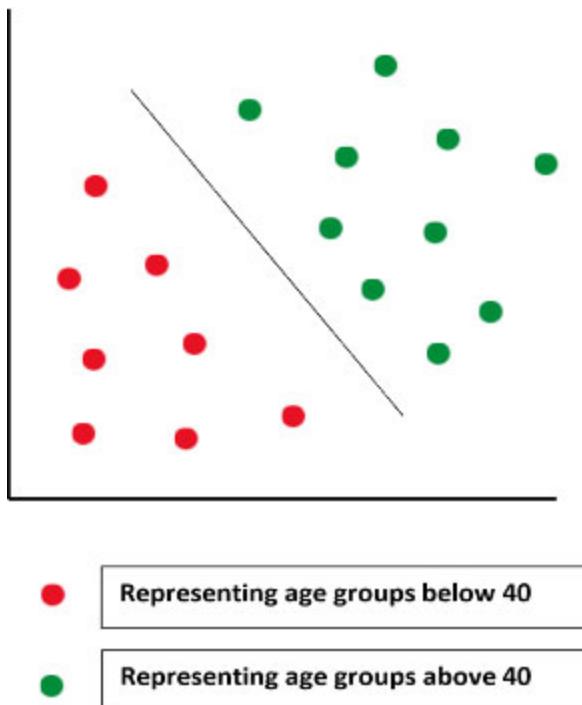
## Logistic regression with an example

To understand how logistic regression works in the real world, we will start with a real-time example where one can apply logistic regression to predict the desired results.

Here is an example problem to understand logistic regression in context to a real-world scenario:

Let's say that an e-commerce website has collected some data on a particular product and wants to understand as to which age groups will buy that product. Now imagine that the age groups are split into two categories - Age group of 40 and above (represented as 1) and Age-group below 40 (represented as 0). Our problem statement will involve predicting which of these age groups show a tendency to buy the product so that the company can focus more on branding for that age group.

If we draw a line to represent our problem, this is what it would look like:



*Figure 6.1: Line separating two classes in logistic regression*

When you think of this in a real-time scenario, you cannot predict with 100% certainty that a particular age group will buy the product or not, but you could be 80% sure or 50% sure, and therefore the concept of probability comes into the picture.

In simple words, logistic regression is a regression analysis used when the dependent variable is binary or dichotomous. In logistic regression, we build a model that will predict with a certain amount of probability as to what the most likely outcome would be, for a given value of an attribute.

It is a form of predictive analysis and generates a probability value between 0 and 1.

## Inner workings of logistic regression

In the above-given example, the probabilities generated by a logistic regression indicate to us that if the probability value is higher ( $>0.5$ ), then the result is likely to be 1, in our case, Age group of 40 and above would prefer to buy the product, and if the probability value is lower( $\leq 0.5$ ), then the result is likely to be 0, that is, Age group below 40 would prefer to buy the product.

Logistic regression aims to do the following:

- Based on the independent variables or attributes, we will model the probability of a particular event happening or not. The independent variables can be categorical or non-categorical in this case.
- For a selected observation (validation observation), estimate the probability of an event happening versus not happening
- For a set of given independent variables, predict the target binary variable.

In the above example, we can see that when there are two classes of different categories, logistic regression can be a useful and practical algorithm.

## Logistic regression equation

Logistic regression equations are an improvised version of linear regression, as they are more flexible due to the separation graph, which is more of a curve than a straight line. Linear regression solely relies on straight lines, which are not realistic markers of separation in a real-time use case/ industrial scenarios.

As we already know, the linear regression equation for multiple dependent variables would look like this:

$$Y = m(X1 + X2 + X3 + X4 + X5 + \dots) + c$$

Where  $X1, X2, X3, X4, X5 \dots$  are the independent variables/features present in the dataset.

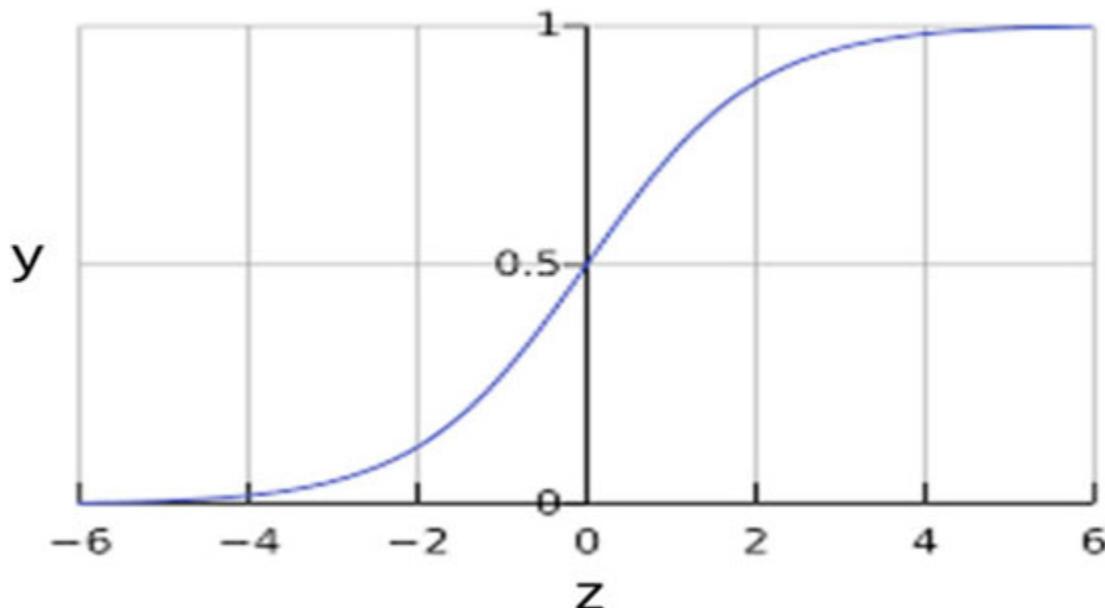
Similarly, the equation for logistic regression, defined by a sigmoid function is as follows:

$$Y' = 1 / (1 + e(-z))$$

Where  $Y'$  is the output of the logistic regression model for sample input and  $z$  is as follows:

$$z = b + w1x1 + w2x2 + \dots + wNxN$$

In the equation mentioned above,  $b$  is the bias, and the  $w$  values are the weights learned by the model. To visualize how a logistic regression curve (also known as the logit curve) would look like, we have plotted a graph which is shown in the following image:



*Figure 6.2: Logistic regression curve*

[Figure 6.2](#) gives us an idea of how the logistic regression curve looks like. The sigmoid function is a logarithmic function that gives us a curve instead of a straight line so that the prediction can become much more comfortable.

Our logistic regression equation can also be represented as follows:

$$z = \log(y/(1-y))$$

In this case,  $z$  is also known as log-odds, as it gives us the log of the odds of an event happening.

The expression  $(y/(1-y))$  is known as the odds. In logistic regression, the dependent variable is logit, which is the natural log( $\ln$ ) of the odds.

## Gradient descent

As we all know, machine learning algorithms are all about optimizing. In our previous chapter about linear regression, we were trying to optimize the values of the slope and the intercept to obtain a line that fits well for our dataset.

Gradient descent is meant to do precisely that! Gradient descent can solve a lot of optimization problems by descending to the lowest point in our loss function. It identifies the optimal values by taking more significant steps when the actual value is far away and smaller steps when the value is closer to the desired value.

**A loss function is a function that determines how well our algorithm can model the given data.**

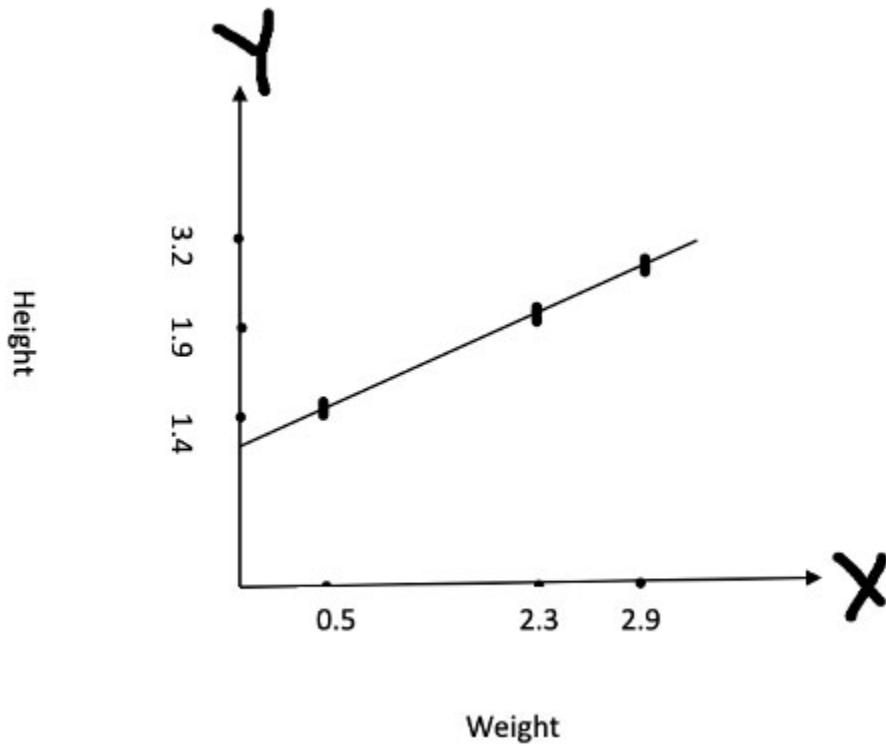
## How does gradient descent work?

Here are the steps for running a gradient descent algorithm for a given linear regression example:

1. Use the sum of the squared residuals as a loss function to evaluate how well a line fits the data.
2. We take the derivative of the RSS to the intercept and to the slope to find the minima.
3. Pick random slope and intercept values, calculate the new values, and incorporate them in the equation again.

Let's take an example of an equation of a line where we are trying to optimize the values of the intercept and the slope to arrive at the best fit line for our dataset.

Here, we have a height versus weight graph wherein; we need to predict the height given the weight of a person. The training data has some data points collected for the height and the weight. We will see as to how gradient descent will work in the scenario below:



*Figure 6.3: Linear regression line*

We know that the equation of a line is as follows:

$$Y = mX + c$$

Where  $Y$  is the height and  $X$  is the weight of the person,  $m$  is the slope of the line, and  $c$  is the intercept of the line.

The three key pairs  $(X, Y)$  for our given line is:

- (0.5, 1.4)
- (2.3, 1.9)
- (2.9, 3.2)

Listed below are the steps that gradient descent follows, to arrive at the most optimal solution:

## Step 1

In the case of linear regression, our loss function is the **Sum of Squared Residuals/Residual Sum of Squares (RSS)** since that is the way to understand if our algorithm fits the model perfectly. We know that the formula for RSS that we looked into in our last chapter is.

RSS in our case for the three key pairs is:

$$\begin{aligned} \text{RSS} \\ = & (1.4 - (c + m*0.5))^2 \\ + & (1.9 - (c + m*2.3))^2 \\ + & (3.2 - (c + m*2.9))^2 \end{aligned}$$

To move forward or backward, gradient descent uses something called a **step size** whose formula is given by:

$$\text{Step Size} = \text{Slope}/\text{Intercept} * \text{Learning Rate}$$

## Step 2

Derivatives of RSS with respect to intercept (c):

$$\begin{aligned} \frac{d}{dc} = & (-2 * (c + m*0.5)) \\ + & (-2 * (c + m*2.3)) \\ + & (-2 * (c + m*2.9)) \end{aligned}$$

**INFO: Two or more derivatives of the same function is called a gradient.**

Derivatives of RSS with respect to slope (m):

$$\begin{aligned} \frac{d}{dm} = & (-2 * (c + m*0.5) * 0.5) \\ + & (-2 * (c + m*2.3) * 2.3) \\ + & (-2 * (c + m*2.9) * 2.9) \end{aligned}$$

We will use the high gradients to reach the lowest point in the loss function, and that is why this algorithm is known as a gradient descent algorithm!

## Step 3

We will pick  $c = 0$  and  $m = 1$  for our first iteration.

**Iteration 1:**  $m = 1, c=0$

Substituting the values in the above equations

$$\frac{d}{dm} = -0.8$$

$$\frac{d}{dc} = -1.6$$

Considering the *learning rate* = 0.01

$$\text{Step size} = -0.8 * 0.01 = -0.008$$

$$\text{Step size} = -1.6 * 0.01 = -0.016$$

$$\text{new } m = \text{old } m - \text{step size} \quad (\text{new } m = 1 - (-0.008))$$

$$\text{new } c = \text{old } c - \text{step size} \quad (\text{new } c = 0 - (-0.016))$$

**Iteration 2:**  $m = 1.008, c = 0.016$

Substituting the values in the above equations:

$$\frac{d}{dm} = -28.29$$

$$\frac{d}{dc} = -11.58$$

$$\text{Step size} = -28.29 * 0.01 = -2.83$$

$$\text{Step size} = -11.58 * 0.01 = -1.15$$

$$\text{new } m = 1.008 - (-2.83) = 3.83$$

$$\text{new } c = 0.016 - (-1.15) = 1.16$$

**INFO:** The learning rate is a hyperparameter that can be configured and used while training. It has a small positive value, generally in the 0.0 to 1.0 range.

**Iteration 3:**

**Iteration 4:**

All the above iterations will keep updating the values of intercept ( $c$ ) and slope ( $m$ ).

After several iterations, we will arrive at the following values of intercept ( $c$ ) and slope ( $m$ ):

$$c = 0.95$$

$$m = 0.64$$

In practice, a gradient descent algorithm will stop when the **Step Size** is very close to zero (0) or if it has reached the maximum limit on the number of steps. In practice, the maximum number of steps could be 1000 or higher.

Gradient descent is the algorithm behind optimizing the loss function in most of the regression problems, and it is worth understanding the mechanics behind gradient descent.

**NOTE: The sum of squared residuals is just one type of loss function. Every algorithm has its loss function, which follows the same procedure.**

## Case study - I

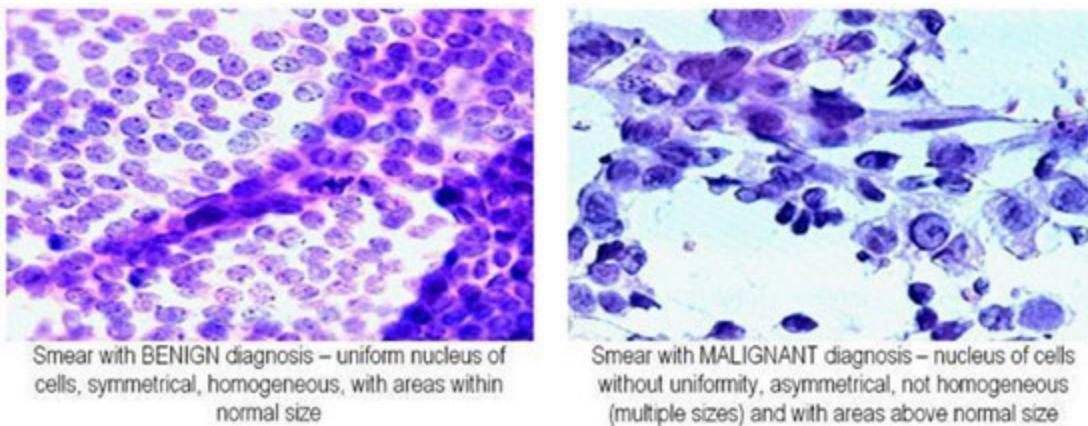
This case study will take us through the various steps that are involved in building a logistic regression model. In this section, we will be focusing on datasets that have different types of independent variables and one dependent variable, which is a binary variable (Yes/No).

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been collected from the UCI machine learning repository. The dataset is known as the **Breast Cancer Wisconsin (Diagnostic) Dataset**.

The features are computed from a digitized image of a **fine needle aspirate (FNA)** of a breast mass. They describe the characteristics of the cell nuclei present in the image. An example image is as follows:



*Figure 6.4: Digitized image of a fine needle aspirate(FNA) of a breast mass*

Several attributes can be derived from [\*Figure 6.4\*](#), which would be used for classifying a given image using our logistic regression algorithm. In the next section, we will look at the different attributes that will act as significant characteristics which will help make the classification more accurate in our case.

## Attribute information

- ID number
- Diagnosis (M = malignant, B = benign)
- 3-32 Attribute number 3 to attribute number 32 are the mean, standard error, and the maximum value defined for each of the following ten features:

Ten features are computed for each cell nucleus (numerical):

1. **radius (mean of distances from the center to points on the perimeter):** mean, standard error, and maximum:
  - Feature number 3,4 and 5
2. **texture (standard deviation of gray-scale values):** mean, standard error, and maximum:
  - Feature number 6,7 and 8
3. **perimeter:** mean, standard error, and maximum:
  - Feature number 9,10 and 11
4. **area:** mean, standard error, and maximum:
  - Feature number 12,13 and 14
5. **smoothness (local variation in radius lengths):** mean, standard error, and maximum:
  - Feature number 15,16 and 17
6. **compactness (perimeter<sup>2</sup> / area - 1.0):** mean, standard error, and maximum:
  - Feature number 18,19 and 20

- 7. **concavity (severity of concave portions of the contour):** mean, standard error, and maximum:
  - Feature number 21, 22 and 23
- 8. **concave points (number of concave portions of the contour):** mean, standard error, and maximum:
  - Feature number 24, 25 and 26
- 9. **symmetry:** mean, standard error, and maximum:
  - Feature number 27, 28 and 29
- 10. **fractal dimension ("coastline approximation" - 1):** mean, standard error and maximum:
  - Feature number 30, 31 and 32

The above attributes are the characteristics that have been retrieved from an image and will be used for pre-processing our data. The importance of these characteristics may vary based on how they contribute to the algorithm.

In the next section, we will look at the step-by-step python code for regression analysis.

## Python code and step-by-step regression analysis

It is one of the most crucial and essential sections of our case study—being able to analyze the data using Python libraries and deriving insights from the data.

The following steps have a Python code snippet and its corresponding output in most of the cases. You can type the following code in your Jupyter Notebook/A google colab notebook or on your local editor and execute them to see the desired results.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

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

```

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
%matplotlib inline

```

2. Read the data (which is stored in the CSV format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `data.csv`. I have stored the datasets under the `/LogisticRegression` folder under the `/Datasets` folder on my local system.

```

df = pd.read_csv('Datasets/LogisticRegression/data.csv',
index_col='id')

```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame.

The following figure represents the output of `df.head()` when executed on a Jupyter Notebook:

```
df.head()
```

Here is the output image of the `df.head()` command:

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave_points_mean	symmetry_mean
id										
842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27780	0.3001	0.14710	
842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	
84300903	M	19.89	21.25	130.00	1203.0	0.10980	0.15990	0.1974	0.12790	
84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	
84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13260	0.1980	0.10430	

*Figure 6.5: Output of df.head()*

4. Analyze the data by looking at the data types, and if the columns observations are numerical, you can use the `describe` function to understand more about the data:

```
df.dtypes
```

The following image shows the output of `df.dtypes`:

```

diagnosis                      object
radius_mean                     float64
texture_mean                    float64
perimeter_mean                 float64
area_mean                       float64
smoothness_mean                float64
compactness_mean               float64
concavity_mean                 float64
concave_points_mean            float64
symmetry_mean                  float64
fractal_dimension_mean         float64
radius_se                       float64
texture_se                      float64
perimeter_se                   float64
area_se                         float64
smoothness_se                  float64
compactness_se                 float64
concavity_se                   float64
concave_points_se              float64
symmetry_se                     float64
fractal_dimension_se           float64
radius_worst                    float64
texture_worst                   float64
perimeter_worst                float64
area_worst                      float64
smoothness_worst               float64
compactness_worst              float64
concavity_worst                float64
concave_points_worst            float64
symmetry_worst                 float64
fractal_dimension_worst        float64
Unnamed: 32                      float64
dtype: object

```

*Figure 6.6: Output of df.dtypes*

5. In *Step 4*, we noticed that there was a column named **Unnamed: 32**, to look at what that column contains and to check if any missing or null values are present in the dataset, we use the `info()` method on our DataFrame:

```
df.info()
```

**Output:**

```

concave points_worst          569 non-null float64
symmetry_worst                569 non-null float64
fractal_dimension_worst       569 non-null float64
Unnamed: 32                   0 non-null float64
dtypes: float64(31), object(1)

```

6. As we saw in Step 5, the **Unnamed: 32** column contains zero values and can be removed from the analysis as it won't be of any use to us. We will use the `drop()` function to drop the columns which aren't useful to us:

```
df = df.drop(columns = ['Unnamed: 32'])
df.head()
```

The below image represents the `df.head()` output once the `Unnamed` column is dropped from the table:

id	texture_worst	perimeter_worst	area_worst	smoothness_worst	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	fractal_dimension_worst
38	17.33	184.60	2019.0	0.1622	0.6656	0.7119	0.2654	0.4601	0.11890
99	23.41	158.80	1956.0	0.1238	0.1866	0.2416	0.1860	0.2750	0.08902
57	25.53	152.50	1709.0	0.1444	0.4245	0.4504	0.2430	0.3613	0.08758
91	26.50	98.87	567.7	0.2098	0.8663	0.6869	0.2575	0.6638	0.17300
54	16.67	152.20	1575.0	0.1374	0.2050	0.4000	0.1625	0.2364	0.07678

*Figure 6.7: Output of `df.head()` once `Unnamed` column is dropped*

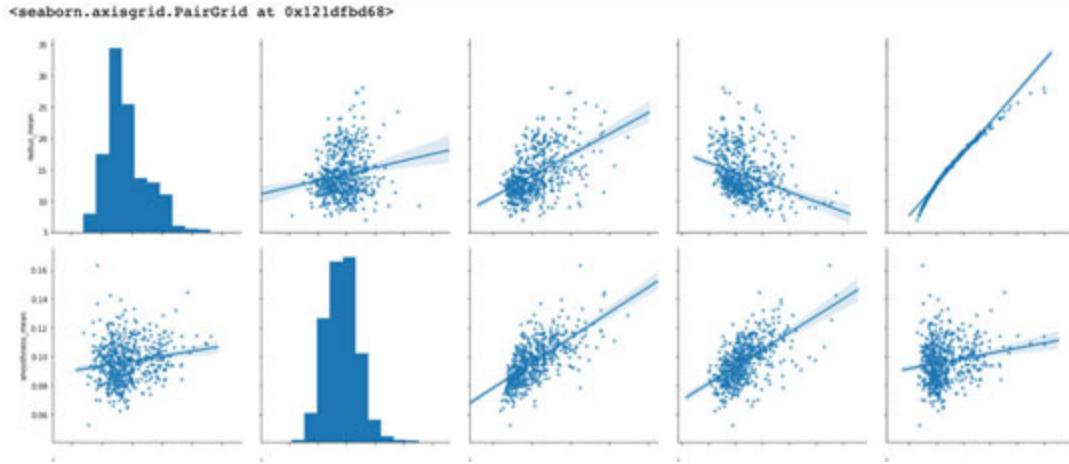
7. We will now visualize the data to look at how the data is distributed across our dataset. We will create pair plots of some selected columns (you can choose the ones of your choice) using the seaborn (`sns`) library. Pair plots are useful when you want to visualize the relationship between two variables.

You can create a pair of plots in such a way that you can visualize the relationship between each pair of attributes. Here, we have chosen to use five attributes.

The following code snippet will help us create all the possible pair plots between these five attributes:

```
df_n =
df[['radius_mean', 'smoothness_mean', 'compactness_mean', 'fractal_dimension_mean', 'area_mean']]
sns.pairplot(df_n, height=4, kind="reg", markers=".")
```

The following image is the output of all the possible pair plots that were plotted using the five attributes chosen in the preceding code snippet:



**Figure 6.8:** Pair plot of the selected attributes

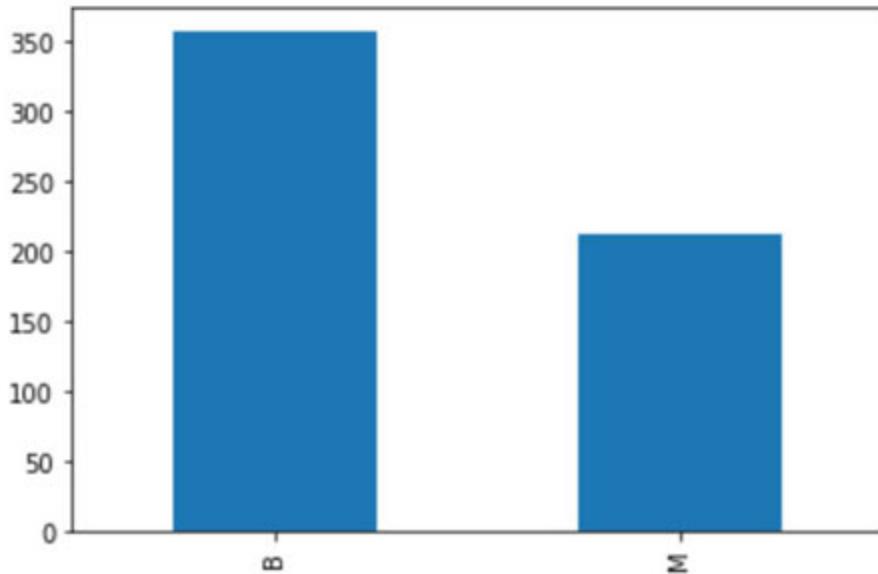
8. The function `value_counts()` is extremely useful for categorical variables. It gives the count of each categorical variable value present in our dataset. Here, we have plotted a bar graph showing the distribution of the dependent variable in our dataset.

The following code snippet will display the count of the categorical variable diagnosis present in our dataset as well draw a bar graph to represent the count visually:

```
df['diagnosis'].value_counts()
df['diagnosis'].value_counts().plot.bar()
```

Here is the output bar graph that shows the category distribution for the 'diagnosis' variable:

```
<matplotlib.axes._subplots.AxesSubplot at 0x121d74358>
```



*Figure 6.9: Bar graph representing the target variable distribution*

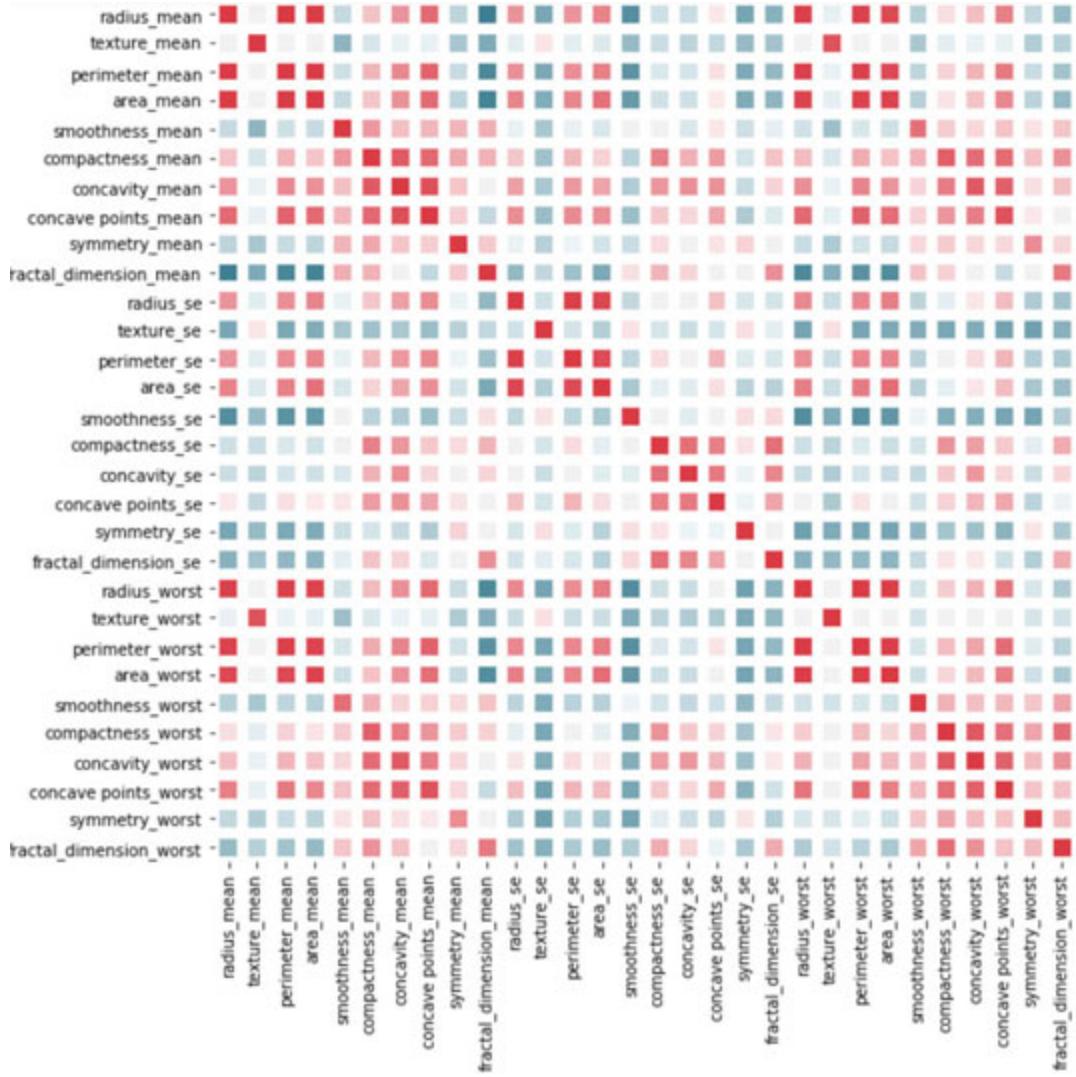
9. Our next step is to build a correlation matrix which will tell us about the relationship between the various attributes present in our dataset. It is important because many times, the attributes may be highly correlated and aren't useful for our analysis.

The correlation matrix, represented as a heatmap, allows us to eliminate the highly correlated variables and only keep the ones extremely useful.

The following code snippet will calculate the correlation for our dataset attributes and create a heatmap to represent the varying degrees of correlation visually:

```
corr = df.corr()  
#Generate a custom diverging colormap  
cmap = sns.diverging_palette(220, 10, as_cmap=True)  
plt.figure(figsize=(12, 20))  
sns.heatmap(corr, cmap=cmap, square=True, linewidths=6)
```

The following image represents the heatmap for the correlation between the different attributes present in our dataset:



**Figure 6.10:** Correlation matrix for the dataset

10. Looking at the above correlation matrix, all the square boxes that are dark red show that the variable (row and column ones) are highly correlated.

If we try to notice a few patterns here:

- We can see that all of the columns which have `_worst` in their titles are highly correlated to the `_mean` columns since the worst value would also be counted while calculating the mean.
- The `perimeter_mean` and `area_mean` is highly correlated to `radius_mean`. That's because both the perimeter and the area values are highly likely to use the radius variable.

- The compactness\_se and the concave points\_se are highly correlated to compactness\_mean and concave\_means.

Based on the above observations, we will drop the columns that are duplicates, that is, highly correlated and keep only one of the columns:

```
cols_drop =
['perimeter_mean', 'area_mean', 'compactness_mean', 'concave
points_mean', 'radius_se', 'perimeter_se', 'radius_worst', 'per
imeter_worst', 'compactness_worst', 'concave
points_worst', 'compactness_se', 'concave
points_se', 'texture_worst', 'area_worst']
df = df.drop(cols_drop, axis = 1 )
```

11. Now we need to encode our target variable in order to convert it to a numeric variable from a categorical variable.

```
le = LabelEncoder()
df["diagnosis"] = le.fit_transform(df["diagnosis"])

y = df["diagnosis"]
x = df.drop(columns=['diagnosis'])
```

12. We will split the data into training and test data and fit our training data to a logistic regression model.

```
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.33, random_state=42)

#create an instance and fit the model
logit = LogisticRegression()
logit.fit(x_train, y_train)
```

13. We use the above fit model to predict the possible Y variable (benign or malign breast cancer) for our test data.

We are also using several accuracy metrics mentioned in this book to understand the performance of our model.

```
y_predict = logit.predict(x_test)
print(classification_report(y_test,y_predict))
```

**Output:**

	<b>precision</b>	<b>recall</b>	<b>f1-score</b>	<b>support</b>
0	0.97	0.96	0.97	121
1	0.93	0.96	0.94	67
			<b>accuracy</b>	0.96
			<b>macro avg</b>	0.95
			<b>weighted avg</b>	0.96

14. Creating a confusion matrix. The cumulative number of right and wrong predictions are present in the confusion matrix. They are summarized with count values and broken down by each class.

The first entry is the True Positives (Target variable is positive and predicted correctly), the next entry to the right is False Negatives (Target variable is positive but predicted as negative), the entry below true positives is False Positives (Target variable is negative but predicted as positive), and the entry next to that is True Negatives (Target variable is negative and predicted correctly).

Here is the output of the confusion matrix:

```
print(confusion_matrix(y_test, Y_predict))

[[116  5]
 [2  65]]
```

The above steps are an end-to-end implementation of the logistic regression algorithm given a dataset with two classes that are opposite. In the next section, we will look at the summary of our case study and get some insights into the model's prediction statistics.

## Summary

The accuracy of our model is 96%, which is a perfect percentage to achieve. You can also observe from the confusion matrix that the number of False Negatives (5) and the number of False Positives (2) are quite less; therefore, this is a good model for our dataset.

The above case study shows us how we can use logistic regression to predict the target variable based on just one given independent variable. It is

essential to understand that **Exploratory Data Analysis or EDA** plays a vital role in regression analysis as it takes you through the various steps of cleaning and pre-processing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data, the attribute strength, and so on to revamp your model.

## Case Study - II

This case study will take us through the various steps that are involved in building a logistic regression model. In this section, we will be focusing on datasets that have different types of independent variables and one dependent variable, which is a binary variable (Yes/No).

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been taken from Kaggle. This dataset has been open-sourced to the community to learn and perform regression analysis.

The data were collected and made available by the "National Institute of Diabetes and Digestive and Kidney Diseases" as part of the Pima Indians Diabetes Database. Several constraints were placed on the selection of these instances from a more extensive database. In particular, all patients here belong to the Pima Indian heritage (a subgroup of Native Americans) and are females of ages 21 and above.

The following attributes were collected for the above dataset,

## Attribute information

- Pregnancies (number of pregnancies)
- Glucose (measured value)

- BloodPressure (measured value)
- SkinThickness (measured value)
- Insulin (measured value)
- BMI (measured value)
- DiabetesPedigreeFunction (measured value)
- Age
- Outcome (1 - Yes / 0 - No)

In the next section, we will look at the step-by-step Python code that will perform the regression analysis on our chosen dataset.

## Python code and step-by-step regression analysis

This case study is similar to our case study - I but uses a different dataset with different types of attributes. The step-by-step regression analysis will help us understand the sequence of different actions that need to be performed when a raw dataset is received for the first time.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
%matplotlib inline
```

2. Read the data (which is stored in the CSV format) using `python-pandas` and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `diabetes.csv`. I have stored the datasets under the `/LogisticRegression` folder under the `/Datasets` folder on my local system.

```
df =
pd.read_csv('Datasets/LogisticRegression/diabetes2.csv')
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

The following image captures the output of `df.head()`:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	
0	6	148	72	35	0	33.6		0.627	50	1
1	1	85	66	29	0	26.6		0.351	31	0
2	8	183	64	0	0	23.3		0.672	32	1
3	1	89	66	23	94	28.1		0.167	21	0
4	0	137	40	35	168	43.1		2.288	33	1

*Figure 6.11: The output of df.head()*

4. Analyze the data by looking at the data types, and if the columns' observations are numerical, you can use the `describe` function to understand more about the data. You could also store the column names in an array to avoid typing them in the future if the column names are lengthy.

The following code snippet captures the output of `df.describe()`, which is used to describe the statistical parameters related to a dataset:

```
df.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578		0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160		0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000		0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000		0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000		0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000		0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000		2.420000	81.000000	1.000000

*Figure 6.12: The statistical output from df.describe()*

The following code snippet gives us an insight into the different data types that are present in our dataset. To understand if normalization is needed in a dataset or not, it is important to look at the different data types of attributes even when all of them are numerical:

```
df.dtypes
```

Output:

```
Pregnancies          int64
Glucose              int64
BloodPressure        int64
SkinThickness        int64
Insulin              int64
BMI                  float64
DiabetesPedigreeFunction   float64
Age                  int64
Outcome              int64
dtype: object
```

5. To look at what that column contains and to check if any missing or null values are present in the dataset, we use the `info()` method on our DataFrame.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
Pregnancies          768 non-null int64
Glucose              768 non-null int64
BloodPressure        768 non-null int64
SkinThickness        768 non-null int64
Insulin              768 non-null int64
BMI                  768 non-null float64
DiabetesPedigreeFunction   768 non-null float64
Age                  768 non-null int64
Outcome              768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

6. We will now visualize the data to look at how the data is distributed across our dataset.

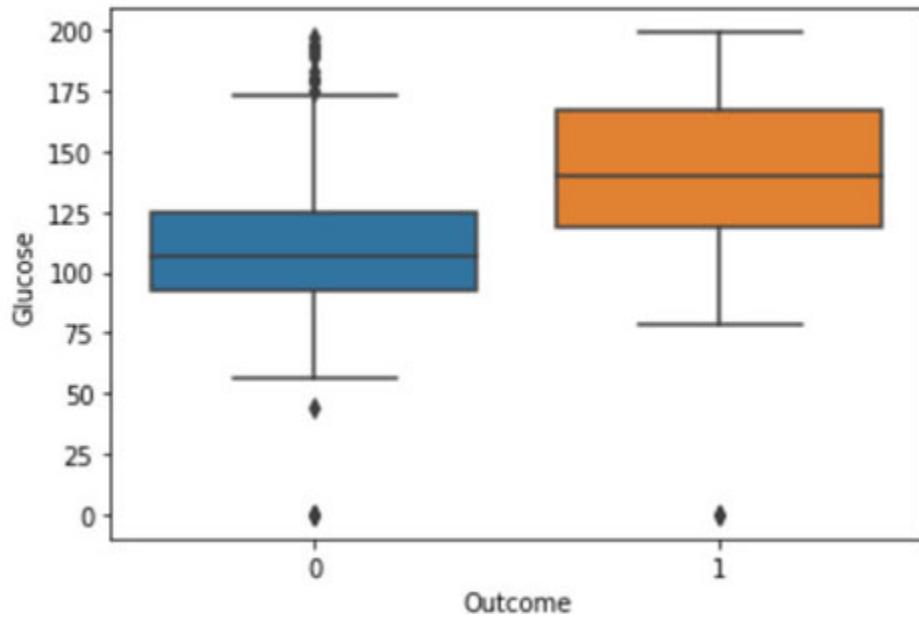
We will first create box plots to understand how different attributes are distributed for the given `Outcome` variable. Box plots will help us understand the mathematical distribution of data in our dataset.

Here, we will be creating three boxplots just to give you an understanding of how boxplots look like.

```
sns.boxplot(df.Outcome, df.Glucose)
```

Here is the image capturing the boxplot between `Outcome` and `Glucose` attributes:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118df47f0>
```



*Figure 6.13: Box plot between Outcome and Glucose*

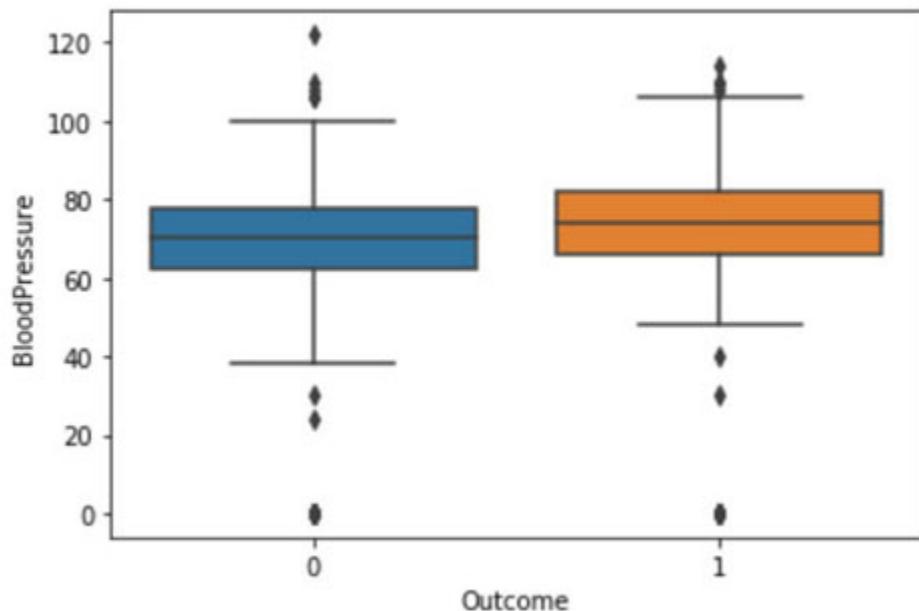
For the above figure, we can observe that for `Outcome = 1`, the median of `Glucose` is slightly higher than the median of `Glucose` for `Outcome = 0`.

In the next code snippet, we will look at the box plot between `Outcome` and `BloodPressure` attributes:

```
sns.boxplot(df.Outcome, df.BloodPressure)
```

Here is the image capturing the boxplot between `Outcome` and `BloodPressure` attributes:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118f11ba8>
```



*Figure 6.14: Box plot between Outcome and BloodPressure*

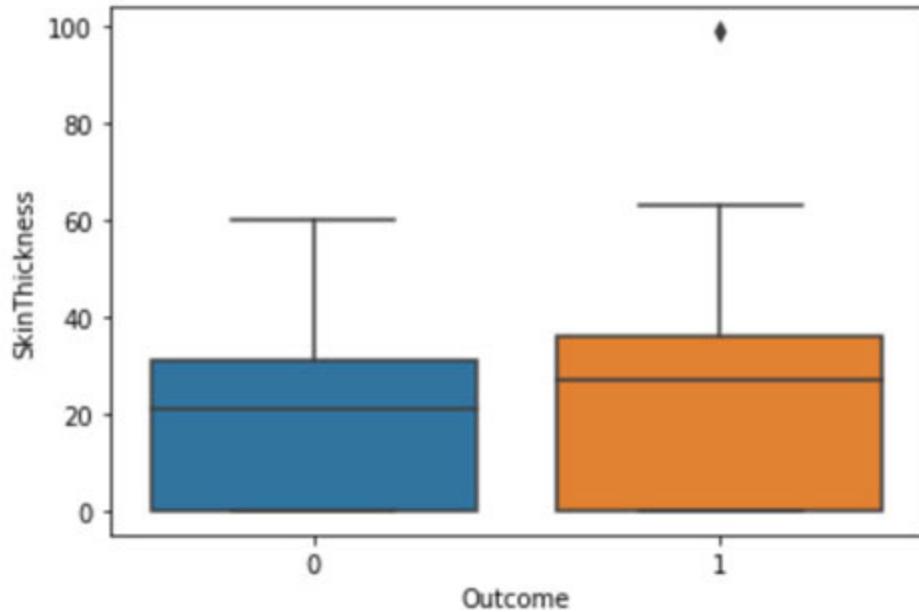
For the above figure, we can observe that for `Outcome = 1`, the median of `BloodPressure` is slightly higher than the median of `BloodPressure` for `Outcome = 0`.

In this code snippet, we will look at the box plot between `Outcome` and `SkinThickness` attributes:

```
sns.boxplot(df.Outcome, df.SkinThickness)
```

Here is the image capturing the boxplot between `Outcome` and `SkinThickness` attributes:

```
<matplotlib.axes._subplots.AxesSubplot at 0x106d367f0>
```



*Figure 6.15: Box plot between Outcome and SkinThickness*

For the above figure, we can observe that for `Outcome = 1`, the median of `SkinThickness` is almost the same as the median of `SkinThickness` for `Outcome = 0`.

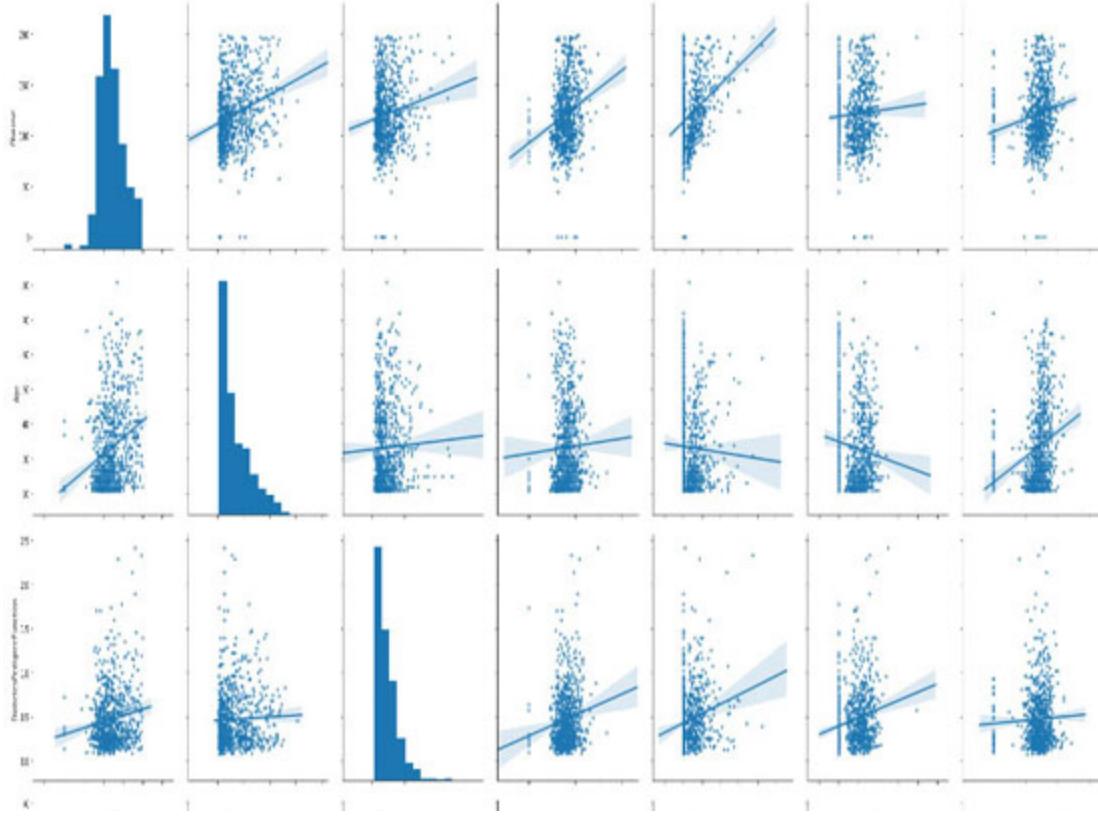
7. We will also create pair plots of some selected columns (you can choose the ones of your choice) using the seaborn (`sns`) library. Pair plots are useful when you want to visualize the relationship between two variables.

You can create a pair of plots in such a way that you can visualize the relationship between each pair of attributes. The image here doesn't display the complete result but a part of the resulting pair plots generated:

```
df_n=df[['Glucose','Age','DiabetesPedigreeFunction','BMI','  
Insulin','SkinThickness','BloodPressure']]  
sns.pairplot(df_n, height=4, kind="reg", markers=".")
```

The following image gives us a glimpse of the pair of plots between all the attributes present in our attributes. The following pair plot is not a complete screenshot but a partial screenshot of the first output:

```
<seaborn.axisgrid.PairGrid at 0x119103860>
```



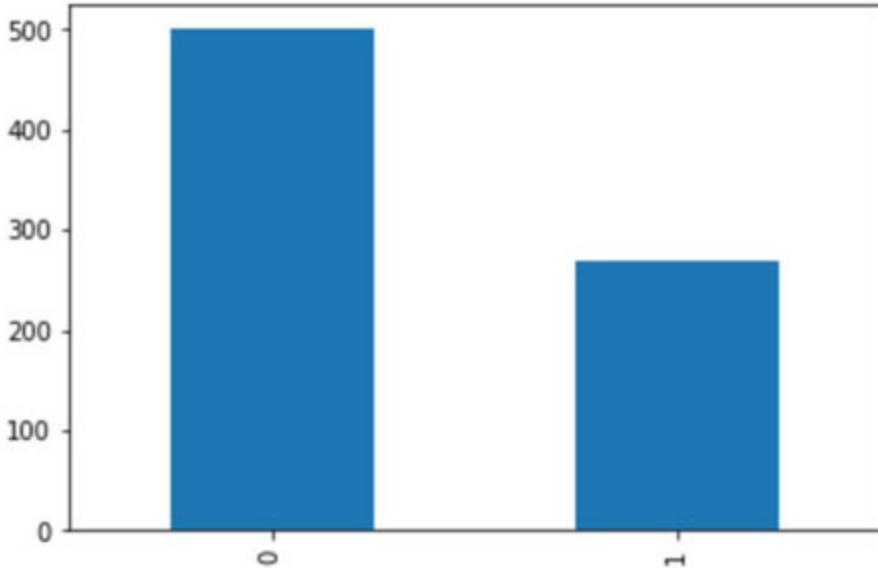
**Figure 6.16:** Pair plots between all the pairs of attributes

The following code snippet plots a bar plot of the `Outcome` variable using the value counts since `Outcome` is a categorical attribute:

```
df['Outcome'].value_counts().plot.bar()
```

The following figure captures the bar plot that shows us the distribution of the `Outcome` variable in our dataset:

```
<matplotlib.axes._subplots.AxesSubplot at 0x11f060a90>
```



**Figure 6.17:** Bar Graph to show the distribution of the target variable 'Outcome'

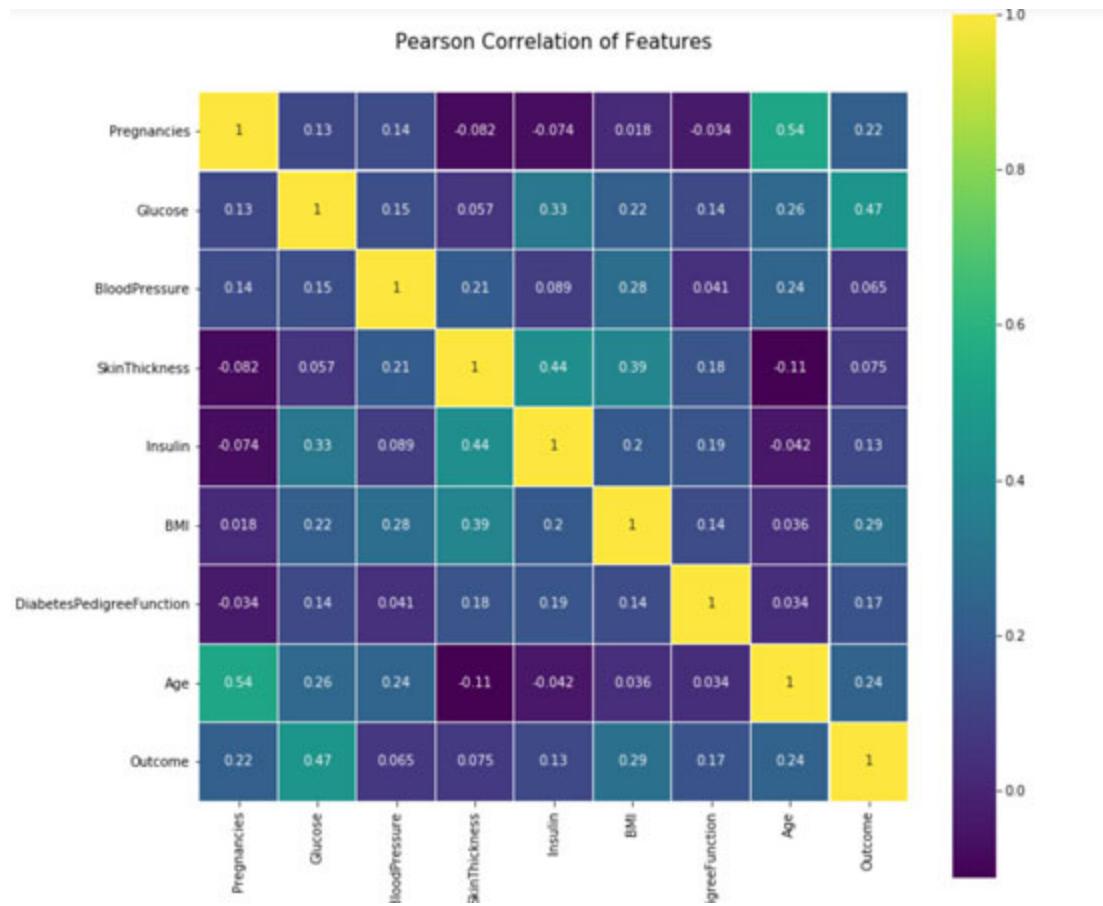
- Our next step is to build a correlation matrix which will tell us about the relationship between the various attributes present in our dataset. It is important because many times, the attributes may be highly correlated and aren't useful for our analysis.

The correlation matrix, represented as a heatmap, allows us to eliminate the highly correlated variables and only keep the ones extremely useful.

The following code snippet will calculate the correlation for our dataset attributes and create a heatmap to represent the varying degrees of correlation visually:

```
corr = df.corr()
plt.figure(figsize=(12,12))
plt.title('Pearson Correlation of Features', y=1.05,
size=15)
sns.heatmap(corr, linewidths=0.1,vmax=1.0, square=True,
cmap=colormap, linecolor='white',
annot=True)
```

The following image represents the heatmap for the correlation between the different attributes present in our dataset:



*Figure 6.18: Correlation heatmap for our dataset*

9. As we can see in [Figure 6.18](#), the correlation values are not too high between any of the attributes, unlike our previous case study, therefore, we will not be eliminating any columns from our dataset.

We will split the data into training and test data and fit our training data to a logistic regression model:

```
y = df['Outcome']
X = df.drop(columns=['Outcome'])
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, random_state=42)

#create an instance and fit the model
logit = LogisticRegression()
logit.fit(X_train, y_train)
```

**Output:**

```

LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True,
    intercept_scaling=1, l1_ratio=None, max_iter=100,
    multi_class='warn', n_jobs=None, penalty='l2',
    random_state=None, solver='warn', tol=0.0001,
    verbose=0,
    warm_start=False)

```

10. We use the above fit model to predict the possible Y variable (benign or malign breast cancer) for our test data.

We are also using several accuracy metrics mentioned in this book to understand the performance of our model:

```

Y_predict = logit.predict(X_test)
print(classification_report(y_test,Y_predict))

```

**Output:**

	precision	recall	f1-score	support
0	0.81	0.85	0.83	168
1	0.67	0.62	0.64	86
<b>accuracy</b>			0.77	254
<b>macro avg</b>	0.74	0.73	0.74	254
<b>weighted avg</b>	0.76	0.77	0.77	254

11. Creating a confusion matrix. The cumulative number of right and wrong predictions are present in the confusion matrix. They are summarized with count values and broken down by each class.

The first entry is the True Positives (Target variable is positive and predicted correctly), the next entry to the right is False Negatives (Target variable is positive but predicted as negative), the entry below true positives is False Positives (Target variable is negative but predicted as positive) and the entry next to that is True Negatives (Target variable is negative and predicted correctly):

```

print(confusion_matrix(y_test, Y_predict))

```

**Output:**

```

[[142 26]

```

[ 33 53 ]

The step-by-step analysis helps us understand the behavior of data at various points in time. This analysis will give us insights into the distribution of our data and the attributes that contribute significantly to the data decisions.

## Summary

The accuracy of our model is 77%, which is a reasonably good percentage to achieve. You can also observe from the confusion matrix that the number of False Negatives (26) and the number of False Positives (33) are not quite less; therefore, the model may need a little bit more tuning or more data.

The above case study shows us how we can use logistic regression to predict the target variable based on just one given independent variable. It is essential to understand that **Exploratory Data Analysis or EDA** plays a vital role in regression analysis as it takes you through the various steps of cleaning and pre-processing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data, the attribute strength, and so on, to revamp your model.

## Practical examples of logistic regression

Logistic regression is one of the most widely used machine learning algorithms. There are several practical examples of logistic regression, some of which are:

- Predicting if an analog sensor reading is normal or abnormal
- Being able to predict the weather
- Medical diagnosis

It is worthwhile to explore several practical problems, especially classification ones, to find a solution by applying logistic regression. Several customers facing research on online patterns and customer behavior can also be done using logistic regression analysis.

There are also two more categories of logistic regression:

- **multinomial:** The target variable can be more than 2; the values of the target variable cannot be ordered in nature (that is, none of them should be related). For example: "type A" versus "type B" versus "type C."
- **ordinal:** It deals with target variables with ordered categories. For example, the quality of a material can be categorized as: "very poor," "poor," "good," "very good." Here, each category can be given a score like 0, 1, 2, 3.

The above two categories also fall under logistic regression analysis and are used to explore the several use cases that fall under the 'classification' umbrella in the industry.

## Conclusion

Logistic regression has several use cases in the industry, especially when it comes to consumer-facing markets since most of the time, the consumer wants to make a Yes/No choice while buying a product. Several decision-making strategies in an industry could also benefit from an algorithm like Logistic Regression, which can accurately predict a binary/multinomial target variable.

Our next chapter will give you a sneak peek into support vector machines, which are used in predictive modeling when the feature set contains a large number of features. Our next chapter will also have some interesting case studies for you to look into and practice machine learning codes hands on.

## Quiz

1. True-False: Is it possible to design a logistic regression algorithm using a neural network algorithm?
2. The logit function (given as  $l(x)$ ) is the log of odds function. What could be the range of logit function in the domain  $x=[0,1]$ ?
3. What would do if you want to train logistic regression on the same data that will take less time as well as give a comparatively similar accuracy (may not be the same)?

4. What are the log odds?
5. What are the false positives and false negatives?

# CHAPTER 7

## A Sneak Peek into the Working of Support Vector Machines

### Introduction

Imagine a situation where a cat is groomed to look like a dog; a human mind can easily be confused whether it's a dog or a cat. But assuming a human mind can eventually detect that it's a cat based on its characteristics, this problem would be extremely complicated to solve for a machine.

It is where Support Vector Machines play a crucial role. **Support Vector Machines**, also known as **SVMs**, is best suited for scenarios where there is a need to examine the dataset exceptionally carefully. It works best in cases when two classes need to be segregated, considering the extreme characteristics of both the classes.

In simple terms, we can also say that SVMs act as a frontier that can segregate two classes in the best possible manner. The frontier that separates two classes is also known as the decision boundary, and SVM helps us get an optimal decision boundary.

### Structure

- Need of an Optimal Decision Boundary
- Workings of the SVM
- Maximal Margin Classifier

### Objectives

- The primary objective of this chapter is to understand the working of Support Vector Machine algorithms

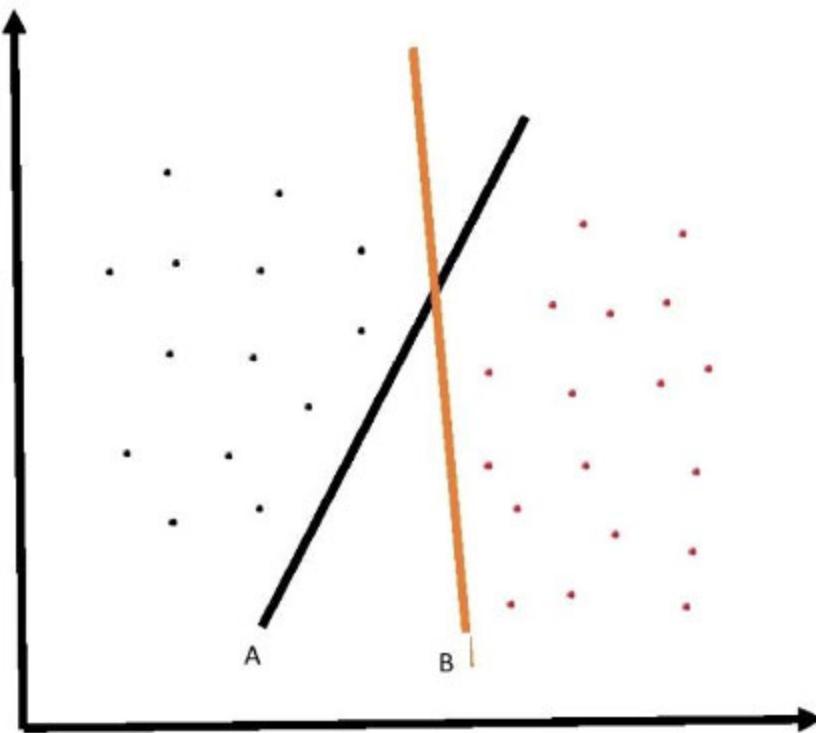
- Get a more in-depth insight into what are the different support vector kernels and what does each kernel signify in a real-time context

## Why do we need an optimal decision boundary?

An optimal decision boundary is needed for us to be able to classify the unknown data/new data correctly. In case of a decision boundary that is leaning towards Class A (example: Dog) or Class B (example: Cat), the new data classification will also be biased towards Class A or Class B, where the decision boundary is leaning towards.

An optimal decision boundary helps split the two classes, considering the extreme points in both the classes. It uses the data points that are closest to both the classes, also known as **Support Vectors**. The optimal decision boundary is also known as a **Hyperplane** in the machine learning context.

Here is an image that depicts the two different decision boundaries that can be drawn for a particular data distribution:



*Figure 7.1: Decision boundaries for any two classes*

As you can see in [Figure 7.1](#), decision boundaries can be of multiple kinds separating two classes, but it is the support vector machine that gives us the

optimal decision boundary, let us understand from the workings as to how it does that.

## Workings of the SVM

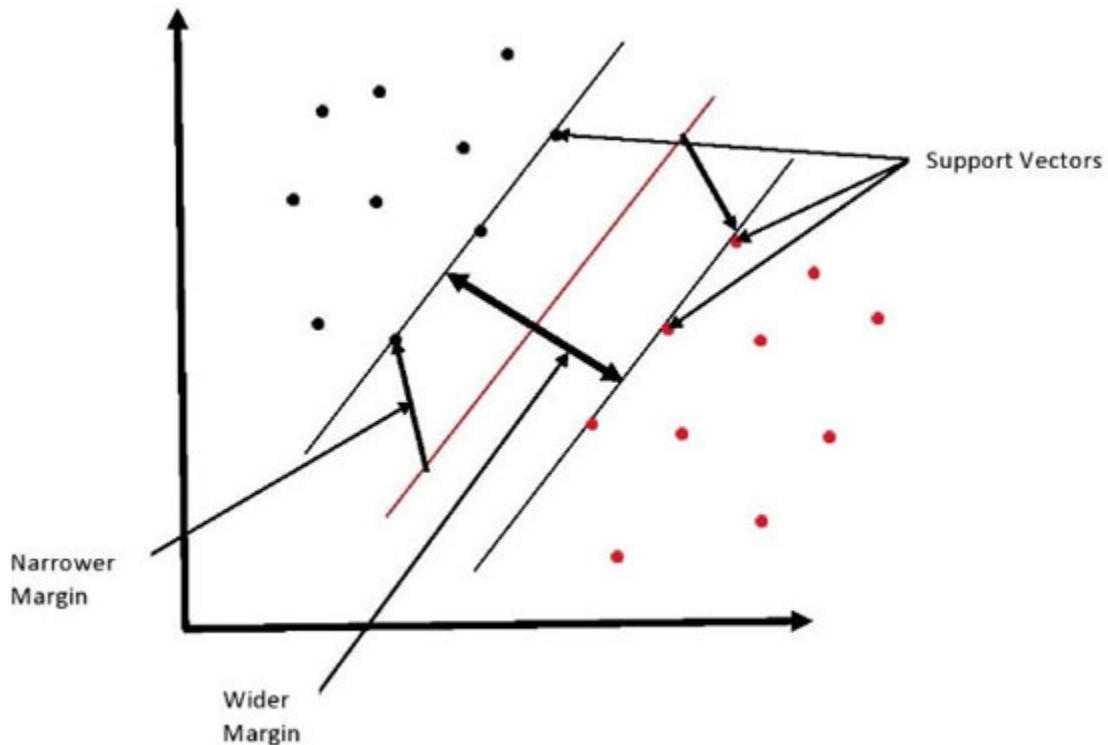
SVMs, as we already mentioned, introduce linear discriminators/hyperplanes, which maximizes the distance between the hyperplane and the “difficult points” close to the hyperplane. As you can see in [Figure 7.2](#), the red hyperplane is one decision boundary separating the two classes (one class is represented by the black dots and another by the red dots).

But if you notice the distance between the boundary points for each class and the red decision boundary, there's a narrow margin, and therefore, the observations can only fall on either side of the boundary. On the contrary, if you notice the two black lines touching the boundary points for each class, the distance between them is much higher, giving us a higher margin between the points.

The red dots and the black dots at the extreme boundaries are known as the support vectors, and those observations act as the crucial points for this algorithm.

The support vectors are nothing but observations from the training dataset that act as the inputs to the decision function. Solving SVMs is a quadratic programming problem statement. Instead of delving deeper into the statistics, we will look at the several successful applications of SVM in the industry.

The following image depicts a hyperplane with narrow and wide margins along with the support vectors for a data distribution:



*Figure 7.2: Hyperplane in a support vector machine*

From the above distribution, we get a picture of how support vector machines would look like for random data distribution. In the next section, we will look at the various applications of support vector machines.

**Sharp points are considered as points that are difficult to classify accurately into either of the classes.**

## Applications of support vector machines

SVM is a versatile machine learning algorithm and has several applications in the industry. Some of the most common application of SVMs are speech and text recognition, face detection, image processing, and many more. We will look at some of the applications of SVMs in the following section:

### Face detection

In recent times, face recognition, object detection has become widespread use cases and have gained a lot of attention in the industry. Apart from the daily usage of these technologies for unlocking our phones, there is a vast

scope of research when it comes to these technologies, for example, in the space of self-driving cars.

All these pattern recognition problem statements fall under the classification category, and support vector machines are applied vastly, to classify the objects. Because SVMs offer us the ability to perform classification even when the features do not follow a linear pattern, they are one of the best choices to use in this scenario.

## **Text categorization/classification**

Handwriting classification and categorization, conversion of handwritten text images into digital text, and so on are becoming extremely popular use cases in our industry. If we look at industries like the banking or insurance sector, there is a lot of KYC that still happens using manual documentation, but as the banks and the insurance sector is moving towards digital records, text classification has become an important tool in this era.

SVM are being extensively used for text classification and have proven to be hugely successful and efficient in this process.

## **Image classification**

From intelligence agencies like CBI and FBI to governments of countries like China, everyone is intrigued by the benefits of facial recognition. Facial recognition is a complicated process and involves a lot of parameters to process to reach a good accuracy percentage.

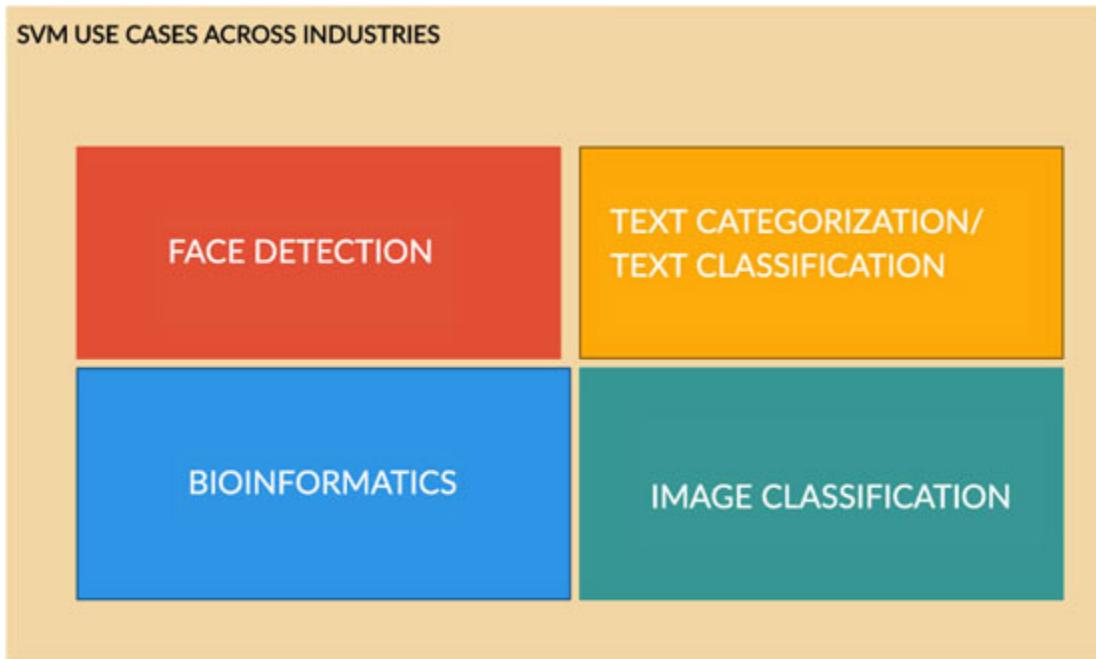
Image classification involves multiple features in the feature vector in the form of pixels. It is where SVM can come in and help with the classification. SVMs are equipped to deal with large feature vectors during classification. It is also a plus that SVMs have kernels that easily support non-linear patterns in the feature vectors.

## **Bioinformatics**

As we have seen above, SVM can classify with lesser prediction errors as compared to other models like neural networks, logistic regression, and many more when the feature set is large. In the field of genomics and chemogenomics, support vector machines are being highly considered for classifying and identifying components in protein structures.

In the case of current research, SVMs have been highly valuable in providing insights into the field of bioinformatics.

Here is an image that gives us an idea of the various SVM use cases across industries:



*Figure 7.3: SVM use cases*

The use cases of SVM are not limited to the above, but we wanted to provide a glimpse of the various industries/real-time problems where SVM can provide significant solutions.

## **Maximal margin classifier**

A maximal margin classifier is used to split the training data in such a manner that the classifier is equidistant from both the output classes. It is essential to create a classifier that is unbiased to either of the classes while training the model. As our classifier maintains an equal distance from both classes, it is less biased.

As we can observe in [Figure 7.2](#), the distance between the closest data points (also known as support vectors) and the line is known as the margin. To achieve a bias-free training, the best or optimal line that fits the above problem statement is the one whose margin is the maximum.

The margin can be calculated as the perpendicular distance from the line to the support vectors/closest points on either side. As we have mentioned before, in a support vector machine algorithm, only the support vectors are considered as significant/relevant points in the calculation.

The selected support vectors are the ones that support the hyperplane construction and define the classifier itself.

**NOTE: One of the points to notice in the case of a maximal margin classifier is that the support vectors are chosen more often as rigid/stationary points, which does not allow much flexibility while training the model.**

## Soft margin classifier

As we saw in the above section, the margin is the distance between our hyperplane and the support vectors, which are the points that are chosen from each class that are the closest to the hyperplane.

In the case of maximal margin classifiers, these support vector points that are chosen are incredibly rigid and only support a strict ‘Class A’ or ‘Class B’ kind of classification and does not account for any anomalies (example: points that could potentially fall on the classifier line itself).

It is where the concept of soft margin classifier comes in. To create a generic training model, the limitations on maximizing the margin should be relaxed. This kind of criterion relaxation allows some of the points to move around on either side of the line.

The above can be accomplished by introducing an additional set of variables that give some room for flexibility while defining the margins. These sets of variables are introduced for each dimension. Also known as slack variables, these coefficients tend to increase the complexity of the model as there are more parameters to fit while modeling creation.

A tuning parameter C is introduced to indicate the magnitude of the wiggle room in the classifier margin; that is, it lets us know as to what extent the points can be expected to violate the classification parameter.

The value of C starts from 0, which would mean that no wiggle room and it’s a Maximal Marginal Hyperplane. A higher value of C indicates more flexibility for the points to be classified in either of the classes.

## Kernels

SVM kernels are widely used in machine learning as they provide a method to solve various machine learning problems using different types of equations that are suitable for data distribution. In the upcoming section, we will get a more in-depth insight into the SVM Kernels and their different applications.

## Introduction to SVM kernels

One of the most commonly used kernels in SVM are linear kernels. Linear kernels are used most often when the data is linearly separable. In cases where the data is not linearly separable, we may need to use other kernels, which are polynomial or radial. In this section, we will look at the different types of kernels supported in SVM.

## Linear kernels

As we mentioned in the previous section, linear kernels are used in situations when the data is linearly separable, that is, we can separate the two different classes present in the data with the help of a line.

The SVM kernel equations are the inner products of two observations present in our dataset. The inner product is the dot product of two vectors. For example, the inner product of the vectors [3, 6] and [2, 5] is  $3*2 + 6*5$  or 36. The above equation represents the same in a matrix format.

Here is the SVM equation that predicts the class of the target variable using a linear kernel:

The equation for predicting a new input using the dot product between the input ( $x$ ) and each support vector ( $xi$ ) is calculated as follows:

$$f(x) = B0 + \sum(ai * (x, xi))$$

It is an equation that involves calculating the inner products of a new input vector ( $x$ ) with all support vectors in training data. The coefficients  $B0$  and  $ai$  (for each input) must be estimated from the training data by the learning algorithm.

Linear SVM kernel is a parametric model, and its complexity remains the same as the size of the training data increases, whereas the other kernels

tend to increase in complexity as the size of the training data increases.

## Polynomial kernels

Polynomial kernels are used to predict the target variable in situations where the data is not linearly separable. In polynomial kernels, we calculate the dot product by increasing the power of the kernel.

Here is the kernel equation for a polynomial kernel:

$$K(X1, X2) = (a + X1'X2)b$$

a is a constant variable, and b is the degree of the polynomial.

We use the kernel trick to convert a linear model into a non-linear model. It replaces the linear predictor by the kernel function we decide to use.

Polynomial kernels can be of any degree, and they help create various polynomial separations amongst the data according to the spread of the classes in the dataset. Polynomial kernels calculate the separation in a higher dimension, which is the kernel trick.

One of the most popular forms of polynomial kernels is quadratic kernels because they generalize well. The higher-order kernels some time tend to overfit the training dataset. Order two/quadratic kernels are mostly used in speech recognition.

## Radial kernels

Radial kernels are an essential feature of SVMs as they help you separate classes that are radially separable. There are not too many machine learning algorithms that work in this fashion, and hence, the radial kernels of SVMs are unique.

Radial kernels, also known as **RBF (Radial Basis Function)** or Gaussian kernels, are commonly used in computer vision applications. The RBF kernel equation is slightly more complicated and is represented as:

$$K(X1, X2) = \exp (||X1 - X2||^2 / 2^2)$$

The Gaussian kernel is an exponential function in the input feature space; it is a decaying function, which attains maxima around the support vectors and decays in a radial fashion creating the radial separation for us.

The RBF kernel SVM decision region is linear. What the kernel does is to create non-linear combinations of your features to uplift your samples onto a higher-dimensional feature space where you can use a linear decision boundary to separate your classes.

RBF and polynomial kernels are used in practice when the data is not linearly separable, but we need to be careful with these kernels as it might be easy to overfit the data using these kernels.

In the next section, we will look at various case studies on SVM that will solve a real-time problem using support vectors.

## Case study - I

This case study will take us through the various steps that are involved in building a support vector machine model. In this section, we will be focusing on datasets that have different types of data distributions.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model. In the following case study, we will build a model to predict the class of an iris flower based on its feature values.

## About the data

The dataset that we are using for this case study is one of the most common datasets used in machine learning. The dataset is called the iris dataset and describes the various types of iris flowers and their features. There are various classes of iris flowers that have different sepal and petal features, based on which we will be writing our classifying algorithm.

The attributes in our dataset define the various features of an iris flower. Here is more information about what the attributes are.

### **Attribute information:**

- sepal-length
- sepal-width
- petal-length
- petal-width

- class

The following section will take us through a step-by-step process of how to write Python code to solve a machine learning problem using SVM.

## Python code and step-by-step regression analysis

The following steps define code snippets and their outputs for a dataset that is being analyzed and worked upon using machine learning techniques. You can use the code and type it on your Jupyter Notebook/Google Colab or an editor on your local computer.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn import svm
%matplotlib inline
```

2. Read the data (which is stored in the .data format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `iris.data`. I have stored the datasets under the /SVM folder under the /Datasets folder on my local system:

```
df = pd.read_csv('Datasets/SVM/iris.data')
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame.

```
df.head()
```

Here is a visual output of the `df.head()` command:

```

<bound method NDFrame.head of      5.1   3.5   1.4   0.2      Iris-setosa
0    4.9   3.0   1.4   0.2      Iris-setosa
1    4.7   3.2   1.3   0.2      Iris-setosa
2    4.6   3.1   1.5   0.2      Iris-setosa
3    5.0   3.6   1.4   0.2      Iris-setosa
4    5.4   3.9   1.7   0.4      Iris-setosa
5    4.6   3.4   1.4   0.3      Iris-setosa
6    5.0   3.4   1.5   0.2      Iris-setosa
7    4.4   2.9   1.4   0.2      Iris-setosa
8    4.9   3.1   1.5   0.1      Iris-setosa
9    5.4   3.7   1.5   0.2      Iris-setosa
10   4.8   3.4   1.6   0.2      Iris-setosa
11   4.8   3.0   1.4   0.1      Iris-setosa
12   4.3   3.0   1.1   0.1      Iris-setosa
13   5.8   4.0   1.2   0.2      Iris-setosa
14   5.7   4.4   1.5   0.4      Iris-setosa
15   5.4   3.9   1.3   0.4      Iris-setosa
16   5.1   3.5   1.4   0.3      Iris-setosa
17   5.7   3.8   1.7   0.3      Iris-setosa
18   5.1   3.8   1.5   0.3      Iris-setosa
19   5.4   3.4   1.7   0.2      Iris-setosa
20   5.1   3.7   1.5   0.4      Iris-setosa
21   4.6   3.6   1.0   0.2      Iris-setosa
22   5.1   3.3   1.7   0.5      Iris-setosa
23   4.8   3.4   1.9   0.2      Iris-setosa
24   5.0   3.0   1.6   0.2      Iris-setosa

```

*Figure 7.4: Output of df.head()*

In the next step, we will take a closer look at the columns in this dataset.

- As we observed in Step 3, the column names were showing up as numbers; therefore, we take a closer look at what the column names are, in the default dataset:

```
df.columns
```

### Output:

```
Index(['5.1', '3.5', '1.4', '0.2', 'Iris-setosa'],
      dtype='object')
```

- In Step 4, we noticed that the column names were not appropriate and that the first row in the dataset is the first set of values. To make our data processing easier, we assign the column names to represent the values in each column:

```
df.columns = ['sepal-length', 'sepal-width', 'petal-length',
              'petal-width', 'class']
df.head
```

The output is shown in [\*Figure 7.5\*](#):

Here is the image that represents the `df.head()` output once the columns have been assigned appropriate names:

```

<bound method NDFrame.head of      sepal-length  sepal-width  petal-length  petal-width    class
0          4.9        3.0         1.4         0.2  Iris-setosa
1          4.7        3.2         1.3         0.2  Iris-setosa
2          4.6        3.1         1.5         0.2  Iris-setosa
3          5.0        3.6         1.4         0.2  Iris-setosa
4          5.4        3.9         1.7         0.4  Iris-setosa
5          4.6        3.4         1.4         0.3  Iris-setosa
6          5.0        3.4         1.5         0.2  Iris-setosa
7          4.4        2.9         1.4         0.2  Iris-setosa
8          4.9        3.1         1.5         0.1  Iris-setosa
9          5.4        3.7         1.5         0.2  Iris-setosa
10         4.8        3.4         1.6         0.2  Iris-setosa
11         4.8        3.0         1.4         0.1  Iris-setosa
12         4.3        3.0         1.1         0.1  Iris-setosa
13         5.8        4.0         1.2         0.2  Iris-setosa
14         5.7        4.4         1.5         0.4  Iris-setosa
15         5.4        3.9         1.3         0.4  Iris-setosa
16         5.1        3.5         1.4         0.3  Iris-setosa
17         5.7        3.8         1.7         0.3  Iris-setosa
18         5.1        3.8         1.5         0.3  Iris-setosa
19         5.4        3.4         1.7         0.2  Iris-setosa
20         5.1        3.7         1.5         0.4  Iris-setosa
21         4.6        3.6         1.0         0.2  Iris-setosa
22         5.1        3.3         1.7         0.5  Iris-setosa
23         4.8        3.4         1.9         0.2  Iris-setosa
24         5.0        3.0         1.6         0.2  Iris-setosa
25         5.0        3.4         1.6         0.4  Iris-setosa

```

*Figure 7.5: Output from df.head() after renaming the columns*

In our next step, we will look at the results of the `df.describe()` command.

6. We now use `df.describe()` to get the statistical insights into our dataset. The describe functions allows us to look at various values like mean, count, standard deviation, maximum of the values present in the iris dataset:

**`df.describe()`**

Here is the visual output of `df.describe()`:

	<b>sepal-length</b>	<b>sepal-width</b>	<b>petal-length</b>	<b>petal-width</b>
<b>count</b>	149.000000	149.000000	149.000000	149.000000
<b>mean</b>	5.848322	3.051007	3.774497	1.205369
<b>std</b>	0.828594	0.433499	1.759651	0.761292
<b>min</b>	4.300000	2.000000	1.000000	0.100000
<b>25%</b>	5.100000	2.800000	1.600000	0.300000
<b>50%</b>	5.800000	3.000000	4.400000	1.300000
<b>75%</b>	6.400000	3.300000	5.100000	1.800000
<b>max</b>	7.900000	4.400000	6.900000	2.500000

*Figure 7.6: Output of df.describe()*

In the next section, we will look at the different data visualization techniques for this dataset.

7. To visualize the data better, we will use specific plots to look at the relationship between the various attributes in our dataset.

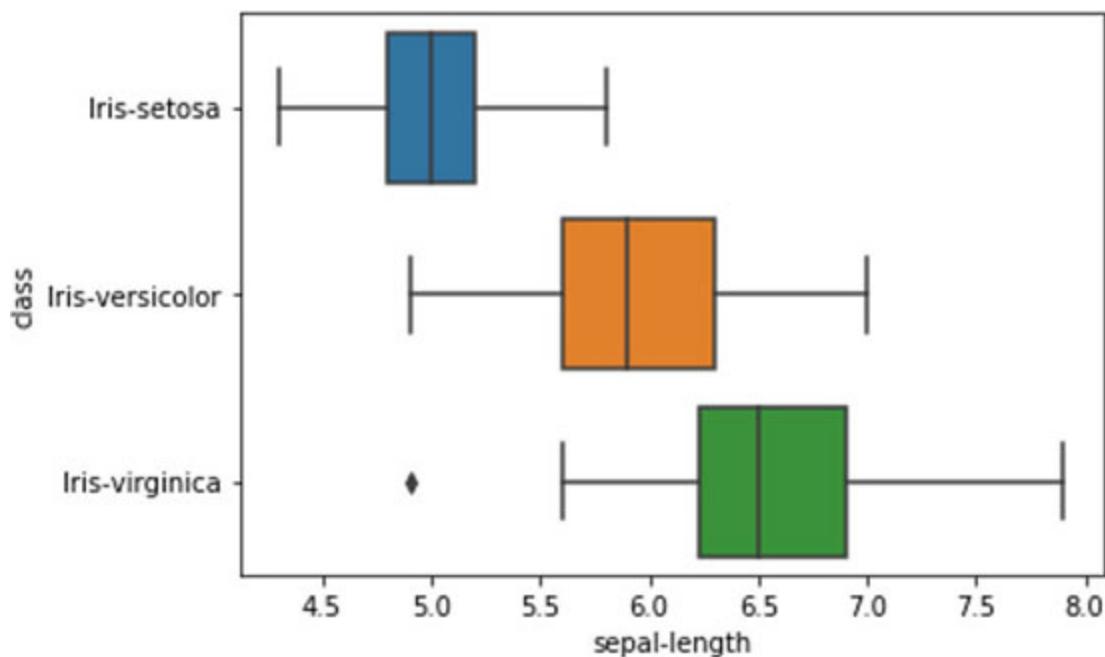
We will draw boxplots to look at the distribution of columns sepal-width and sepal-length for various classes.

The boxplots give us an understanding of how the minimum, maximum, and median values are distributed for the various classes.

The following code snippet will create boxplots between sepal-length and class variables:

```
sns.boxplot(df['sepal-length'],df['class'])
```

Here is the visual image representing the boxplot between sepal-length and class variables:

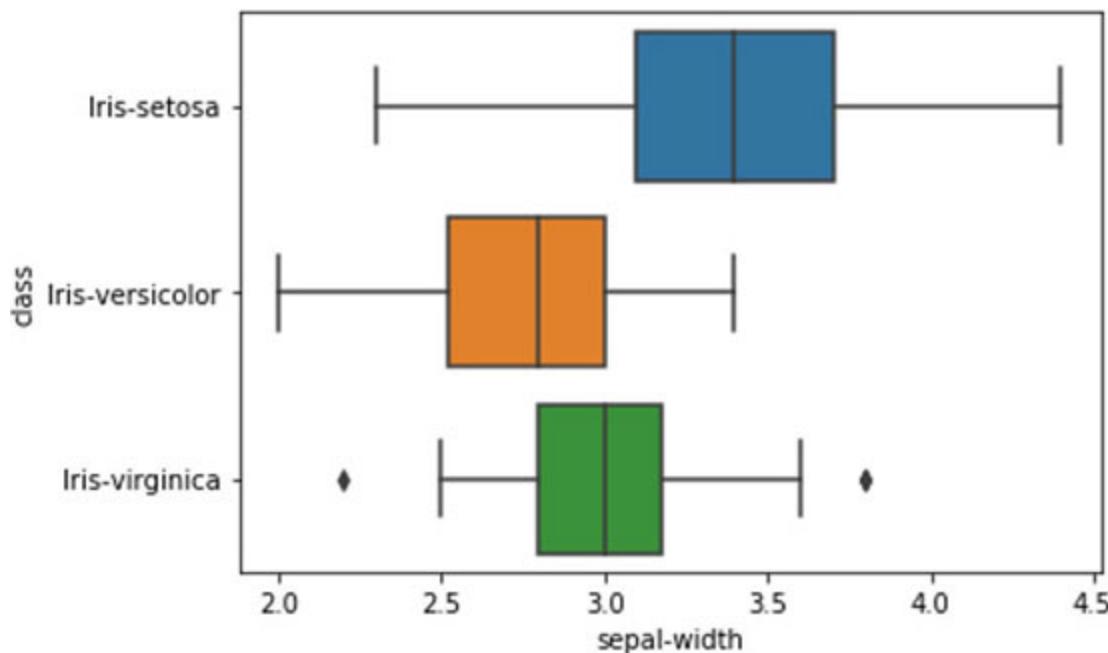


*Figure 7.7: Boxplot between the class and sepal-length*

The following code snippet will create boxplots between `sepal-width` and `class` variables:

```
sns.boxplot(df['sepal-width'], df['class'])
```

Here is the visual image representing the boxplot between `sepal-width` and `class` variables:



*Figure 7.8: Boxplot between the class and sepal-width*

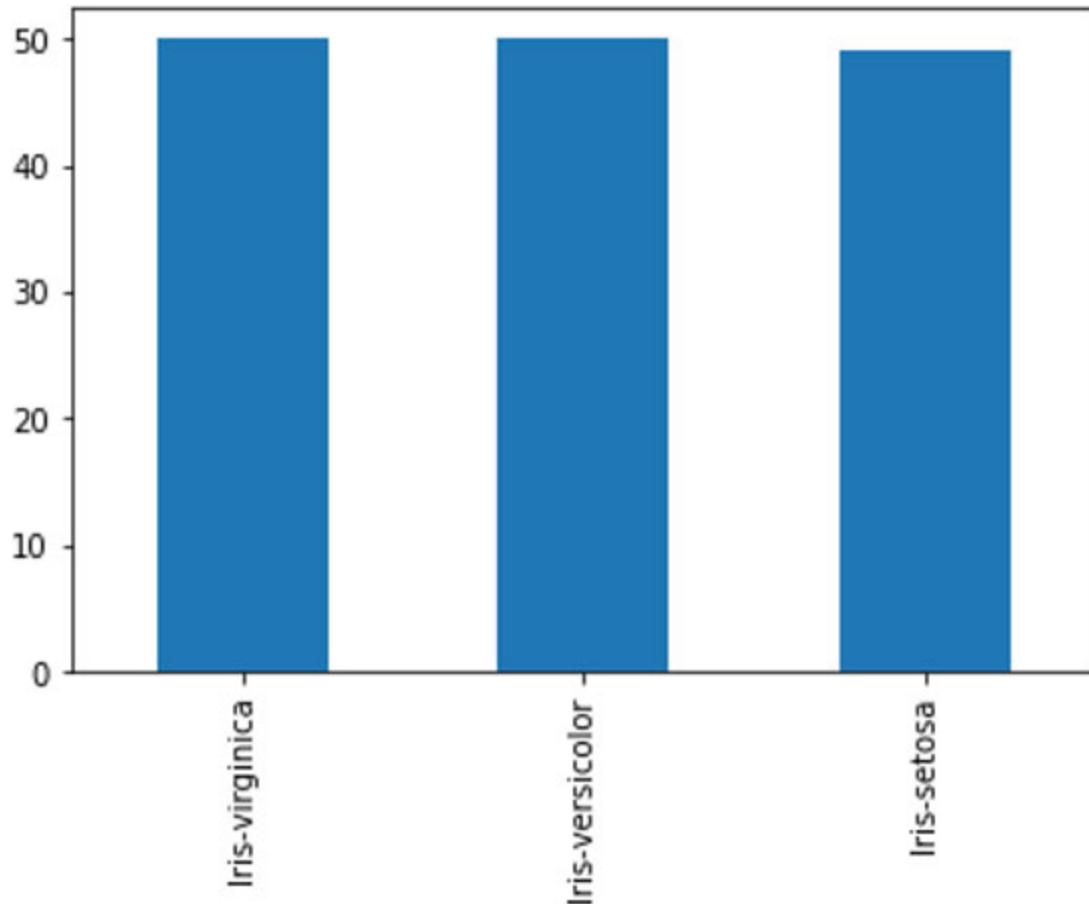
8. Our next step is to plot a bar graph that shows the distribution of the various classes in the dataset. This bar graph will help us understand if our prediction engine is likely to predict any bias in the classification results.

Our bar plot shows that all the classes are equally distributed in our current database, and hence, our model will likely have less bias.

The following code snippet will create a bar plot representing the distribution of the class variable in our dataset:

```
df['class'].value_counts().plot.bar()
```

Here is the image representing the bar plot of the various classes present in our dataset:



*Figure 7.9: Bar plot depicting the distribution of the target variable*

9. We will now separate our independent variable and dependent variables into separate columns to feed them to our training model. Our target variable to predict here is the ‘class’. We will drop the ‘class’ column from the features list:

```
y = df['class']
x = df.drop(columns=['class'])
```

10. We will split the data into training and test data and build an SVM model using the linear kernel for our data:

```
x_train, x_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, random_state=42)
model_svm = svm.SVC(kernel='linear')
```

11. We will fit the linear kernel SVM model to our training data:

```
model_svm.fit(x_train, y_train)
```

#### Output:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3,
gamma='auto_deprecated',
kernel='linear', max_iter=-1, probability=False,
random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

12. We use the above fit model to predict the possible  $y$  variable, also known as the class variable in our case. We are also using several accuracy metrics mentioned in this book to understand the performance of our model:

```

y_predicted = model_svm.predict(X_test)
print(classification_report(y_test,y_predicted))

precision    recall  f1-score   support

 Iris-setosa      1.00      1.00      1.00      19
 Iris-versicolor   0.92      0.80      0.86      15
 Iris-virginica    0.83      0.94      0.88      16

           accuracy                           0.92      50
    macro avg       0.92      0.91      0.91      50
 weighted avg     0.92      0.92      0.92      50

```

13. Creating a confusion matrix. The cumulative number of right and wrong predictions are present in the confusion matrix. They are summarized with count values and broken down by each class.

The first entry is the True Positives (Target variable is positive and predicted correctly), the next entry to the right is False Negatives (Target variable is positive but predicted as negative), the entry below true positives is False Positives (Target variable is negative but predicted as positive) and the entry next to that is True Negatives (Target variable is negative and predicted correctly):

```

print(confusion_matrix(y_test, y_predicted))

[[19  0  0]
 [ 0 12  3]
 [ 0  1 15]]

```

## Case study conclusion

The accuracy of our model is 92%, which is a perfect percentage to achieve. The above case study shows us how we can use the linear kernel of the SVMs to predict the target variable based on the given feature set.

It is essential to understand that **Exploratory Data Analysis** or EDA plays a vital role in the machine learning process as it takes you through the various steps of cleaning and preprocessing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may

need to look at several characteristics like the amount of data (number of rows), the attribute strength, and so on to revamp your model.

## Case study - II

This case study will take us through the various steps that are involved in building an SVMs model. In this section, we will be focusing on datasets that have different types of data distributions.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model. In the following case study, we will build a model to predict the class of an iris flower based on its feature values.

## About the data

The dataset that we are using for this case study has been collected from the UCI machine learning repository. The dataset is known as the **Ionosphere Dataset**.

The features consisting of the radar data were collected by a system in **Goose Bay, Labrador**. This system consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts. See the paper for more details. The targets were free electrons in the ionosphere.

"Good" radar returns are those showing evidence of some type of structure in the ionosphere. "Bad" returns are those that do not; their signals pass through the ionosphere.

The following section gives us some insights into the attributes present in the dataset.

## Attribute information

We do not have information about the attributes except the target variable, which can either indicate a good return from the radar or a lousy return.

Thirty-four other attributes describe the structure/readings that were collected from the radar. We will rename the attributes as attr1-attr34 for the

34 attributes present in the dataset.

## Python code and step-by-step regression analysis

The following steps define code snippets and their outputs for a dataset that is being analyzed and worked upon using machine learning techniques. You can use the code and type it on your Jupyter Notebook/Google Colab or an editor on your local computer.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn import svm
from sklearn import preprocessing
from sklearn import utils
from sklearn.preprocessing import MinMaxScaler
%matplotlib inline
```

2. Read the data (which is stored in the .csv format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `ionosphere.csv`. I have stored the datasets under the /SVM folder under the /Datasets folder on my local system:

```
df = pd.read_csv('Datasets/SVM/ionosphere.data')
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the output image of the `df.head()` command:

```

Out[7]: <bound method NDFrame.head of
   0   1   0   1.00000  -0.18829  0.93035  -0.36156  -0.10868  -0.93597  1.00000
   1   1   0   1.00000  -0.03365  1.00000  0.00485  1.00000  -0.12062  0.88965
   2   1   0   1.00000  -0.45161  1.00000  1.00000  0.71216  -1.00000  0.00000
   3   1   0   1.00000  -0.02401  0.94140  0.06531  0.92106  -0.23255  0.77152
   4   1   0   0.02337  -0.00592  -0.09924  -0.11949  -0.00763  -0.11824  0.14706
   5   1   0   0.97588  -0.16602  0.94601  -0.20800  0.92806  -0.28350  0.85996
   6   0   0   0.00000  0.00000  0.00000  0.00000  1.00000  -1.00000  0.00000
   7   1   0   0.96355  -0.07198  1.00000  -0.14333  1.00000  -0.21313  1.00000
   8   1   0   -0.01864  -0.08459  0.00000  0.00000  0.00000  0.00000  0.11470
   9   1   0   1.00000  0.06655  1.00000  -0.18388  1.00000  -0.27320  1.00000
  10   1   0   1.00000  -0.54210  1.00000  -1.00000  1.00000  -1.00000  1.00000
  11   1   0   1.00000  -0.16316  1.00000  -0.10169  0.99999  -0.15197  1.00000
  12   1   0   1.00000  -0.86701  1.00000  0.22280  0.85492  -0.39896  1.00000
  13   1   0   1.00000  0.07380  1.00000  0.03420  1.00000  -0.05563  1.00000
  14   1   0   0.50932  -0.93996  1.00000  -0.26708  -0.03520  -1.00000  1.00000
  15   1   0   0.99645  0.06468  1.00000  -0.01236  0.97811  0.02498  0.96112
  16   0   0   0.00000  0.00000  -1.00000  -1.00000  1.00000  1.00000  -1.00000
  17   1   0   0.67065  0.02528  0.66626  0.05031  0.57197  0.18761  0.08776

```

*Figure 7.10: Output of df.head()*

In the next step, we will assign names to all the attributes.

- As we observed in *Step 3*, the attributes do not have any names/titles to them, therefore, we will be assigning attribute names to all the columns along with the target variable which is titled ‘label’:

```

df.columns = ['attr1', 'attr2', 'attr3', 'attr4', 'attr5',
              'attr6', 'attr7', 'attr8', 'attr8', 'attr9', 'attr10',
              'attr11', 'attr12', 'attr13', 'attr14', 'attr15', 'attr16',
              'attr17', 'attr18', 'attr19', 'attr20', 'attr21', 'attr22',
              'attr23', 'attr24', 'attr25', 'attr26', 'attr27', 'attr28',
              'attr29', 'attr30', 'attr31', 'attr32', 'attr34', 'label']

```

- We now use `df.describe()` to get the statistical insights into our dataset. The describe functions allows us to look at various values like mean, count, standard deviation, maximum of the values present in the iris dataset:

```
df.describe()
```

Here is the output image of the `df.describe()` command:

	attr1	attr2	attr3	attr4	attr5	attr6	attr7	attr8	attr9	...	attr24	attr25	
count	350.000000	350.0	350.000000	350.000000	350.000000	350.000000	350.000000	350.000000	350.000000	...	350.000000	350.000000	
mean	0.891429	0.0	0.640330	0.044667	0.600350	0.116154	0.549284	0.120779	0.510453	0.181756	...	0.395643	-0.069928
std	0.311546	0.0	0.498059	0.442032	0.520431	0.461443	0.493124	0.520816	0.507117	0.484482	...	0.579206	0.508675
min	0.000000	0.0	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	...	-1.000000	-1.000000
25%	1.000000	0.0	0.471518	-0.065388	0.412555	-0.024868	0.209105	-0.053483	0.086785	-0.049003	...	0.000000	-0.323745
50%	1.000000	0.0	0.870795	0.016700	0.808620	0.021170	0.728000	0.015085	0.682430	0.017550	...	0.549175	-0.014915
75%	1.000000	0.0	1.000000	0.194727	1.000000	0.335318	0.970445	0.451572	0.950555	0.536192	...	0.907165	0.157922
max	1.000000	0.0	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	...	1.000000	1.000000

*Figure 7.11: Output of df.describe()*

6. We can look at the data types of the columns to understand what kind of data types we have in place. One can observe that except for the target variable, all the other columns are numeric. We have a combination of integer and float variables in our columns:

```
df.dtypes
attr1      int64
attr2      int64
attr3      float64
attr4      float64
attr5      float64
attr6      float64
attr7      float64
attr8      float64
attr8      float64
attr9      float64
attr10     float64
attr11     float64
attr12     float64
attr13     float64
attr14     float64
attr15     float64
attr16     float64
attr17     float64
attr18     float64
attr19     float64
attr20     float64
attr21     float64
attr22     float64
attr23     float64
attr24     float64
attr25     float64
attr26     float64
attr27     float64
attr28     float64
attr29     float64
attr30     float64
attr31     float64
```

```
attr32    float64  
attr34    float64  
label     object  
dtype: object
```

7. To visualize the data better, we will use specific plots to look at the relationship between the various attributes in our dataset.

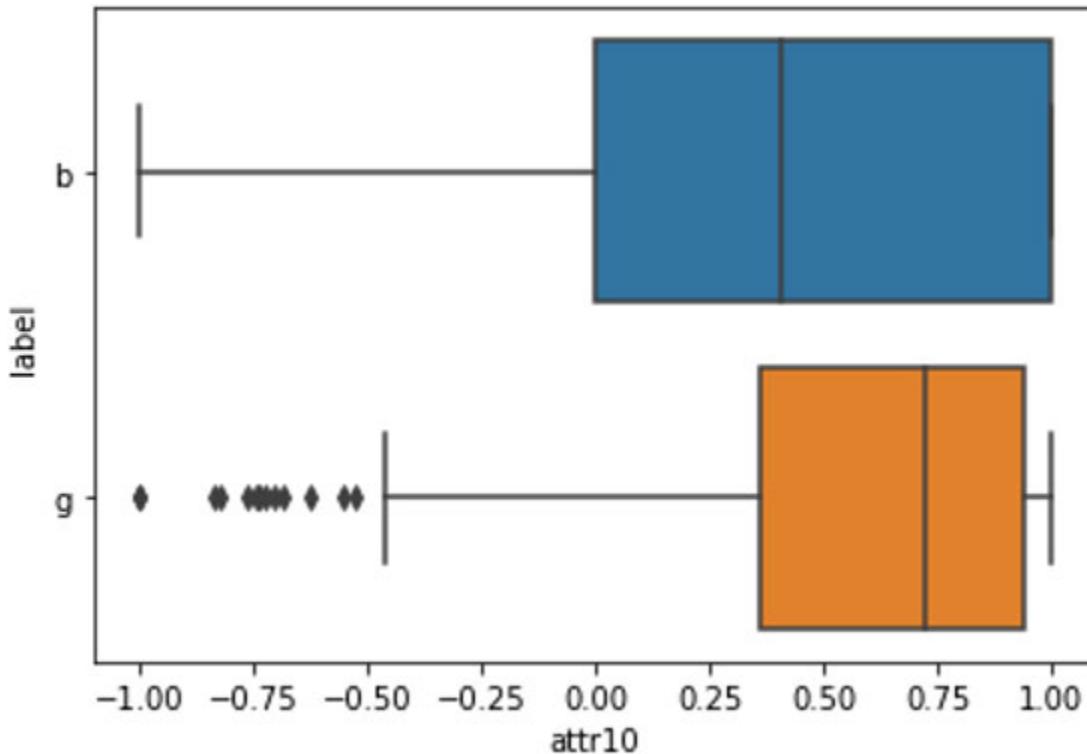
We will draw boxplots to look at the distribution of various columns with the target variable.

The boxplots give us an understanding of how the minimum, maximum, and median values are distributed for the various classes.

The following code snippet will create boxplots between `df.attr10` and `label` variables:

```
sns.boxplot(df.attr10,df.label)
```

Here is the visual image representing the boxplot between `df.attr10` and `label` variables:

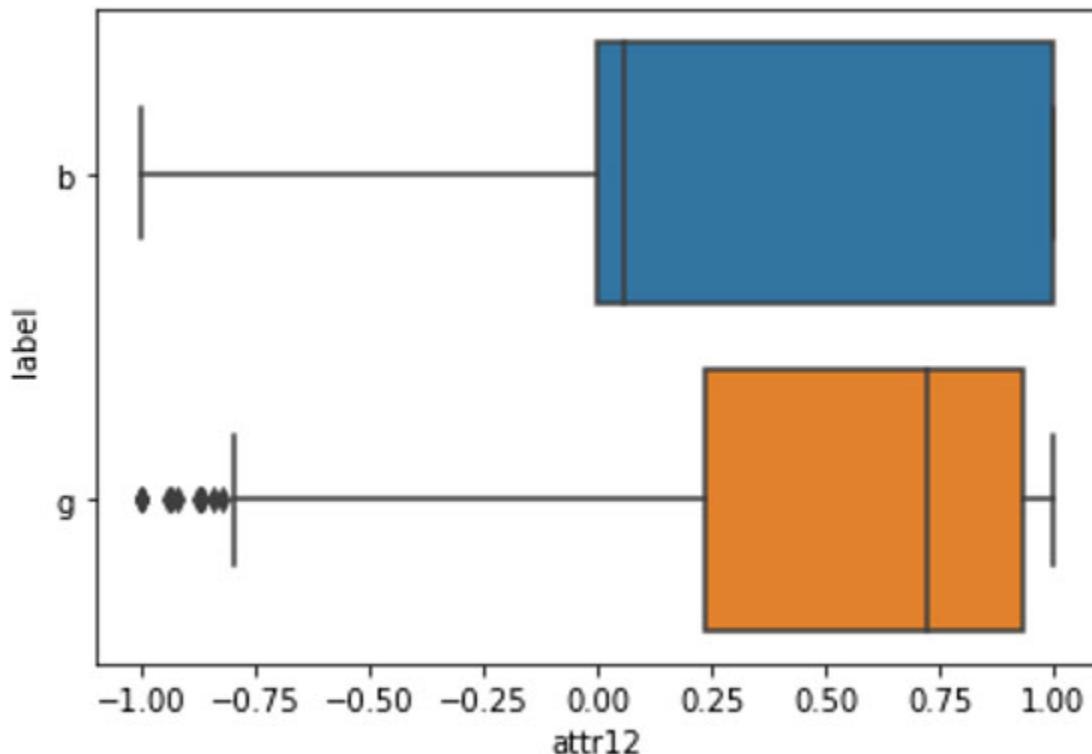


**Figure 7.12:** Boxplot between `attr10` and the `label`

The following code snippet will create boxplots between `df.attr12` and `label` variables:

```
sns.boxplot(df.attr12,df.label)
```

Here is the visual image representing the boxplot between `df.attr12` and `label` variables:

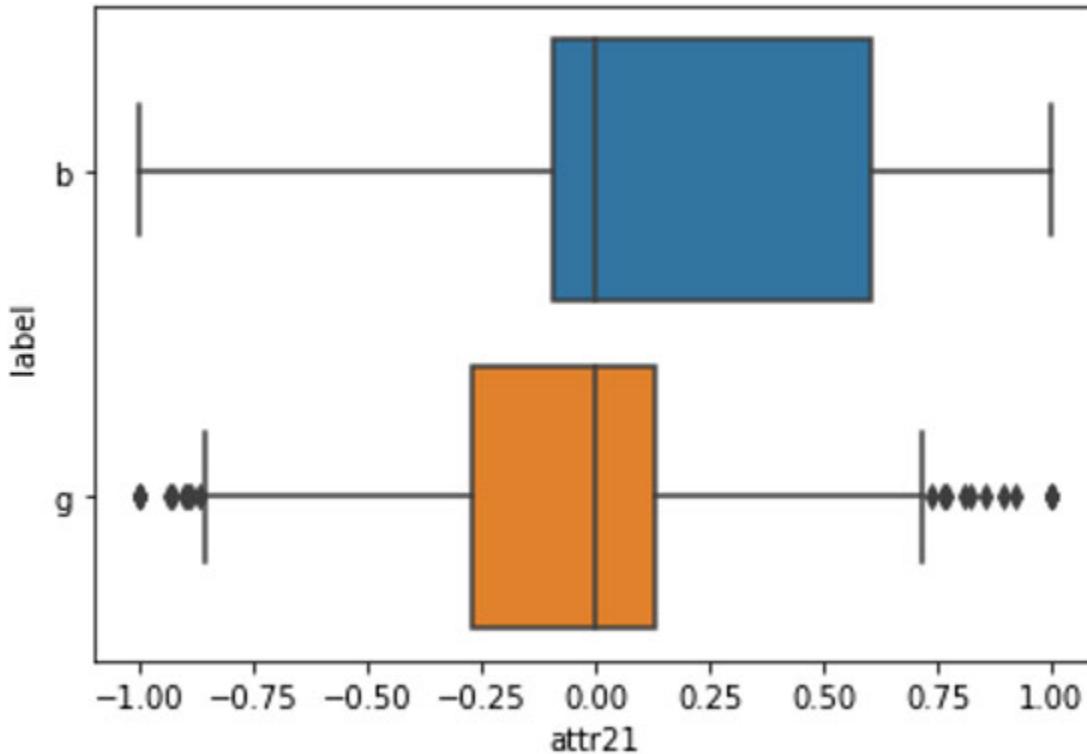


*Figure 7.13: Boxplot between attr12 and the label*

The following code snippet will create boxplots between `df.attr21` and `label` variables:

```
sns.boxplot(df.attr21,df.label)
```

Here is the visual image representing the boxplot between `df.attr21` and `label` variables:



**Figure 7.14:** Boxplot between `attr21` and the label

In the next step, we will look at some more visualization in the form of pair plots.

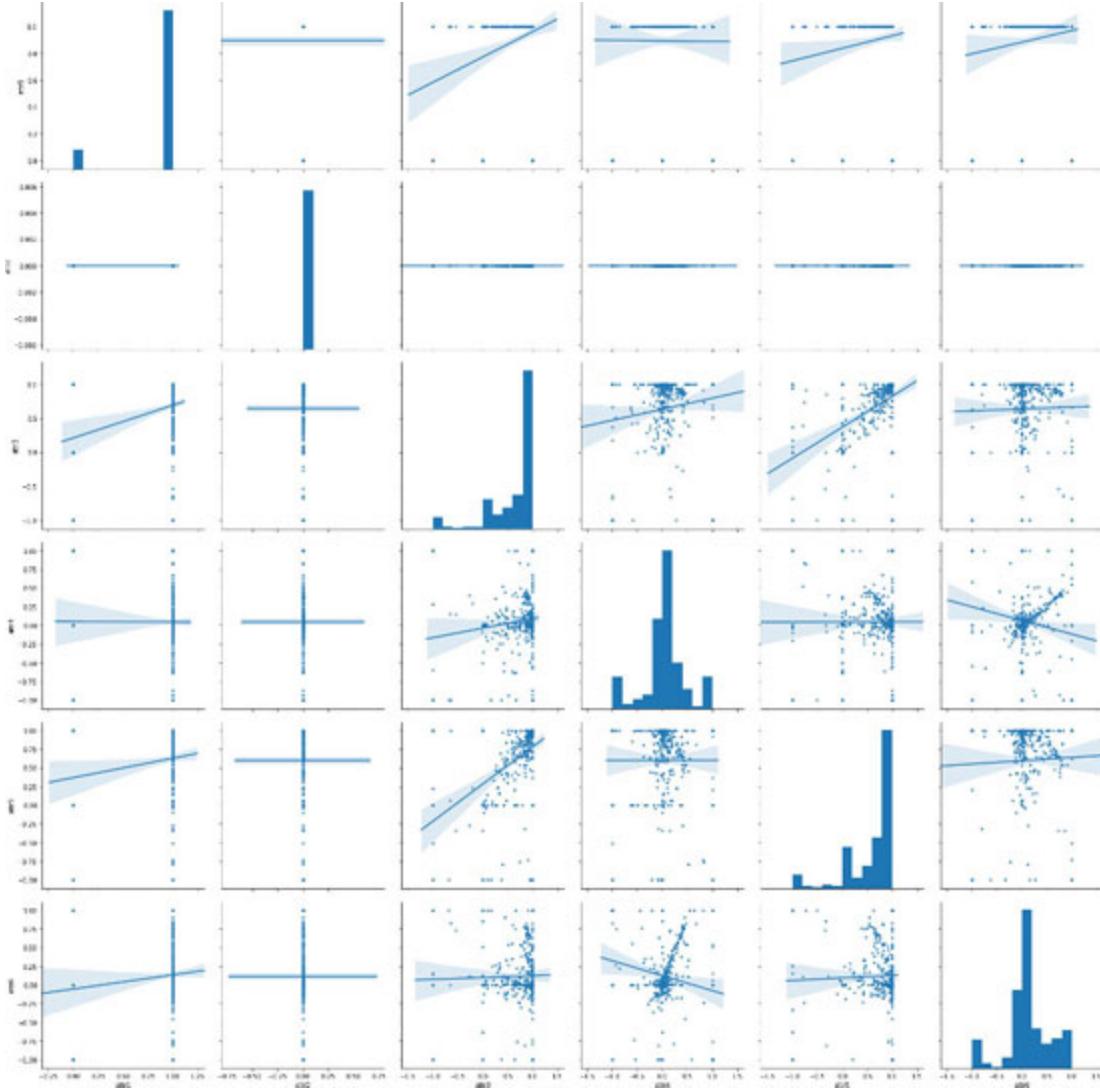
8. We will also create pair plots of some selected columns (you can choose the ones of your choice) using the seaborn (`sns`) library. Pair plots are useful when you want to visualize the relationship between two variables.

You can create pairplots in such a way that you can visualize the relationship between each pair of attributes. We have created pair plots by choosing the first six attributes along with the label, just to have a glimpse at how the data is related to each other in pairs.

The following code snippet will create pairplots between the different mentioned attributes. Here, we are choosing the first six attributes:

```
df_n=df[['attr1','attr2','attr3','attr4','attr5','attr6','label']]
sns.pairplot(df_n, height=4, kind="reg", markers=".")
```

Here is the visual image representing the pairplot between the first six attributes:



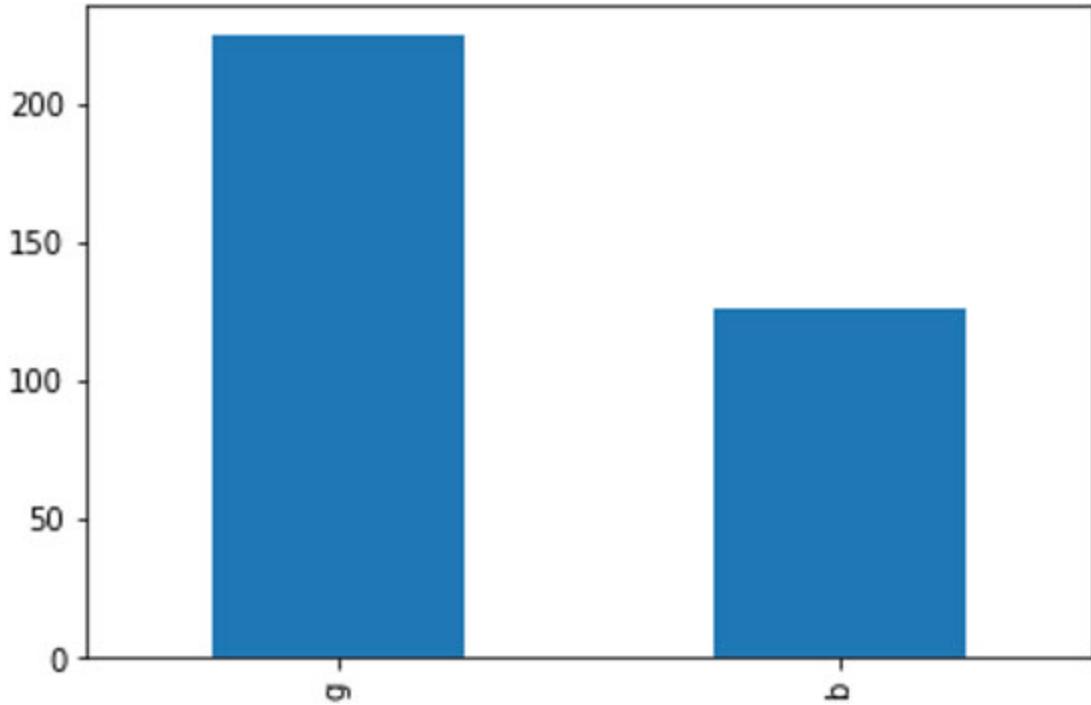
*Figure 7.15: Pairplots between the first few attributes and label*

9. We also need to look at the target variable distribution in the form of a bar graph. We can observe in the bar plot that there are more of ‘good’ readings as compared to the ‘bad’ ones. This bar plot gives us clarity and understanding of how the target variable is skewed.

The following code snippet will create a barplot to show the distribution of the label variable in our dataset:

```
df['label'].value_counts().plot.bar()
```

Here is the visual image representing the barplot of the label variable:



**Figure 7.16:** Bar plot showing the label distribution

In the next step, we will separate the label and the feature variables.

10. We will now separate the target variable and the attributes for creating the machine learning model. We can drop the target variable column from our current DataFrame:

```
y = df['label']
x = df.drop(columns = ['label'])
```

11. We need to encode the target variable using the `LabelEncoder` since it's a categorical variable:

```
lab_enc = preprocessing.LabelEncoder()
training_scores_encoded = lab_enc.fit_transform(y)
```

12. We will split the data into training and test data and build an SVM model using the linear kernel for our data:

```
x_train, x_test, y_train, y_test = train_test_split(x,
                                                    training_scores_encoded, test_size=0.33, random_state=42)
```

13. Since we have a mix of integer and floating variables, it is always a good idea to scale the data in order to normalise the values. We will

use the `MinMaxScaler` here to scale the testing and the training DataFrames:

```
scaling = MinMaxScaler(feature_range=(-1,1)).fit(X_train)
X_train = scaling.transform(X_train)
X_test = scaling.transform(X_test)
```

14. We now create an SVM model with the '`rbf`' kernel:

```
model_svm = svm.SVC(kernel='rbf')
print(model_svm)
```

15. We will fit our training data in the SVM model that we defined above with the `rbf` kernel.

You can see that the value of C here is 1.0:

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

### Output:

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
     decision_function_shape='ovr', degree=3,
     gamma='auto_deprecated',
     kernel='rbf', max_iter=-1, probability=False,
     random_state=None,
     shrinking=True, tol=0.001, verbose=False)
```

16. Here, we will predict the target labels for our test dataset, and you can see the accuracy is about 91%, which is pretty good. You can also alternatively experiment with the other kernels and check the accuracy for those:

```

y_predicted = model_svm.predict(X_test)
print(classification_report(y_test,y_predicted))

Output:

precision    recall   f1-score   support
          0       0.97      0.78      0.86      40
          1       0.89      0.99      0.94      76

   accuracy                           0.91      116
  macro avg       0.93      0.88      0.90      116
weighted avg       0.92      0.91      0.91      116

```

17. Creating a confusion matrix. The cumulative number of right and wrong predictions are present in the confusion matrix. They are summarized with count values and broken down by each class.

The first entry is the True Positives (Target variable is positive and predicted correctly), the next entry to the right is False Negatives (Target variable is positive but predicted as negative), the entry below true positives is False Positives (Target variable is negative but predicted as positive) and the entry next to that is True Negatives (Target variable is negative and predicted correctly):

```
print(confusion_matrix(y_test, y_predicted))
```

**Output:**

```
[[31  9]
 [ 1 75]]
```

## Case study conclusion

The accuracy of our model is 91%, which is a high percentage to achieve. You can also observe from the confusion matrix that the number of False Negatives (9) and the number of False Positives (1) are extremely less; therefore, this is a good model for our dataset.

The above case study shows us how we can use the various kernels in SVMs to predict the target variable when there are a lot of attributes. It is essential to understand that Exploratory Data Analysis or EDA plays a vital role in Regression Analysis as it takes you through the various steps of

cleaning and preprocessing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data, the attribute strength, and so on, to revamp your model.

## Conclusion

For data with multiple dimensions, support vector machines are high algorithms to work with. Different types of kernels help us define the different types of hyperplanes, depending on the way the classes are separated. SVMs give us the ability to define hyperplanes for various distributions like linear, polynomial, and radial distributions.

There are several pros of using SVMs, some of which are:

- If the number of attributes is much higher
- Best fit if the classes are separable
- Depends on support vectors; therefore, outliers are not affected too much

In this chapter, you have learned the working of SVM, its kernels, and how to solve an SVM problem using Python.

Our next chapter will introduce you to decision trees, which is another widely used Supervised Learning methodology for a large number of features. A decision tree is an exciting concept and highly relatable as we use them while taking several decisions in our day-to-day life. You will learn more about decision trees and their context in the machine learning world in the upcoming chapter.

## Quiz

1. What is the generalization error in the context of SVM?
2. What is the role of the C-parameter in the SVM algorithms?
3. What are the different parameters that shape the effectiveness of a Support Vector Machine algorithm?

4. Which are different kernels in SVM, and what's their role?

# CHAPTER 8

## Decision Trees

### Introduction

This chapter will introduce us to a supervised learning algorithm known as a decision tree. Decision trees are a visual way of looking at the dataset given to us and making decisions based on the different scenarios/patterns that we encounter in the data.

It is an algorithm that works on a conditional statement, creating a tree-like structure to visualize our model and be able to explain the decisions that the model has taken. In this chapter, we will take a look at decision tree construction, some terminologies associated with decision trees, and a step-by-step case study in Python.

### Structure

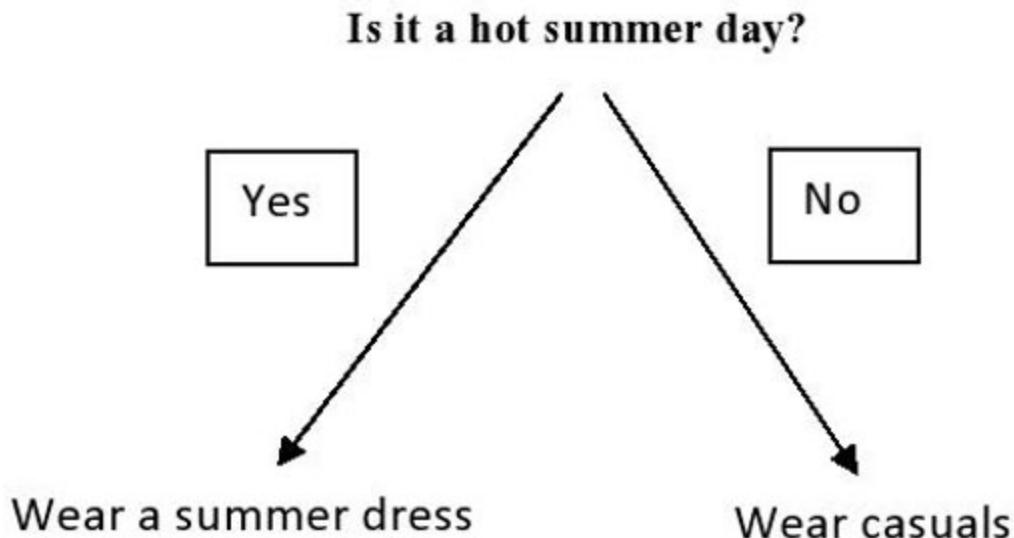
- What are decision trees?
- CART algorithm
- Decision tree construction
- Truncation and pruning
- Calculating information gain
- Case Study - Decision trees
- Quiz

### Objectives

- Understand and explore decision trees
- Learn the various ways of creating an efficient decision tree
- Learn to apply the decision tree algorithm on a dataset using Python

## What are decision trees?

Decision trees are a common phenomenon in our day to day lives where we try to make a decision based on the various options presented to us. From deciding what to wear on a hot summer afternoon to decide on the label for a machine learning observation, decision trees are highly useful tools for several applications:



*Figure 8.1: Decision tree example*

For example, a simple decision tree is shown in [Figure 8.1](#), where we see that based on the weather, one decides on how they want to get dressed for the day. Though the above is a single **Yes/No** decision tree, there are several conditions that one could add to the child nodes to make it a more extended tree.

In machine learning, decision trees can be used for both classification and regression. A decision tree splits the data into several sets, and as you keep asking more and more questions, these sets are split into more subsets to arrive at a final decision. Decision trees represent all possible paths to reach a solution or a decision. As the final decision is made with the help of several questions and answers, the decision is easy to trace back to.

One of the most significant advantages that decision trees offer is the ability to explain the decisions or the classification that was made with concrete proofs and examples.

## CART algorithm (classification and regression trees)

The CART algorithm is one of the most efficient and well-known techniques in machine learning.

As we have already looked at the decision tree construction above, here is a brief explanation of what a decision tree represents when it comes to classification trees:

- The decision tree is top to bottom or left to right flowcharts like structure, which represents a series of questions and their outcomes.
- The root node denotes the beginning of the decision tree or the first attribute where we want to start splitting the data.
- The non-leaf nodes are decision nodes that lead to specific outcomes.
- The leaf nodes or terminal nodes are outcomes or labels that are used for classification.

## Advantages of CART

- CART is a visual appeal to the users and can help explain the decisions simply and straightforwardly.
- It can be used for both numerical and categorical data.
- We can perform feature selection or attribute screening using CART.
- It does not require users to put in much effort when it comes to preparing the data to feed the machine learning models.

Some of the disadvantages of decision trees can be overfitting the dataset and instability in the model due to small changes in the trees. Both of these can be tackled using efficient methods like truncation and pruning as well as advanced algorithms like random forests, which use more than one decision tree to model a problem.

## Decision tree construction

As we already know, a decision tree is a graphical representation of the process of making a final decision by repeatedly asking multiple questions

as we go along exploring our dataset.

If you are familiar with trees in the programming context, decision trees follow a similar pattern. They start with a root, and as more and more questions are added, it keeps branching out, leaving child nodes or leaf nodes at the very bottom.

Let us take a use case and understand how we can construct a decision tree. Here is an example dataset that we will be using to explain the construction of decision trees. In the below dataset, we have three columns - **Fruit Colour** (colour of the fruit), **Diameter** (diameter/length of the fruit), and the **Name** (name of the fruit).

Fruit Colour and Diameter are our independent variables, and the Name is the dependent variable in our dataset. We will construct a decision tree that will help us arrive at the right kind of label/name for the given characteristics of the fruit. The challenge here is to identify a unique label for the given characteristics.

The following table is an example, dataset that will help us construct a decision tree:

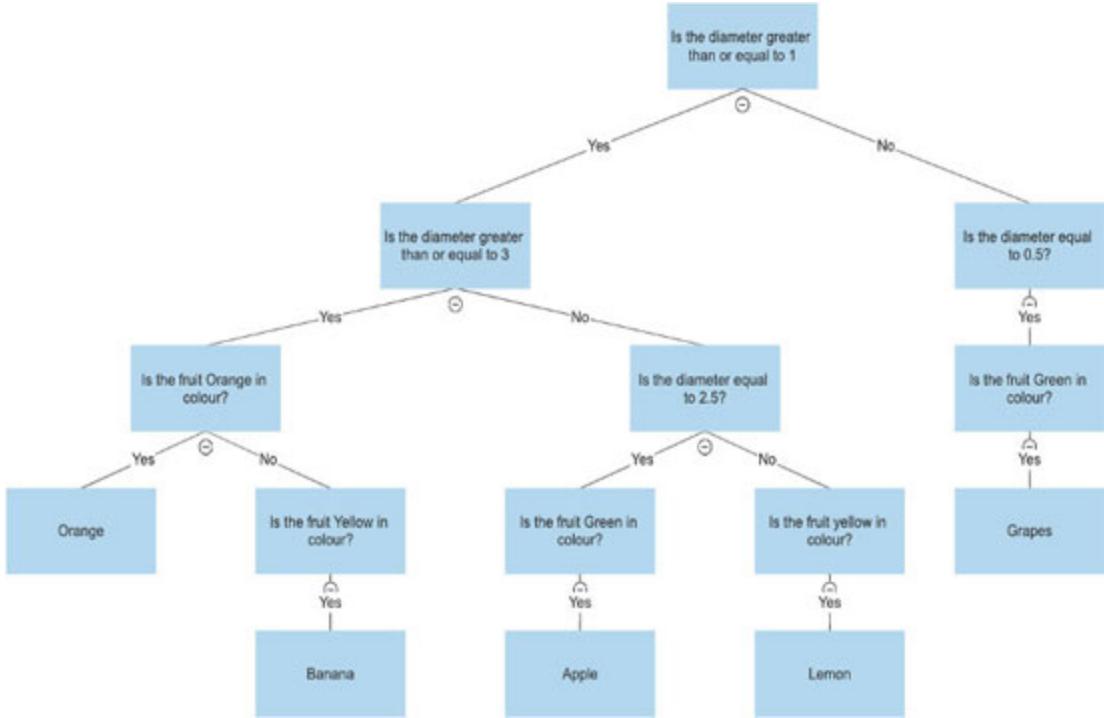
Fruit Colour	Diameter (in inches)	Name
Orange	3	Orange
Yellow	1	Lemon
Yellow	5	Banana
Green	0.5	Grapes
Green	2.5	Apple

*Table 8.1: Example dataset for decision tree construction*

To start the tree construction, we will take the first question, let's start with the **Diameter** in consideration. We will only take fruits that have 1 inch or bigger diameters, and the one less than that will be labeled differently.

Our first question is - *Is the diameter greater than or equal to 1?*

The first question is represented as the root of the tree, and the following questions will be branches for this tree. [\*Figure 8.2\*](#) is the tree representation after several questions that were asked in the process to label the fruits given to us:



*Figure 8.2: Decision tree construction for Table 8.1*

As you can see in [Figure 8.2](#), at every level of the tree, we ask different questions best to classify the target variable for our given dataset. In some cases, you could also arrive at the labels much earlier than depicted in the decision tree by the process of elimination. It's not as frequently recommended because to create a model that considers all possible questions; it's always better to have an exhaustive set of questions and answers before you make a final decision, that is, assign a label in this case.

In the next section, we will come across techniques that help reduce the complexity of a decision tree. Truncation and pruning are techniques that are used to reduce the size of the decision trees by removing branches that do not necessarily add value to the classification.

## Truncation and pruning

Truncation and pruning helps reduce the decision tree size, thereby avoiding overfitting the dataset. Decision tree algorithms are highly prone to overfitting as they analyze the dataset in-depth and come to decisions after several iterations of questions. Effective pruning can prevent it from happening.

As we go down the decision tree, we see that the decision tree keeps splitting until we reach one single leaf node. Pruning signifies cutting down the tree. There are different pruning strategies that we can apply to avoid overfitting:

- **Minimum error:**

We want to minimize the cross-validation error, that is, we need to understand at what point in time do we need to prune the tree to get the minimum cross-validation error. We calculate the error at every step of the tree and keep track of the cross-validation error values. We prune the tree at the point where we find the minimum cross-validation error value. This way, we can make sure that any new data point that comes in has the smallest error.

As we have already seen in the previous chapters, cross-validation can help us identify the strength of our model for new incoming test data.

- **Smallest tree:** The smaller tree is a tree that is pruned a little before we reach the minimum cross-validation error. The smaller tree has a lambda value, which minimizes the cross-validation error, plus there is a buffer of 1 standard error provided.

Typically, one standard error is chosen as its values are not that far from the range of values that encompasses the lambda variable.

Early stopping or pre-pruning is a great way to avoid overfitting the dataset. In this process, we will restrict ourselves from building the tree earlier on in the process. To perform early stopping successfully, we will be noting down the cross-validation error at every step in our tree building process. If the value of the error does not decrease significantly over the iterations, we can stop building the tree and achieve pre-pruning.

Early stopping saves us a lot of time as compared to post pruning but involves the overhead of calculating cross-validation error at every step.

For best accuracy, minimum error pruning is the right choice. It may or may not be with early stopping implementation.

## Calculating information gain

The variable, like information gain, will help us understand the mathematics behind decision tree workings.

In a classification problem, it is essential to understand how the sample set is distributed before modeling the decision tree. Before diving into the concept of information gain, we will understand two essential concepts, namely entropy, and Gini index, which will help us understand the distribution of classes in the sample set.

## What is Entropy?

To describe a sample (in our case, it can be a sample dataset), we can use the entropy factor. It is the amount of information that is needed to describe the sample. If all the sample elements are similar to each other, then the entropy value is minimum, that is, 0 whereas if the different types of elements in the sample are equally spread, then the entropy value is maximum, that is, 1.

Mathematically, entropy can be written as:

$$\text{Entropy} = -i = I \sum_{i=1}^{n} p_i * \log(p_i)$$

Entropy and Gini index are useful measures while trying to understand our sample set. In the next section, we will look at what is Gini index and how we can use these properties to calculate the information gain.

## What is the Gini index?

To calculate the inequality in the sample, we use a parameter called the Gini index. It takes up values between 0 and 1. If the value of the Gini index is 0, it would mean that the data is homogeneously spread in the sample set.

If the Gini index holds a value of 1, it would mean that there is maximum inequality in the sample set.

Gini index is calculated as the sum of the square of probabilities of all the classes present in the dataset:

$$\text{Gini Index} = 1 - \sum_{i=1}^n p_i^2$$

Here,  $p$  is the probability of each class, and  $i$  is the loop over  $n$  classes that exist.

## What is the information gain?

Once the dataset is split based on one of the attributes, the information gain is calculated based on the decrease in the entropy. To construct a decision tree, we need to find out the attribute that gives us the highest information gain.

Here are the steps to calculate the information gain based on entropy:

1. Calculate the entropy of the target using the entropy formula given above.
2. Split the given dataset into different attributes. Calculate the entropy for each of the branches and then add them to get the total entropy for that particular split. Subtract this entropy from the entropy calculated in *Step 1*. The resulting answer is the information gain.
3. After performing *Step 2* for various attributes, choose the attribute as the decision node, which gives the maximum information gain, and repeats the process for every branch.
4. Branch nodes with entropy 0 are leaf nodes as there is no more entropy decrease further.
5. Contrary to the previous point, branches with entropy higher than 0 need to be split further
6. The same steps are repeated for every branch that shows an entropy greater than 0.

Information gain helps us construct an efficient decision tree based on the values of the entropy at each level. In the next section, we will look at a Case study that will take us through an end to end decision tree algorithm applied on a dataset.

## Case Study - I

This case study will take us through the various steps that are involved in building a decision trees model. In this section, we will be focusing on datasets that can be modeled using a decision tree model.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model. In the following case study, we

will build a model to predict the class of an iris flower based on its feature values.

## About the data

The dataset that we are using for this case study is one of the most common datasets used in machine learning. The dataset is called the titanic dataset and describes the various features of the passengers who were traveling on the titanic. Based on these different features, we will predict whether a patient survived the titanic disaster or not. The data in Kaggle contains both training and test files separately, but in our case, we will use one set of data, which is the training set to predict the target variable using decision trees.

The attributes in our dataset describe the various features of a passenger. Here is more information about what the attributes are:

### **Attribute Information:**

Variable Name	Variable Meaning	Values(if any)
PassengerId	ID is given to each passenger	
Survived (target variable)	Survival chances	0 = No, 1 = Yes
Pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
Name	Name of the Passenger	
Sex	Sex	
Age	Age in years	
SibSp	# of siblings/spouses aboard the Titanic	
Parch	# of parents/children aboard the Titanic	
Ticket	Ticket number	
Fare	Passenger fare	
Cabin	Cabin number	
Embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

*Table 8.2: Attribute information for our decision tree dataset*

In the next section, we will be looking at the Python code and go through each step of the regression analysis to find out how the decision tree algorithm applies to our dataset.

## **Python code and step-by-step regression analysis**

It is one of the most crucial and essential sections of our case study - being able to analyze the data using Python libraries and deriving insights from the data.

The following steps have a Python code snippet and its corresponding output in most of the cases. You can type the following code in your Jupyter Notebook/A google colab notebook or on your local editor and execute them to see the desired results.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
# Load libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier # Import
Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import
train_test_split function
from sklearn import metrics #Import scikit-learn metrics
module for accuracy calculation
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn import preprocessing
```

2. Read the data (which is stored in the .csv format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `titanic_train.csv`. I have stored the datasets under the `/DecisionTrees` folder under the `/Datasets` folder on my local system:

```
df =
pd.read_csv("/Datasets/DecisionTrees/titanic_train.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the `df.head()` output obtained on our dataset:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th... Heikkinen, Miss. Laina	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	26.0	0	0	STON/O2.3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

*Figure 8.3: Output of df.head()*

In the next step, we will look at the column names of our dataset.

4. Now we will use the `df.columns` to look at all the column names (sometimes, all the columns are not displayed appropriately on Jupyter due to space constraints):

```
df.columns
```

### Output:

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex',
       'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

5. We now use `df.describe()` to get the statistical insights into our dataset. The describe functions allows us to look at various values like mean, count, standard deviation, maximum of the values present in the Titanic dataset:

```
df.describe()
```

Here is an image representing the output of `df.describe()` for our dataset:

	<b>PassengerId</b>	<b>Survived</b>	<b>Pclass</b>	<b>Age</b>	<b>SibSp</b>	<b>Parch</b>	<b>Fare</b>
<b>count</b>	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
<b>mean</b>	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
<b>std</b>	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
<b>min</b>	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
<b>25%</b>	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
<b>50%</b>	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
<b>75%</b>	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
<b>max</b>	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

*Figure 8.4: Output of df.describe()*

In the next step, we will use the `dtype` function to look at the different types of attributes present in our dataset.

6. We will now use the `dtype` function to understand the types of our different attributes:

```
df.dtypes
```

### Output:

PassengerId	int64
Survived	int64
Pclass	int64
Name	object
Sex	object
Age	float64
SibSp	int64
Parch	int64
Ticket	object
Fare	float64
Cabin	object
Embarked	object
dtype:	object

7. For all categorical variables, it is necessary to understand the spread of the categorical variables in the dataset. The categorical variables play a massive role in the final prediction, and value counts will help us understand if the variable is significant or not.

You can see that the below section has several outputs for the `value_counts` of the categorical variables.

In the below outputs, we can see that the Name variable is not significant because almost all the names are unique and do not contribute to the dataset overall as a game-changing feature and therefore, we can drop that variable from our dataset before feeding the training dataset to the model:

```
df['Name'].value_counts()
```

**Output:**

Abbott, Mrs. Stanton (Rosa Hunt)	1
Moutal, Mr. Rahamin Haim	1
Greenberg, Mr. Samuel	1
Cann, Mr. Ernest Charles	1
Asplund, Master. Clarence Gustaf Hugo	1
Ringhini, Mr. Sante	1
Kink-Heilmann, Miss. Luise Gretchen	1
Meyer, Mr. Edgar Joseph	1
Meanwell, Miss. (Marion Ogden)	1
Ahlin, Mrs. Johan (Johanna Persdotter Larsson)	1
...	

The next output of value counts is for the variable `Sex` present in our dataset:

```
df['Sex'].value_counts()
```

**Output:**

```
male 577  
female 314  
Name: Sex, dtype: int64
```

The next output of value counts is for the variable `Ticket` present in our dataset:

```
df['Ticket'].value_counts()
```

**Output:**

1601	7
CA. 2343	7
3101295	6
CA 2144	6
347088	6
382652	5
S.O.C. 14879	5
PC 17757	4
LINE	4
113781	4
17421	4
349909	4
W./C. 6608	4...

The next output of value counts is for the variable `Cabin` present in our dataset:

```
df['Cabin'].value_counts()
```

### **Output:**

C23 C25 C27	4
B96 B98	4
G6	4
E101	3
F2	3
D	3
F33	3
C22 C26	3
B35	2
C125	2
...	

The next output of value counts is for the variable `Embarked` present in our dataset:

```
df['Embarked'].value_counts()
```

### **Output:**

S	644
---	-----

```
C      168  
Q      77  
  
Name: Embarked, dtype: int64
```

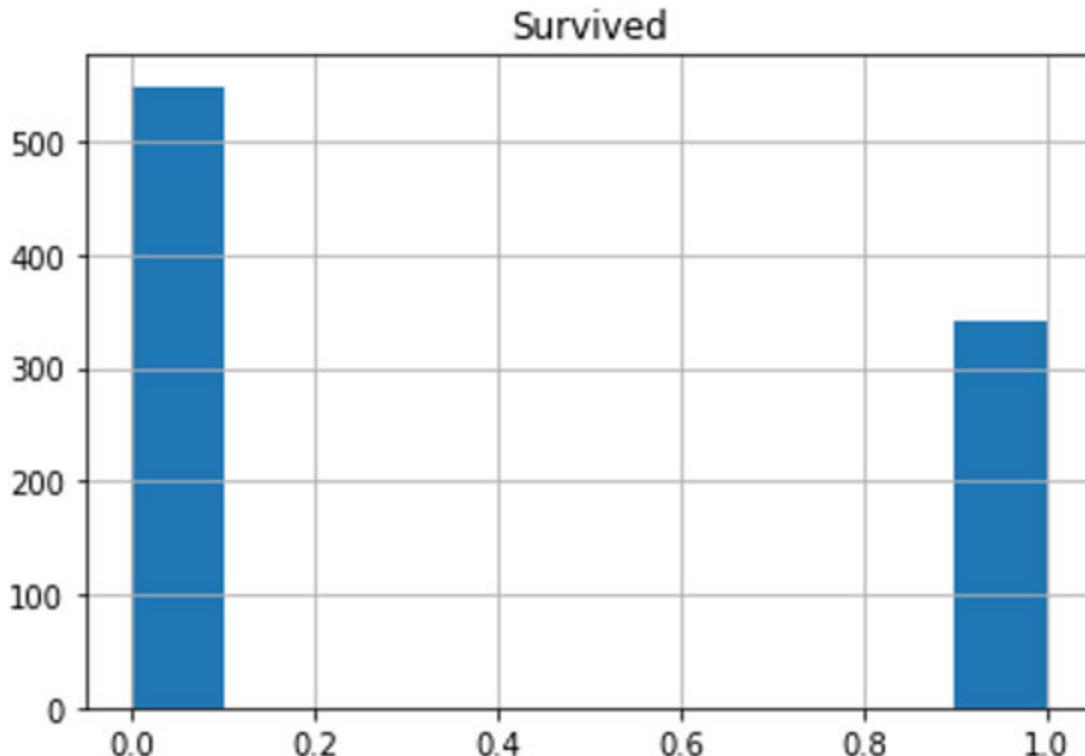
The above series of outputs for value counts gives us an idea of how the categorical variables are distributed in our dataset. The next step will show us a visual plot of the target variable's distribution.

8. Our next step is to plot a histogram that shows the distribution of the target variable in the dataset. This histogram will help us understand if our prediction engine is likely to predict any bias in the classification results.

Our histogram shows that the 'No/0.0' class is more prevalent in our current database, and hence, our model may have slightly more bias towards No. It may not be necessarily the case, but it's good to get this insight early in the Data Processing stage:

```
df.hist('Survived')
```

Here is a distribution graph of our target variable:



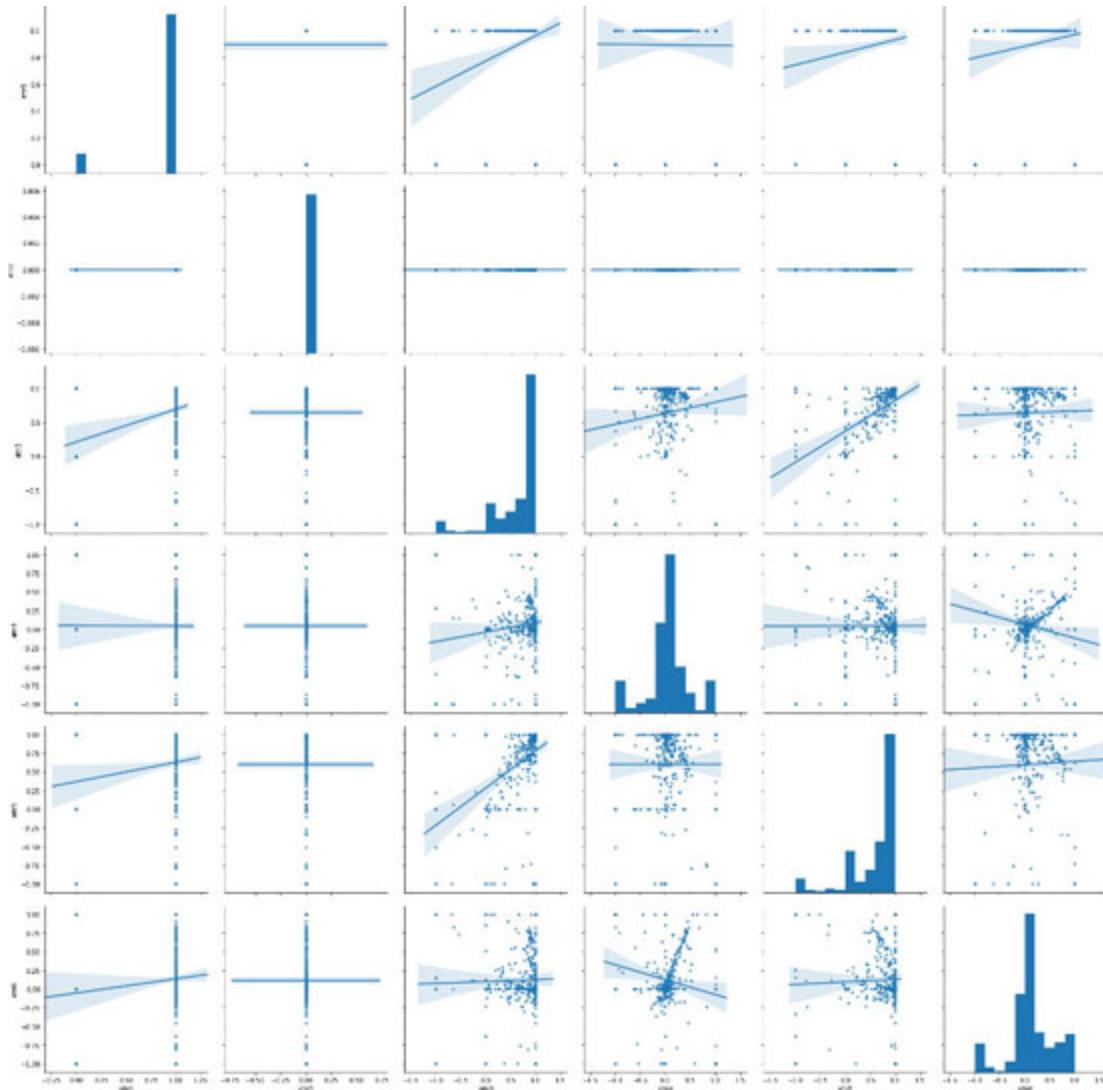
**Figure 8.5:** Histogram describing the distribution of the target variable

In our next step, we will plot pair plots to have a broader angle look at our dataset.

9. Our next step is to plot pair plots, which will show several graphs with the features paired together. Pair plots will give us a good insight into how the features in our dataset vary from each other.

Our pair plots help us understand the correlations and behaviour between the various attributes in the dataset:

```
df_n=df[['PassengerId', 'Survived', 'Pclass', 'Name',  
'Sex', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin',  
'Embarked']]  
sns.pairplot(df_n, height=4, kind="reg", markers=".")
```



**Figure 8.6:** Pair plots of the distribution of the attributes concerning each other in our dataset

In the next step, we will identify the columns that have missing or NA values and fill them with zero or other appropriate values.

10. We will now drop the columns that are not essential attributes for our machine learning process and fill the NA values. It is important to give the parameter `inplace` if you are not creating a new DataFrame after the modifications on the current one:

```
df = df.drop(columns = ['Cabin', 'Name', 'PassengerId',
'Ticket', 'SibSp'])
df.fillna(method='ffill', inplace=True)
df.fillna(method='ffill', inplace=True)
```

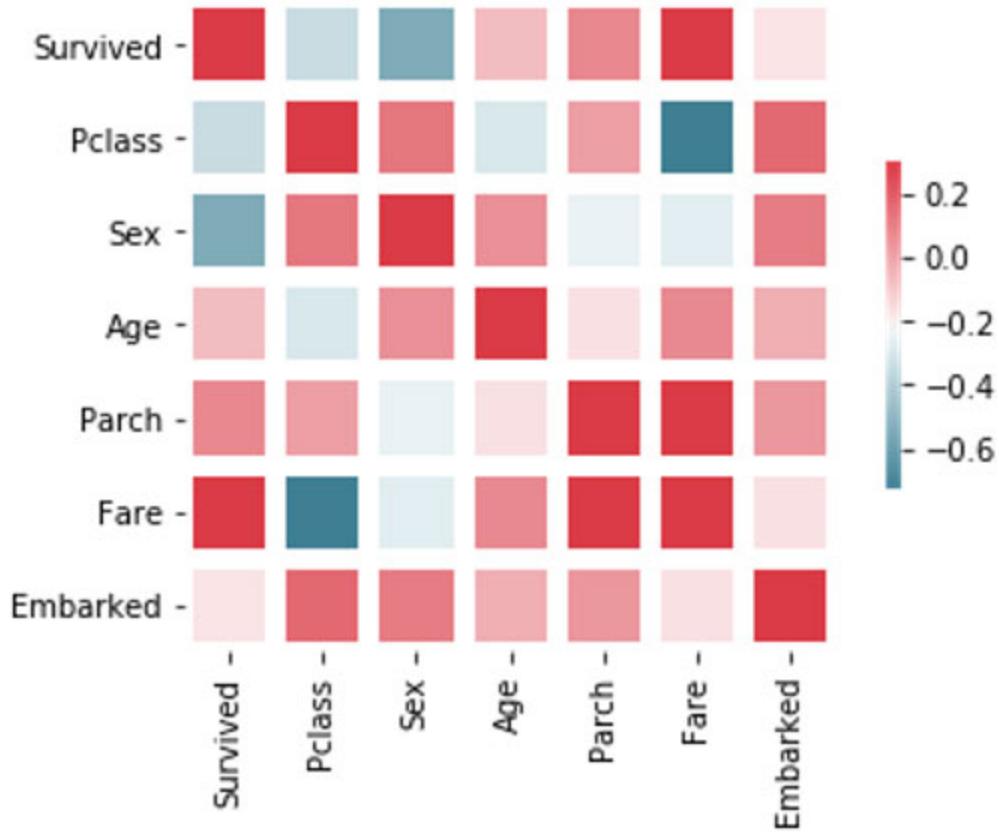
11. In this step, we will perform preprocessing to convert the categorical variables to labels using our `LabelEncoder`:

```
lab_enc = preprocessing.LabelEncoder()
df['Sex'] = lab_enc.fit_transform(df['Sex'])
df['Fare'] = lab_enc.fit_transform(df['Fare'])
df['Embarked'] = lab_enc.fit_transform(df['Embarked'])
```

12. In this step, we will create a correlation matrix to look at the correlation between the various attributes in our dataset:

```
corr = df.corr()
# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.heatmap(corr, cmap=cmap,
vmax=.3, square=True, linewidths=6, cbar_kws={"shrink": .5})
colormap = plt.cm.viridis
```

The following is the output of a correlation heatmap:



**Figure 8.7: Correlation heatmap**

In the next step, we will create a training and test dataset out of our existing dataset.

13. We will now separate our independent variable and dependent variables into separate columns to feed them to our training model. Our target variable to predict here is the `Survived`. We will drop the `Survived` column from the original features list:

```
Y = df['Survived']
X = df.drop(columns = ['Survived'])
```

14. We will split the data into training and test data:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.33, random_state=42)
```

15. We build a decision tree model using the `DecisionTreeClassifier` present in the `sklearn.tree` library. Using the model, we fit our existing training data and try to predict the results for the test data:

```

# Create Decision Tree classifier object
clf = DecisionTreeClassifier()

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

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

```

16. We use the above predictions of the possible Y variable, also known as the Survived variable, in our case, to understand the accuracy of our model.

In this case, we have achieved an accuracy of 75%, which is not extremely efficient but a good accuracy for a huge dataset:

```

# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))

```

### **Output:**

Accuracy: 0.752542372881356

## **Summary**

The accuracy of our model is 75%, which is a high percentage to achieve.

The above case study shows us how we can use the decision trees model to predict the target variable based on the given feature set.

It is essential to understand that Exploratory Data Analysis or EDA plays an essential role in the Machine Learning process as it takes you through the various steps of cleaning and preprocessing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data (number of rows), the attribute strength, and so on to revamp your model.

## **Conclusion**

Decision trees are widely used in machine learning and help you create algorithms that are robust and explainable. Analytics that generally needs to give out recommendations can use decision trees as the predictions are backed up by a ‘why’ factor, and model explainability is much higher as compared to the other algorithms.

In this chapter, the readers have gained insights into what is a decision tree, how to build one, and what are the various parameters to determine a good decision tree. We have discussed concepts like pruning, entropy, and information gain in the context of trees. Towards the end, we have worked on an end to end case study in python and applied the decision tree algorithm to the given dataset.

In the next chapter, we will be looking at a flavor of decision trees, which are random forests to understand how they contribute to machine learning.

## **Quiz - Chapter review**

1. What does entropy increase/decrease signify in terms of homogeneity?
2. What do the leaf nodes signify in a decision tree?
3. PRACTICE: Try and imagine a routine activity that you perform and create a Decision Tree workflow for the same.
4. What is the CART algorithm?
5. What are the different Python libraries that provide Decision Tree libraries for modeling?

# CHAPTER 9

## Random Forests

### Introduction

As we have already seen that decision trees are one of the most efficient machine learning algorithms, having multiple decision trees will make it more accurate when it comes to training the machine learning dataset.

Random forest is a versatile algorithm, and it can be used for both regression and classification. As today's industry juggles with multiple use cases of machine learning, there has been an increase in the allocation of computational power for these algorithms.

Increasing computational power has ensured that algorithms like a random forest with intensive mathematical equations have evolved to become one of the most basic machine learning algorithms in today's era.

### Structure

- What are random forests?
- The bagging technique
- Why are random forests better than a single decision tree?
- Industry use case of random forests
- Case study
- Quiz

### Objectives

The objectives of this chapter are as follows:

- Understand the working of random forest
- Learn to solve a machine learning problem using random forest
- Understand why random forests are better than single decision trees

- Implementing end-to-end Python code using random forest

## What are random forests?

Random forest is a combination of multiple random decision trees. As we have seen in the previous chapter, decision tree models are one of the most famous supervised learning algorithms in machine learning, both for regression and classification purposes.

Having multiple decision trees ensures model explainability and ease of interpreting a machine learning model.

So, how is the randomness created in a random forest? Here are two ways of creating randomness in a random forest:

- Each tree is built using a randomly chosen sample taken from our training dataset
- At each decision node, the tree is further split based on a series of randomly chosen attributes

Each of the decision trees built during a random forest algorithm creates a biased classifier because each decision tree is created using a random subset of the complete training dataset. These individual trees capture different trends in the various subsets of the data.

Each of these decision trees is like an individual expert, who, when put together with other experts, can give us the most accurate answers. An ensemble of these decision trees is used for either classification or regression problems in the random forest algorithm.

## Here is how we do it

- For classification, we take a majority keyboard of the majority tress and predict the class that has been identified by the majority of the decision trees.
- For regression, we can take an average of the value predicted by the ensemble of decision trees.

We can also assign different types of weights to the various decision trees based on our intuition or some validation from the cross-validation data.

## Hyperparameters used in random forest algorithm

Here are the significant hyperparameters used in the random forest:

- `ntree`: Number of decision trees in the random forest. The most commonly used value is 100. An increased number of trees could lead to overfitting of the data.
- `mtry`: Number of variables which are randomly sampled as selectors at each split for the decision tree.
- `replace`: Parameter that decides if sampling should involve replacement or not.

## The bagging technique

Bagging is a type of ensemble learning that helps reduce the variance of different kinds of models. Bagging stands for bootstrap aggregation. We know that random forests use an ensemble of decision trees, each of which produces an output, and therefore can be called **learners** who make independent errors. To ensure that independent errors are made by the learners, we can provide them with different data samples, different algorithm parameters, and so on.

In bootstrap aggregation/bagging, the sample sets are picked in such a way that they are different from each other.

The process of bagging works:

- Let us assume that we have a training dataset  $T$
- If we have  $n$  learner/decision trees, we can randomly select a sample size  $k$  (this sample size should be large enough to avoid too much overfitting)
- We will now repeatedly keep picking  $k$  elements from the training data, with replacement, to create  $T_1, T_2, T_3, \dots$  until  $T_n$  training sets are available
- The average of all the trees will be considered as the final prediction
- Due to the difference in sample subsets, the individual trees will not be correlated to each other and will be able to offer unique insights into

the data patterns

The critical differentiator in the process of bagging is that it lets us create multiple models that are making independent errors. If the errors are correlated, then this technique will not be successful. This point should be kept in mind while using bagging for training the dataset in random forests.

## **Why are random forests better than regular decision trees?**

To start with, a decision tree is built on an entire dataset, whereas the random forest uses several sampling and aggregation techniques and works on the part of the dataset. In a decision tree, the final decision is taken by one single tree, whereas in random forests, multiple trees get to cast their votes, and a combined decision is taken with the help of the casted votes.

There are two main reasons as to why random forests outperform decision trees:

- Decision trees can be easily overfitted as it is one single tree that decides the labels for your dataset.
- The random forest offers diversity in labeling. Since random forests consist of several different trees, it is easier to get diverse options in random forests as compared to decision trees.

In the next section, we will look at how random forest models are trained.

## **Random forest training**

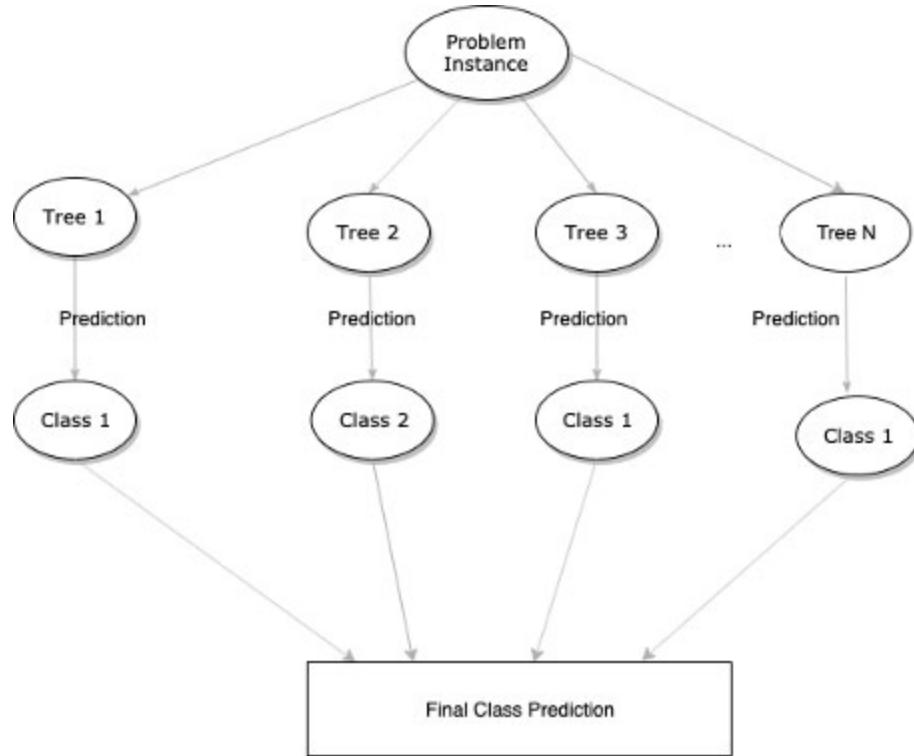
Random forests use the bootstrap aggregating or bagging method while training. As we have already seen in the bagging section, the whole dataset is split into several samples, and decision trees are created for each of these samples. Once we have the individual decision tree models, we will aggregate the results of those and predict the final results.

Here is a series of steps that gives us the life of a data point as it moves through a random forest:

- Similar to decision trees, each data point starts at the top node and goes to the next level based on its different characteristics.

- The difference in random forests is that the point will go through this classification process in all the different data samples that it would be present in.
- In the case of classification, the final class will be an aggregation of all the predictions by the different trees, and in regression, it would be the numerical average.
- Using the behavior of a particular data point, we can determine the critical features thereby making feature selection easier in random forests

Here is an image depicting the process of random forest training:



*Figure 9.1: An example of random forest classification*

As you can see in the above example, for a given problem instance, the random forest training consists of training several trees and then taking the final class prediction.

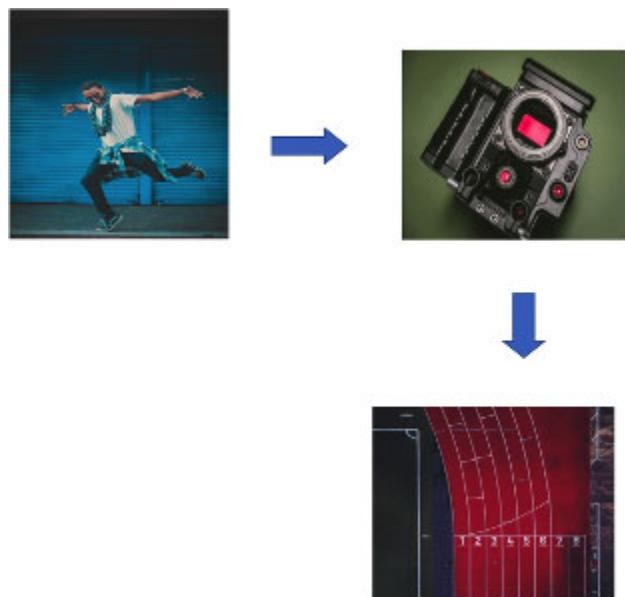
## Industry use case of random forests

If you are a gaming enthusiast, you would have heard of the Xbox. It is a gaming console product owned and developed by Microsoft. For motion

sensor related games that one can play on the Xbox, we need a motion sensor device named Kinect.

Kinect is an input device that can sense motion and provides the necessary outputs for the motion-sensing game to continue/scores based on the user's motions. The Kinect motion sensor tracks the movement of the various body parts using the Random Forest algorithm and re-creates it in the game console.

The following picture series depicts the flow of actions that happens when a movement is captured on Kinect:



**Figure 9.2:** Pictures showing example actions performed during a Kinect game

The above figure captures the first set of actions that happen, which are as follows:

1. The user performs a movement in front of the device
2. The Kinect sensor captures the action
3. It tries to calculate the accuracy of the user action and gives the user points accordingly

It is action number 3 stated above, which uses the random forest algorithm with the help of public data collected to train and recognize the user's different body parts. The random forest algorithm learns from this training and identifies the user's body parts and their corresponding movements made by those body parts.

In the end, the algorithm scores the user based on the accuracy of the movements of the body parts.

The above example is a classic use case of random forests where the several body parts and their corresponding images are being trained using random forest algorithm where a huge dataset containing several images is split into numerous samples, and each sample creates a decision tree, the aggregate of which is used as the final label.

In the next section, we will look at a case study which will show us an end to end implementation of random forests using Python.

## Case study – I

This case study will take us through the various steps that are involved in building a random f

Forest machine learning model. In this section, we will be focusing on a dataset that has one independent variable and several independent variables, as we will see in the data attributes section.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model.

## About the data

The dataset that we are using for this case study has been collected from an open-source website which provides several types of data sample for regression analysis, time series analysis, and many more. The data describes the auto insurance claims files in Sweden in thousands of Swedish Kronor for geographical regions.

This dataset has two columns—the dependent variable is the 'Number of Claims' and the independent variable is the 'Total Payment,' that is, the total payment that is likely to be paid based on the number of claims files. Our problem statement involves predicting the total payment to be paid, given the number of claims in Sweden for data collected over some time.

## Attribute information

The various attributes that we have gathered from the metadata file that was available along with the dataset file are:

```
Class : This is the type of wine
Alcohol : Alcohol percentage in the wine
Malic acid : Percentage of malic acid in this wine
Ash : Amount of ash present in the wine
Alkalinity of ash : Alkalinity of ash present in the wine
Magnesium : Amount of magnesium present in the wine
Total phenols : Total phenols present in the wine
Flavanoids : Amount of flavanoids present in the wine
Nonflavanoid phenols : Percentage of nonflavanoid phenols present in the wine
Proanthocyanins : Percentage of proanthocyanins present in the wine
Color intensity : Colour intensity of the wine
Hue : Hue of the wine
OD280/OD315 of diluted wines : OD280/OD315 of the diluted wines
Proline : Amount of proline present in the wine
```

Each type of wine will contain a different combination of these attributes that are stated above. In the next steps, we will build a model that will classify a wine based on its characteristics (these characteristics are the attributes mentioned above).

## Python code and step-by-step regression analysis

The following steps have a python code snippet and its corresponding output in most of the cases. You can type the following code in your Jupyter Notebook/A google colab notebook or on your local editor and execute them to see the desired results:

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
from sklearn import metrics
```

```

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

```

2. Read the data (which is stored in the .data format) using `python-pandas` and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `wine.csv`. I have stored the datasets under the `/RandomForests` folder under the `/Datasets` folder on my local system.

```

df = pd.read_csv('/Datasets/RandomForests/wine.data')
df.head()

```

Here is the output of the head command initially:

	1	14.23	1.71	2.43	15.6	127	2.8	3.06	.28	2.29	5.64	1.04	3.92	1065
0	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
1	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
2	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
3	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735
4	1	14.20	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450

*Figure 9.3: Output of head*

In the next step, we will assign the attributes of the names that we have got from the metadata file.

You can use the following attribute names to identify your columns quickly.

3. In this step, we will create an array with the columns names of the various columns present in the dataset (we get the columns names from the metadata given to us)

```

attributes = ["Class", "Alcohol", "Malic acid", "Ash",
"Alcalinity of ash", "Magnesium", "Total phenols",
"Flavanoids", "Nonflavanoid phenols", "Proanthocyanins",
"Color intensity", "Hue", "OD280/OD315 of diluted wines",
"Proline"]

```

4. In this step, we will assign the names of the columns to the columns variable of our dataset.

```
df.columns = attributes
```

5. We can see that the `df.head()` output now reflects the names of the assigned columns:

```
df.head()
```

Here is the output of the head command after updating the attribute names:

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
1	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
2	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
3	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735
4	1	14.20	1.76	2.45	15.2	112	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450

*Figure 9.4: Output of head command after updating the attribute names*

In the next step, we will look at the different data types of the attributes.

6. Analyze the data by looking at the data types. An understanding of the various data types gives us an insight into the kind of data we are working with:

```
df.dtypes
```

**Output:**

```
Class                      int64
Alcohol                     float64
Malic acid                  float64
Ash                         float64
Alcalinity of ash           float64
Magnesium                   int64
Total phenols                float64
Flavanoids                   float64
Nonflavanoid phenols        float64
Proanthocyanins              float64
Color intensity               float64
Hue                          float64
OD280/OD315 of diluted wines float64
Proline                      int64
dtype: object
```

7. As most of our columns are numerical, we will use the describe function to understand more about the data:

```
df.describe()
```

Here is the output for `df.describe()`:

	Class	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue
count	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000	177.000000
mean	1.943503	12.993672	2.339887	2.366158	19.516949	99.587571	2.292260	2.023446	0.362316	1.586949	5.054802	0.95696
std	0.773991	0.808808	1.119314	0.275080	3.336071	14.174018	0.626465	0.998658	0.124653	0.571545	2.324446	0.22913
min	1.000000	11.030000	0.740000	1.360000	10.600000	70.000000	0.980000	0.340000	0.130000	0.410000	1.280000	0.48000
25%	1.000000	12.360000	1.800000	2.210000	17.200000	88.000000	1.740000	1.200000	0.270000	1.250000	3.210000	0.78000
50%	2.000000	13.050000	1.870000	2.360000	19.500000	98.000000	2.350000	2.130000	0.340000	1.550000	4.680000	0.96000
75%	3.000000	13.670000	3.100000	2.560000	21.500000	107.000000	2.800000	2.860000	0.440000	1.950000	6.200000	1.12000
max	3.000000	14.830000	5.800000	3.230000	30.000000	162.000000	3.880000	5.080000	0.660000	3.580000	13.000000	1.71000

*Figure 9.5: Output of df.describe()*

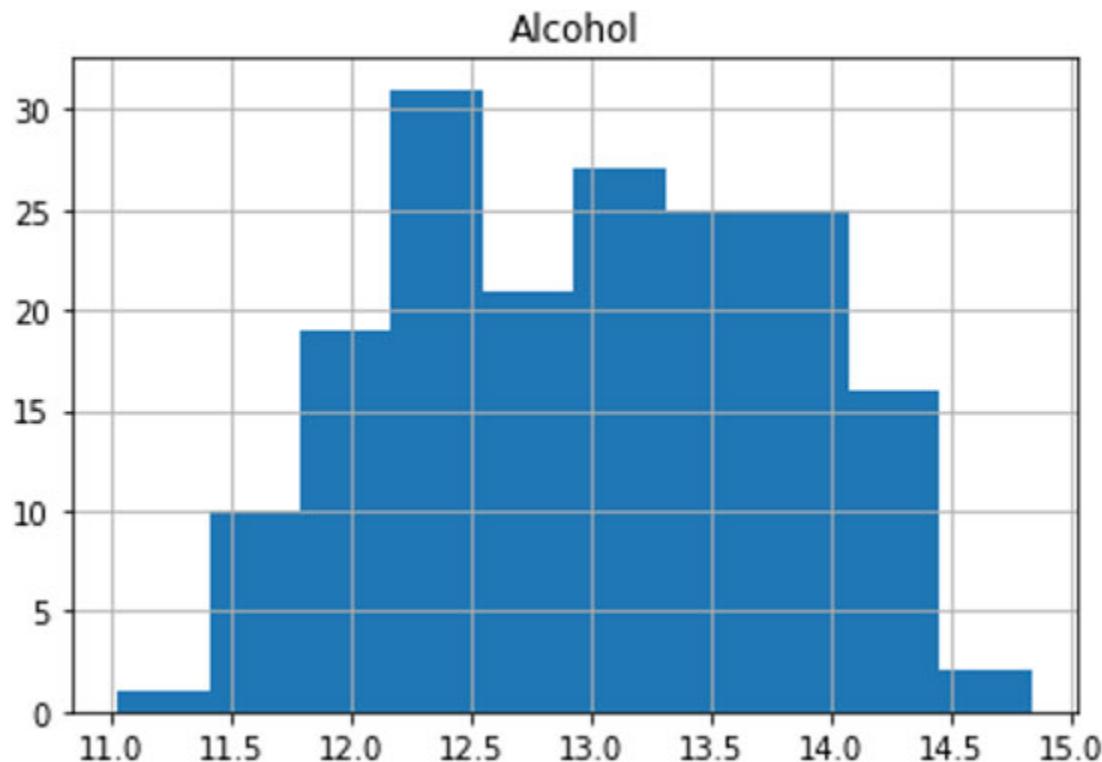
Once we have some necessary information about our data, we will try to visualize the information in the form of graphs in the upcoming steps.

8. In our next steps, we will explore the various data plots and visualizations that will help us understand the data much better.

Here is the command that will output a histogram for the variable 'Alcohol', which is one of our attributes in the dataset:

```
df.hist('Alcohol')
```

Here is a histogram image for the 'Alcohol' variable showing us the distribution of the variable in the dataset:



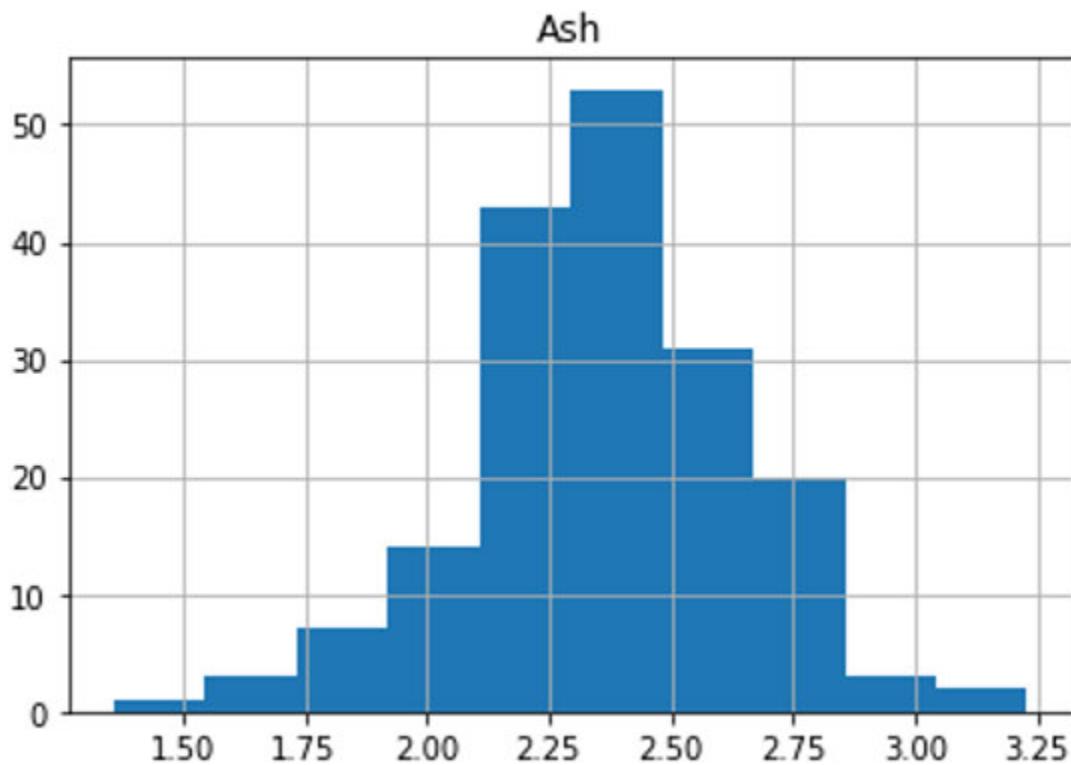
**Figure 9.6:** Histogram plot for variable Alcohol

Our next histogram is that of another attribute named 'Ash', present in our dataset.

Here is the command that will output a histogram for the variable 'Ash', which is one of our attributes:

```
df.hist('Ash')
```

Here is a histogram image for the 'Ash' variable showing us the distribution of the variable in the dataset:

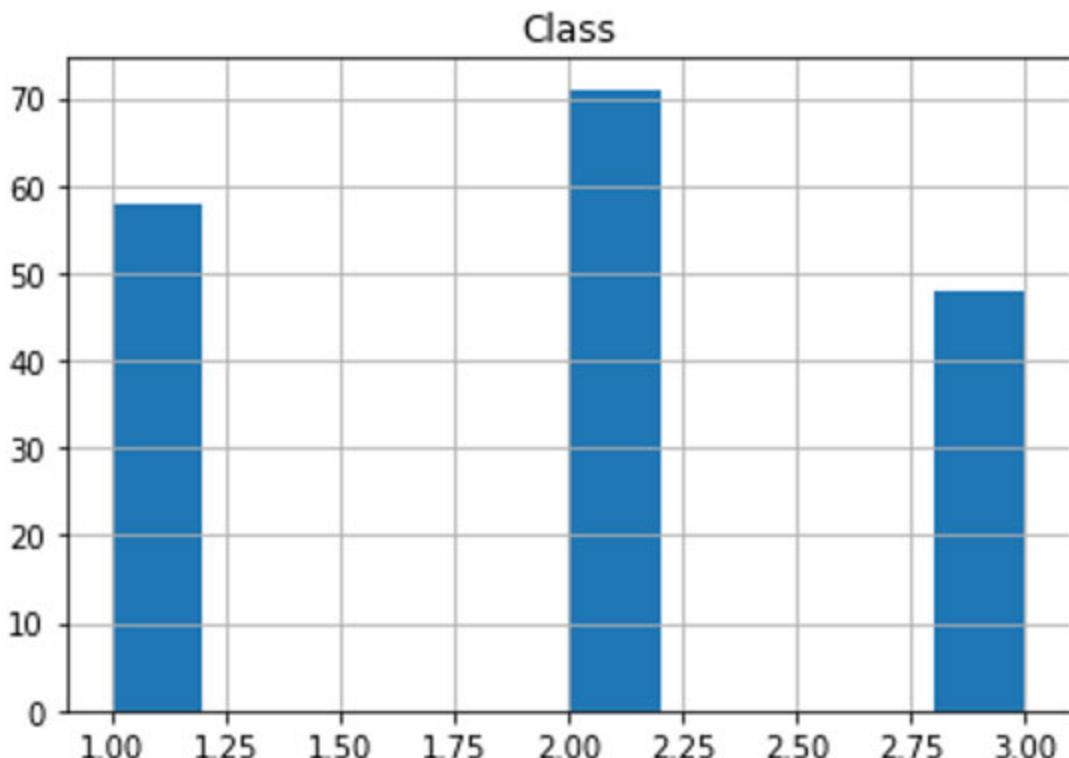


*Figure 9.7: Histogram plot for variable Ash*

Here is the command that will output a histogram for the variable 'Class', which is our target variable:

```
df.hist('Class')
```

Here is a histogram for the 'Class' variable showing us the distribution of the variable in the dataset:



*Figure 9.8: Histogram plot for target variable Class*

Our next step would be to understand the data distribution using pair plots.

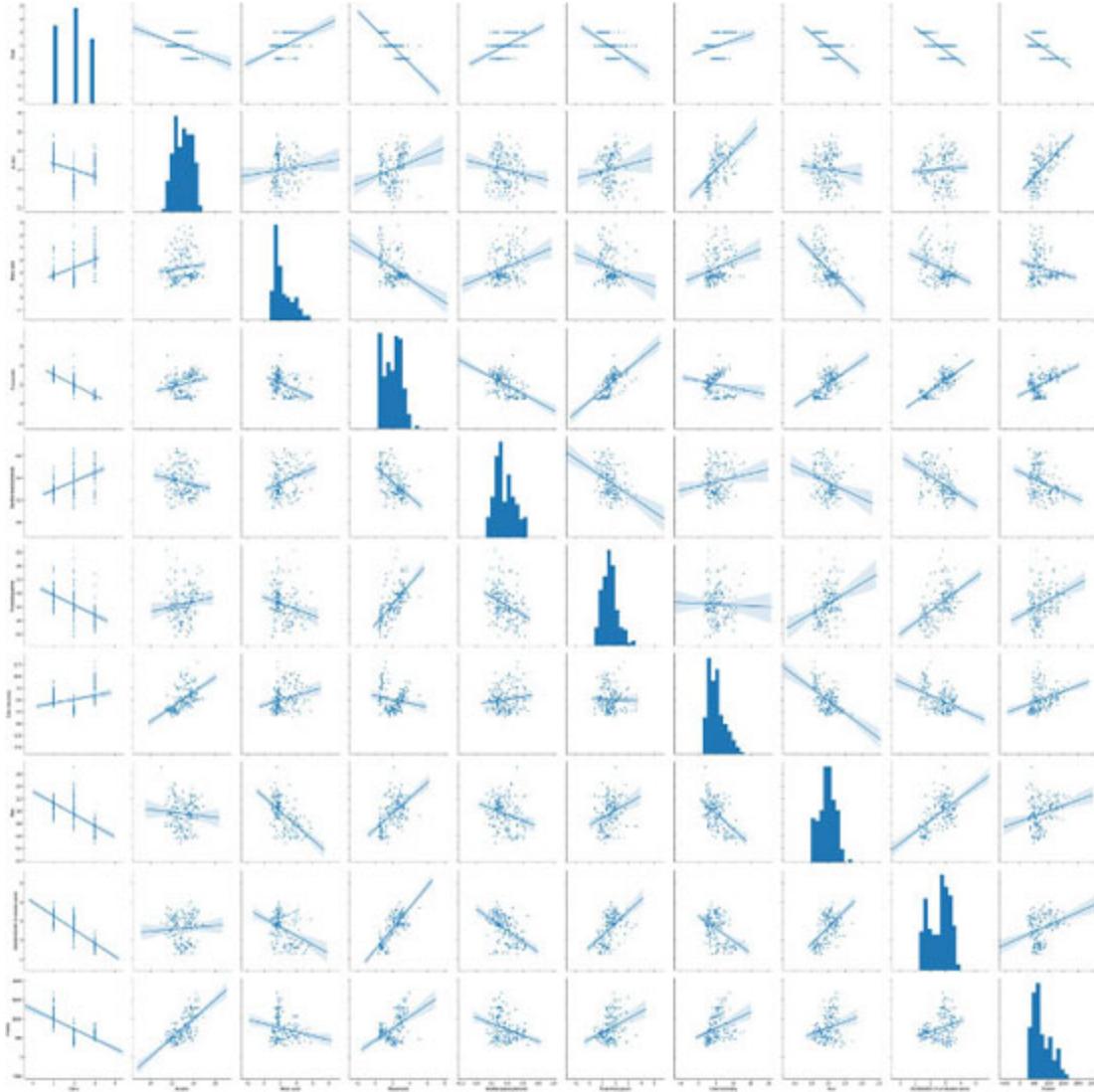
9. Our next step is to plot pair plots, which will show several graphs with the features paired together. Pair plots will give us a good insight into how the features in our dataset vary from each other.

Our pair plots help us understand the correlations and behavior between the various attributes in the dataset.

Here is the command that will create pair plots between our class attributes:

```
df_n=df[["Class", "Alcohol", "Malic acid", "Flavanoids",
"Nonflavanoid phenols", "Proanthocyanins", "Color
intensity", "Hue", "OD280/OD315 of diluted wines",
"Proline"]]
sns.pairplot(df_n, height=4, kind="reg", markers=".")
```

Here is the output image of the pair plots:



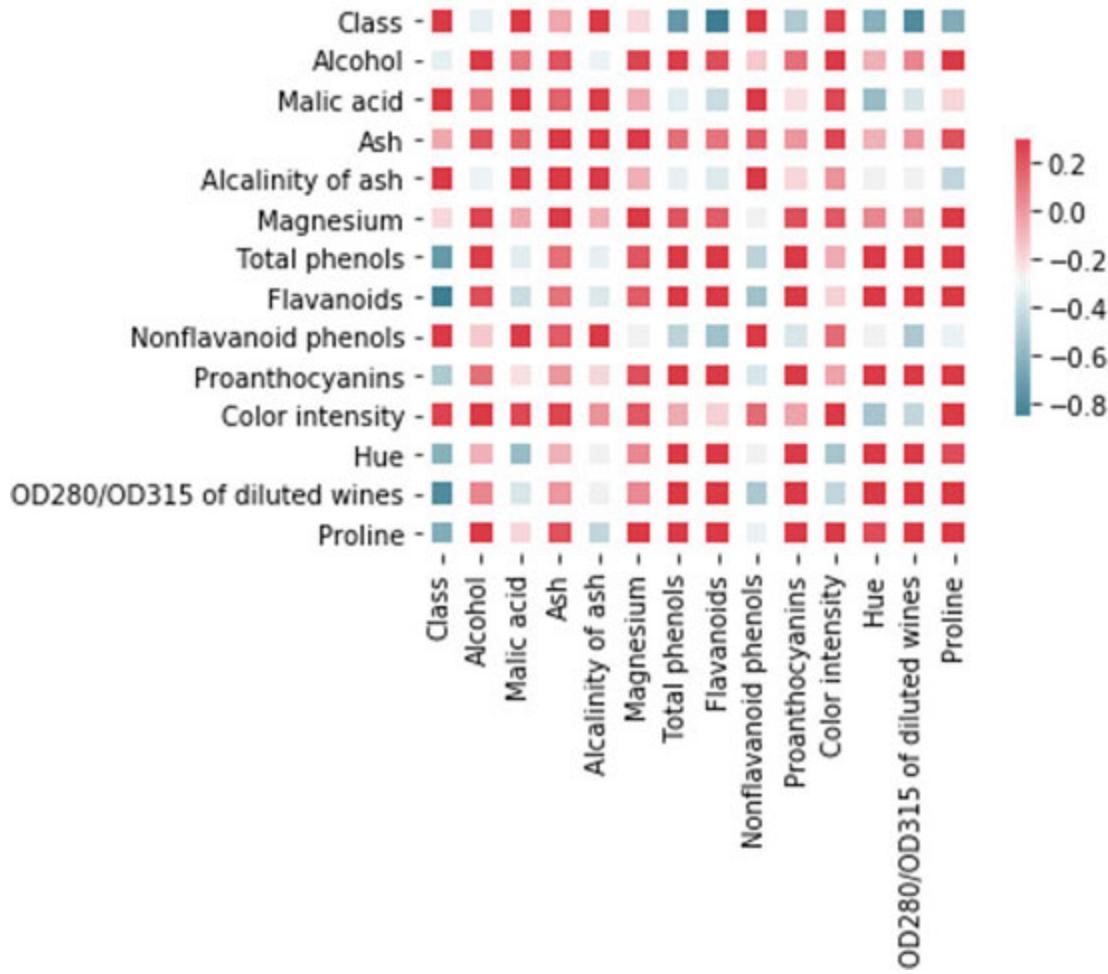
*Figure 9.9: Pair plots between the attributes of our dataset*

In the next step, we will examine a vital feature called correlation between the attributes.

10. In this step, we will create a correlation matrix to look at the correlation between the various attributes in our dataset:

```
corr = df.corr()
cmap = sns.diverging_palette(220, 10, as_cmap=True)
sns.heatmap(corr,cmap=cmap,
vmax=.3,square=True,linewidths=6, cbar_kws={"shrink": .5})
colormap = plt.cm.viridis
```

Here is the image that displays the correlation matrix:



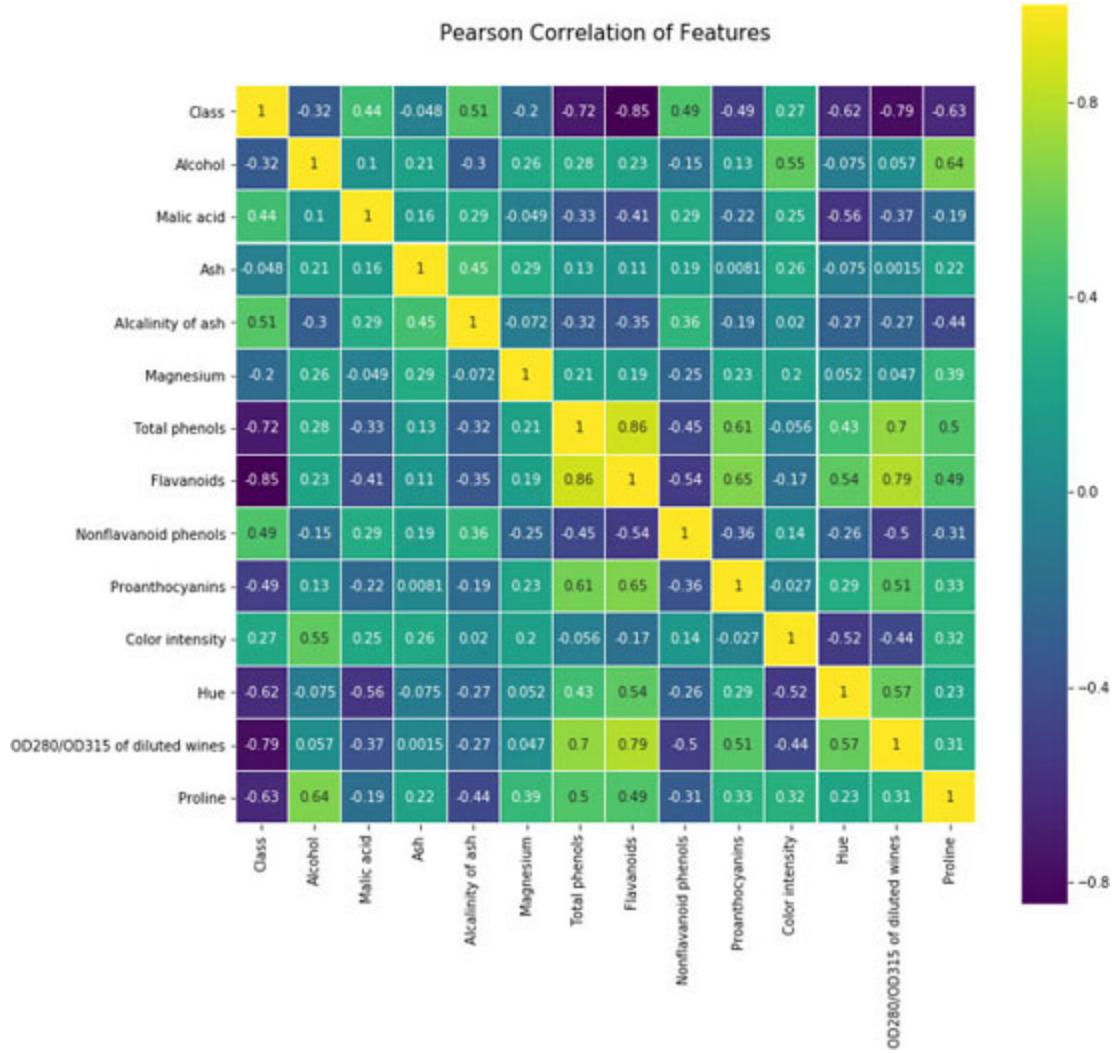
**Figure 9.10:** Correlation matrix for the wine dataset

In the next step, we will look at another correlation matrix, which will give us a better understanding of the correlation between the various variables present in the dataset.

11. In this step, we will plot another correlation matrix which has slightly different dimensions in order to get a better insight into our dataset's feature correlation:

```
plt.figure(figsize=(12,12))
plt.title('Pearson Correlation of Features', y=1.05,
size=15)
sns.heatmap(df.corr(), linewidths=0.1, vmax=1.0, square=True,
cmap=colormap, linecolor='white',
annot=True)
```

Here is the output image for correlation:



**Figure 9.11:** Correlation heatmap – 2 for the given wine dataset

In the next step, we will separate the target variable from the other attributes to build our model.

12. We will now separate our independent variable and dependent variables into separate columns to feed them to our training model:

```
df = df.drop(columns=['Flavanoids'])
Y = df['Class']
X = df.drop(columns=['Class'])
```

13. We will split the data into training and test data. In this step, the `test_size` determines the portion of data from our current dataset that will be used for testing. In this case, it is 33%:

```
x_train, x_test, y_train, y_test = train_test_split(X, Y,  
test_size=0.33, random_state=42)
```

14. In the next code step, we will create our RandomForestClassifier:

```
classifier = RandomForestClassifier(n_jobs=2,  
random_state=42)
```

15. In the next code step, we will fit our classifier to the existing training dataset. We can see that the output gives us the classifier parameters that have been applied to the dataset.

```
classifier.fit(x_train, y_train)
```

### Output:

```
RandomForestClassifier(bootstrap=True, class_weight=None,  
criterion='gini',  
max_depth=None, max_features='auto',  
max_leaf_nodes=None,  
min_impurity_decrease=0.0, min_impurity_split=None,  
min_samples_leaf=1, min_samples_split=2,  
min_weight_fraction_leaf=0.0, n_estimators=10,  
n_jobs=2,  
oob_score=False, random_state=42, verbose=0,  
warm_start=False)
```

16. In this step, we will use the test dataset and predict the labels for the training dataset. We will also try to understand using the accuracy parameter as to what is the accuracy of our prediction algorithm:

```
#Predict the response for test dataset  
y_pred = classifier.predict(x_test)  
# Model Accuracy  
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

### Output:

Accuracy: 0.9491525423728814

17. In this step, we will look at the confusion matrix to gain some insights into our results.

We are creating a confusion matrix. The cumulative number of right and wrong predictions are present in the confusion matrix. They are summarized with count values and broken down by each class.

The first entry is the True Positives (Target variable is positive and predicted correctly), the next entry to the right is False Negatives (Target variable is positive but predicted as negative), the entry below true positives is False Positives (Target variable is negative but predicted as positive), and the entry next to that is True Negatives (Target variable is negative and predicted correctly).

```
confusion_matrix(y_test, y_pred, labels=None,  
sample_weight=None)
```

#### Output:

```
array([[20, 0, 0],  
       [0, 22, 3],  
       [0, 0, 14]])
```

## Case study conclusion

From the above case study, we can conclude that for the dataset that we have used (wine dataset), the random forest algorithm for predictive modeling can give us a good accuracy while predicting the labels.

The accuracy of our model is close to 95%, which is a high percentage when it comes to accuracy. The end to end python case study helps us understand the different python commands and libraries that can help us explore the dataset.

It is essential to understand that Exploratory Data Analysis or EDA plays a vital role in the machine learning process as it takes you through the various steps of cleaning and pre-processing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data (number of rows), the attribute strength, and so on, to revamp your model.

## Summary

In this chapter, we did a deep dive into the random forest algorithm to understand the algorithm and its use cases. We could get an understanding of why random forests are better than a single decision tree and how the models are trained. We walked through an end to end implementation of a random forest algorithm in python using an open-source dataset.

After having gone through this chapter, the readers should be able to understand and differentiate between use cases where random forests can be useful and where they might consume more time as compared to the traditional algorithms. This chapter also helps you understand how to create visualizations in python and use the random forest algorithm for prediction on an existing dataset.

In the next chapter, we will look at a new form of supervised learning, which is applied to data that is collected over time. We will look at what is time series and how it can help us forecast useful information based on the past data collected.

## Quiz

1. Is there a limit on the number of trees that can be created in random forests?
2. How does random forest predict results for both classification and regression?
3. Why is the bagging technique useful in random forests?
4. Give us another real-time use case of random forests and how they are being used to solve real-time problems in the industry.
5. What is the difference between a histogram and a scatter plot?

# CHAPTER 10

## Time Series Models in Machine Learning

### Introduction

The idea of time series is age-old, and we have been practicing it since the inception of time itself. Several straightforward examples of time series are using the patterns created by sunlight to predict the time of day (in the olden days), determining the approximate time of eclipse by observing the changing patterns of the moon, using forecasting to predict the share price of stocks based on their rise and fall pattern, evolutionary changes in the ecosystem, and so on.

Time series is a collection of data that was obtained over some time. The data could have been collected periodically at specific intervals (like performance data of a system) or continuously (like continuous sensor data). Time series analysis, also known as trend analysis helps identify trends or patterns in data over time. The important thing about time series data is that it should have been collected at consistent intervals to get the best results from the analysis.

Time series analysis helps us uncover the underlying parameters leading to a particular trend, and the collected data points help forecast specific patterns in the future by fitting appropriate models for these patterns.

### Structure

- Analysis methods in time series
- Seasonal and cyclic patterns in time series
- Moving average model
- ARIMA
- Time series case study

- Conclusion
- Quiz

## Objectives

- In this chapter, the reader will be able to understand the various patterns that they can observe in a time series data. They will also be able to understand how data needs to be collected to analyze it efficiently.
- Readers will be able to understand basic mathematical concepts used in time series like moving averages and solve an end to an end case study in Python.

## Analysis methods in time series

There are several methods of analysis that can be performed on a time series dataset based on the output that we want and based on the trends that the data shows. Here is a glimpse of some methods of analysis that are used in time series:

- **Descriptive analysis:** Using graphical methods to understand and determine the trends or patterns in a time series. Graphs or other tools help give us a visual picture of how the various trends or patterns look like. This descriptive analysis can help us identify various cyclic patterns, the overall trends in data, outliers, and so on.
- **Spectral analysis:** The idea is to decompose a stationary time series into a combination of sinusoids. This kind of analysis helps separate the periodic and cyclic components in a time series dataset. It is also known as the **frequency domain**.
- **Forecasting:** It is a predictive analysis method based on historical trends in data. It is extensively used in business forecasting, budgeting, and many more.
- **Intervention analysis:** Intervention analysis is used to understand the effect of an intervention on a time series, that is if a particular event has triggered a change of pattern in the time series. One of the primary use cases of this is incorporated where the change in the appraisal

system can sometimes cause an employee to perform better or worse, which can be tracked using intervention analysis.

- **Explanative analysis:** Studies the cross-correlation or relationship between two-time series and the dependence of one on another. For example, the study of employee turnover data and employee training data to determine if there is any dependence on employee training programs on employee turnover rates over time.

Each of these methods is applied to the dataset based on the type of data that we collect. In the next section, we will look at the different patterns that we observe in a time series dataset.

## Seasonal/periodic and cyclic patterns in time series

The two main types of patterns that are found in a time series dataset are seasonal or periodic patterns and cyclic patterns. We will take a detailed look at what these patterns are and some practical examples of these patterns.

### Seasonal patterns

A time series is known as a seasonal or periodic time series if the same behavior repeats over some time, and the periodic interval is fixed/known. The series is influenced by seasonal changes that contribute to the deviation in the data patterns.

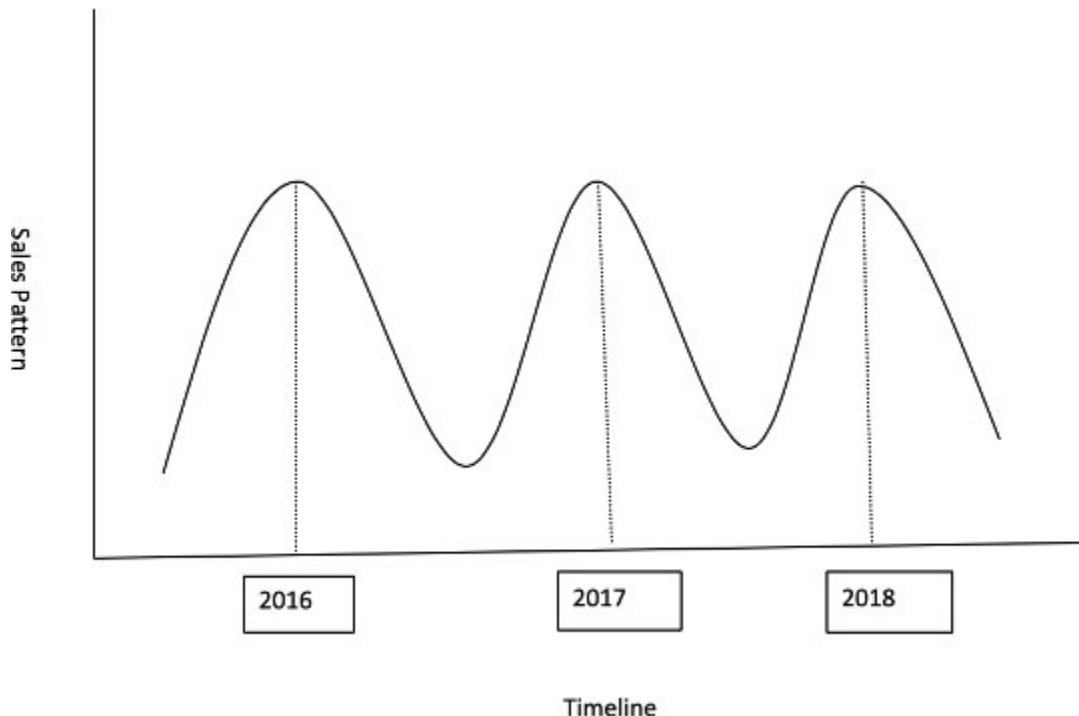
There can be several types of seasons that your time series data may contain; some of the examples are:

- Changes every quarter
- Changes every month
- Changes every day

An example would be: You are an ice cream company that is noticing fall and rise in the sales of ice cream during the same seasons across the year, and you decide to collect the sales pattern data to observe over three years.

You observe that data collected in the year 2016, 2017 and 2018 shows an increase in sales in the months of summer and a decrease in sales in the months of winter. To apply either or the ARMA or ARIMA models, we need to remove the seasonality. We will show you in the future sections as to what are ARMA and ARIMA models and how we can implement time series analysis using the ARIMA model.

Here is an example graph of how a seasonal pattern might look like for the ice cream sales pattern:



*Figure 10.1: Seasonal Sales pattern for ice cream sales*

**Seasonal patterns are patterns that need to appear within one year**

## Cyclic patterns

Cyclic patterns are mostly patterns that do not have fixed periods. We will see various patterns spanning across years but not necessarily foresee what the fixed periods are going to look like. These are also known as non-seasonal patterns.

For example:

**Business cycles for certain products that are being revamped are not known beforehand.**

When trying to identify patterns in the time series, users are sometimes unsure of the difference between cyclic patterns and seasonal patterns, and that can be a deterrent to your analysis. The simple thing to remember is that if the period is fixed and is associated with an aspect of the calendar, then it's seasonal and if it's not fixed, but there's a pattern over a more extended period, it's cyclic.

## Moving average model

Moving average model is a famous model used in the time series analysis. We will explain the moving average model using an example.

Let's say that you are a student at a university, and every month, your professor has a meetup on a new technology topic wherein he assigns you to bring a certain number of doughnuts for every meetup. Let's say the number of doughnuts you bring is = 10.

We will represent that by using  $\mu$ :

$$\mu = 10$$

But every month, your professor points out that you have brought the wrong number of doughnuts by 'x' amount. That can be represented in the form of et:

*et = The number of doughnuts which are extra or less*

The function equation to calculate moving averages is as follows:

$ft' = \mu + \Theta_1 * et-1$  (Mean + Standard Coefficient \* previous error)

\*t: Current time period

\*t-1: Previous time period

$\Theta_1 = 0.5$  (50% of the previous error is being considered)

The following table gives us an understanding of how moving averages work, with the help of a practical example which we will walk through in the next section:

t	ft'	et	ft
1	10	-2	8

2	9	1	10
3	10.5	0	10.5
4	10	2	11
5	11	1	12

**Table 10.1:** Moving averages explained

**ft'**: Number of doughnuts you brought based on the error in the previous period

**ft**: Number of doughnuts you should have got during that period

**et**: Erroneous number of cupcakes

In the next section, we will walk through an example of moving averages step by step with a practical example.

## Walking through Table 10.1

The following steps show us how moving averages are calculated for a problem using a real-time example where you are asked to bring a certain number of doughnuts to a party that happens every fortnightly. The time variable (t) represents every fortnight as we go through multiple iterations.

1. t = 1

The ft' at t=1 remains ten as you have just begun the experiment, that is, you bring ten doughnuts as asked by the professor, but the professor says that you should have brought two less, and therefore, your ft becomes 8.

We will use this error to calculate the ft' in *Step 2*.

2. t = 2

$$ft' = 10 + (0.5 * (-2)) = 9$$

et = 1, that is, it was observed that you brought one doughnut less to the party

Therefore,  $ft = 9 + 1 = 10$

3. t = 3

$$ft' = 10 + (0.5 * (1)) = 10.5$$

$\epsilon_t = 0$ , that is, it was observed that you brought one correct number of doughnuts to the party.

Therefore,  $f_t = 10.5 + 0 = 10.5$

From the table, you can see that we have calculated the moving averages of 5 terms starting from time  $t=1$  to time  $t=5$ .

Moving averages are useful when it comes to time series as they help in analyzing data while being able to create averages of different data points throughout the time series distribution. The example we shared above, calculates the moving average of a subset of data that contains five observations.

Moving averages help smoothen the short term deviations/fluctuations in data and help focus on the long term trends and highlights.

**INFO:**  $\epsilon_t$  is normally distributed between  $e(\text{mean} = 0)$  and  $e(\text{standard deviation} = 1)$

## ARIMA

One of the most frequently used models in time series to forecast data and analyze the time series dataset to derive insights in the form of patterns and trends. ARIMA stands for **AutoRegressive Integrated Moving Average**.

The critical elements of the model, described by the acronym, are:

- **AutoRegression:** A model that uses the relationship between an observation and some observations that happened before time.
- **Integrated:** The methodology of subtracting the observation values to result in a series that is more stationary and uniform.
- **Moving Average:** As we have already seen in the previous section, the moving average helps you calculate the mean of observations in subsets throughout the time series.

Each of the entities mentioned above is mentioned as parameters in the ARIMA model. The parameters are generally known as p, d, and q.

The ARIMA parameters are defined below:

- **p**: The number of lag observations included in the model, also called the lag order.
- **d**: The number of times that the raw observations are differenced also called the degree of difference.
- **q**: The size of the moving average window, also called the order of moving average.

Now that we know what an ARIMA model is, and what are its associated parameters, we will look at how the model works in the upcoming section.

## How does it work?

Here are the steps to perform forecasting using ARIMA models:

- If necessary, stationaries the series. Stationarity means that the mathematical properties of a time-driven event do not change too much over some time.
- Plot the autocorrelations and partial autocorrelation plots and observe the lags.
- Check if the errors are significant and if they need to be included while forecasting.
- Fit the model and check for the errors
- Plot the forecasted pattern versus the original pattern

ARIMA models are often used for forecasting when it comes to time series.

## Forecasting

Forecasting is the ability to predict future information based on the information that we have collected over a specified period. Time series models like ARIMA give us the ability to analyze the data collected over some time and make specific predictions.

One of the most common examples of time series is annual sales revenue. In an organization, it is essential to understand when the sales revenue would be high, and when would it be low so that the efforts of the sales teams can be calculated for each season.

There are also other trends like employee performance over some time, a company's growth over some time, an organization's attrition rate, and so on. Forecasting is slowly time-dependent, and hence, it is different from other models in a sense where the other models use any kind of data(numerical/categorical) to predict the output, but the forecasting models can use only the time series.

The following case study will help you understand how the ARIMA models are implemented for real-time data and how we can use the models to forecast the values for the upcoming periods.

## **Time series case study**

This case study will take us through the various steps that are involved in building an ARIMA Forecasting. In this section, we will be focusing on a time series dataset.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building, and looking at the forecasted results by our ARIMA model.

## **About the data**

Our dataset describes the minimum daily temperatures over ten years (1981-1990) in the city of Melbourne, Australia. The units are in degrees Celsius, and there are 3,650 observations. The source of the data is credited as the Australian Bureau of Meteorology.

## **Step by step Python Analysis for an ARIMA based time series forecasting model**

It is one of the most crucial and essential sections of our case study—being able to analyze the data using python libraries and deriving insights from the data.

The following steps have a python code snippet and its corresponding output in most of the cases. You can type the following code in your Jupyter Notebook/A google colab notebook or on your local editor and execute them to see the desired results.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import numpy as np
import pandas as pd
from pandas import DataFrame
import matplotlib.pyplot as plt
import seaborn as sns

#model related libraries
from statsmodels.tsa.arima_model import ARIMA
from pandas.plotting import autocorrelation_plot
from sklearn.metrics import mean_squared_error
from statsmodels.graphics.tsaplots import plot_pacf
%matplotlib inline
```

2. Read the data (which is stored in the .csv format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is MinimumDailyTempratures.csv. I have stored the datasets under the /TimeSeries folder under the /Datasets folder on my local system:

```
df =
pd.read_csv('Datasets/TimeSeries/MinimumDailyTemperatures.csv', index_col=['Date'])
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

Here is the output image of the df.head() command:

```

<bound method NDFrame.head of
Date
01/01/81  20.7
02/01/81  17.9
03/01/81  18.8
04/01/81  14.6
05/01/81  15.8
06/01/81  15.8
07/01/81  15.8
08/01/81  17.4
09/01/81  21.8
10/01/81  20.0
11/01/81  16.2
12/01/81  13.3
13/01/81  16.7
14/01/81  21.5
15/01/81  25.0
16/01/81  20.7
17/01/81  20.6
18/01/81  24.8
19/01/81  17.7
20/01/81  15.5
21/01/81  18.2
22/01/81  12.1
23/01/81  14.4
24/01/81  16.0

```

*Figure 10.2: Output of df.head()*

In the next step, we will take a look at the column names for this dataset.

4. You can look at the column names in the dataset by using the following command:

`df.columns`

### Output:

`Index(['Temp'], dtype='object')`

5. To check if any missing or null values are present in the dataset, we use the `info()` method on our DataFrame:

`df.info()`

### Output:

```

<class 'pandas.core.frame.DataFrame'>
Index: 3650 entries, 01/01/81 to 31/12/90
Data columns (total 1 columns):

```

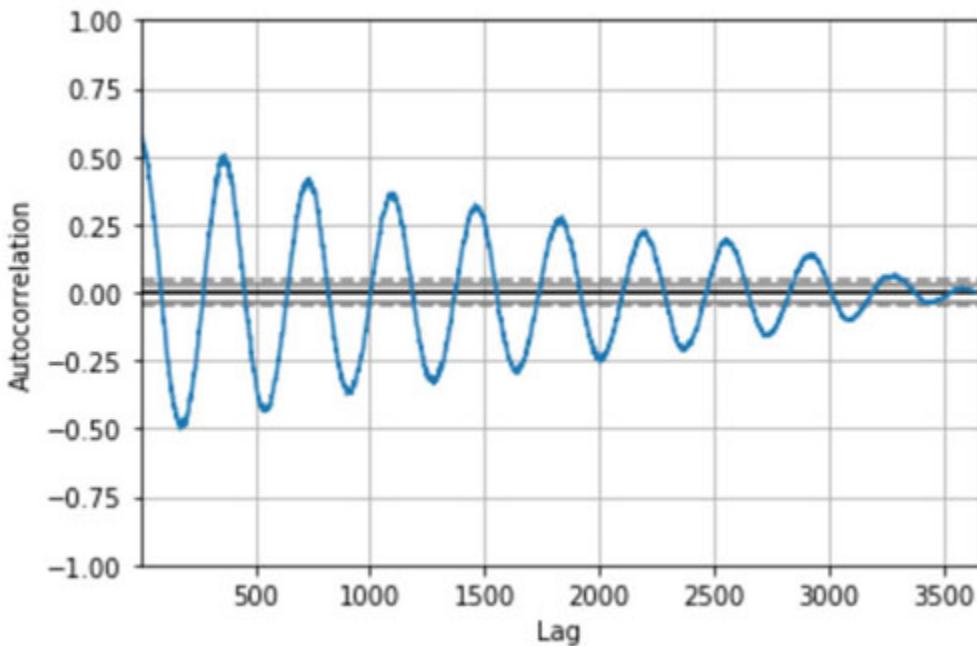
```
Temp 3650 non-null float64  
dtypes: float64(1)  
memory usage: 57.0+ KB
```

6. As we saw in *Step 5*, all of our columns are non-null, so we can efficiently skip the null check for our dataset. The next thing that we will need to plot for our dataset is an autocorrelation plot and a partial correlation plot.

**Autocorrelation and Partial Autocorrelation:** It describes the relationship between an observation at one point in time with observations in the past after eliminating the relationships of intervening observations:

```
autocorrelation_plot(df)  
plt.show()
```

Here is the image that represents the autocorrelation plot for our time series dataset:



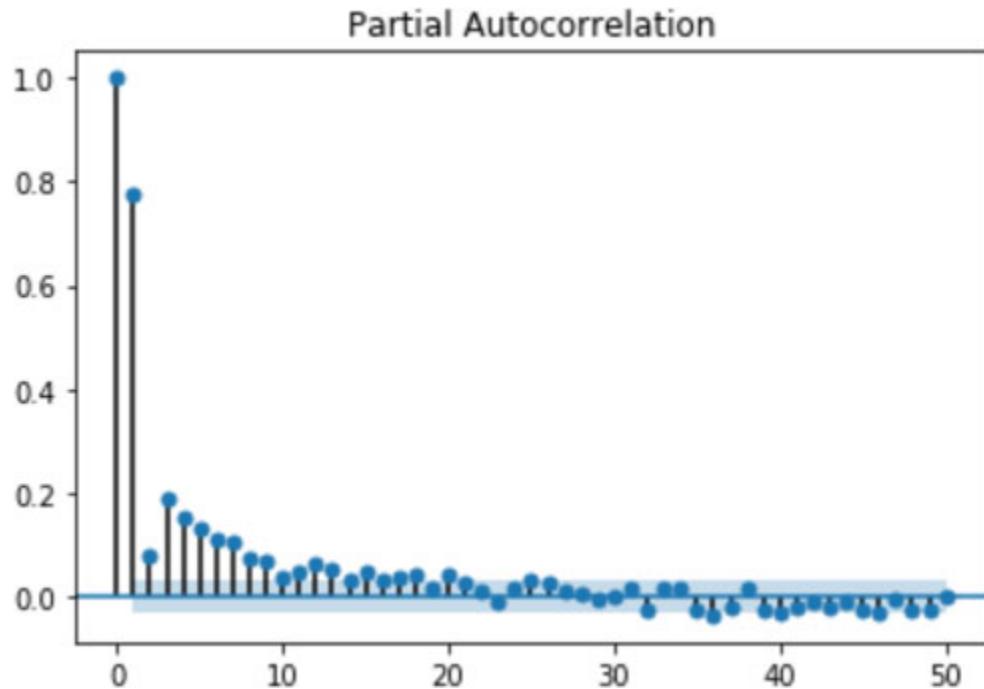
*Figure 10.3: Autocorrelation in time series*

In the next step, we will take a look at the partial autocorrelation graph for our dataset:

```
plot_pacf(df, lags=50)
```

```
plt.show()
```

Here is the image that represents the partial autocorrelation plot for our time series dataset:



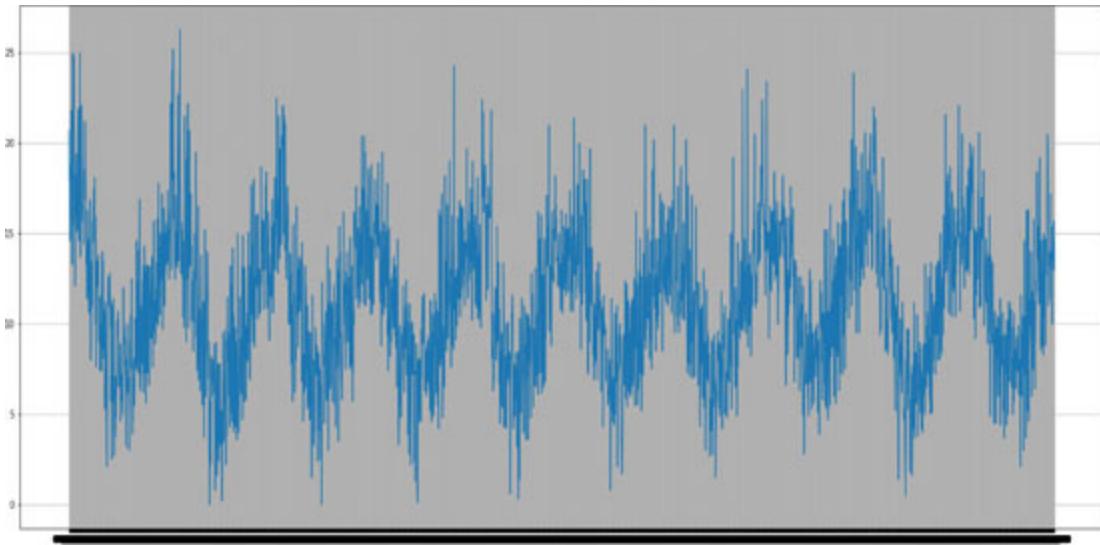
*Figure 10.4: Partial autocorrelation in time series*

In the next step, we will visualize the way data is distributed across the dataset.

7. We will now visualize the data to look at how the data is distributed across our dataset. We will create a plot of our columns to look at the trend in our time series data:

```
plt.figure(figsize=(15, 7))
plt.plot(df.Temp)
plt.grid(True)
plt.show()
```

Here is a plot that shows the time series representation of the patterns in our dataset:



*Figure 10.5: Time series representation*

In the above representation, the data looks more or less stationary, which is an expected observation in case of autoregressive model.

In the next step, we will build an ARIMA model for our dataset.

8. The next step for us is to build an ARIMA model using our dataset. We will use the ARIMA library from `statsmodels.tsa.arima_model`. The (p,d,q) order of the model that we are using currently is (5,1,0):

```
model = ARIMA(df, order=(5,1,0))
model_fit = model.fit(disp=0)
print(model_fit.summary())
```

The following figure represents the ARIMA model that we have created for our dataset:

ARIMA Model Results						
Dep. Variable:	D.Temp	No. Observations:	3649			
Model:	ARIMA(5, 1, 0)	Log Likelihood	-8495.810			
Method:	css-mle	S.D. of innovations	2.482			
Date:	Tue, 10 Sep 2019	AIC	17005.620			
Time:	12:45:14	BIC	17049.036			
Sample:	1	HQIC	17021.082			
<hr/>						
	coef	std err	z	P> z	[0.025	0.975]
const	-0.0013	0.017	-0.076	0.940	-0.035	0.033
ar.L1.D.Temp	-0.3358	0.016	-20.469	0.000	-0.368	-0.304
ar.L2.D.Temp	-0.3911	0.017	-23.037	0.000	-0.424	-0.358
ar.L3.D.Temp	-0.2942	0.018	-16.804	0.000	-0.329	-0.260
ar.L4.D.Temp	-0.2077	0.017	-12.235	0.000	-0.241	-0.174
ar.L5.D.Temp	-0.1361	0.016	-8.298	0.000	-0.168	-0.104
<hr/>						
	Roots					
	Real	Imaginary	Modulus	Frequency		
AR.1	0.7142	-1.1395j	1.3448	-0.1609		
AR.2	0.7142	+1.1395j	1.3448	0.1609		
AR.3	-1.6724	-0.0000j	1.6724	-0.5000		
AR.4	-0.6410	-1.4207j	1.5586	-0.3175		
AR.5	-0.6410	+1.4207j	1.5586	0.3175		
<hr/>						

*Figure 10.6: ARIMA model results*

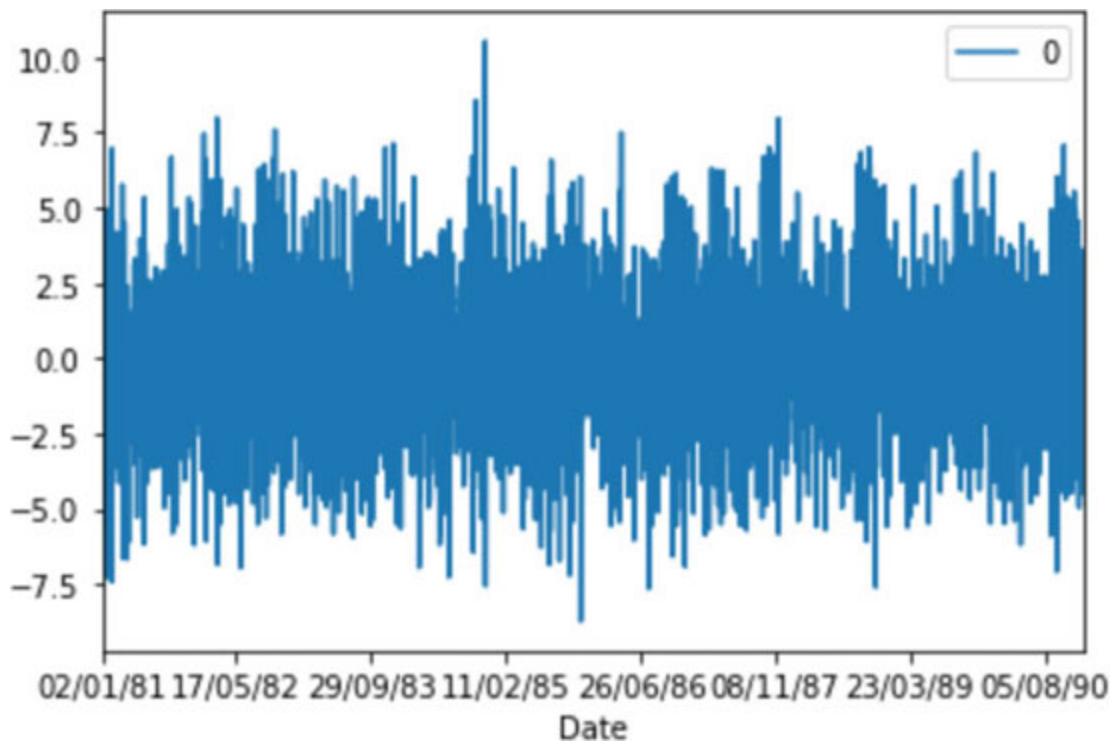
In the next step, we will look at calculating and plotting the residual errors in our model results.

9. In this step, we use the model defined in the previous step to plotting the residual errors.

**Residual errors:** The difference between the predicted values and the expected values. We can plot the residual error graph and observe the pattern of the series.

```
# plotting the residual errors
residualerrors = DataFrame(model_fit.resid)
residualerrors.plot()
plt.show()
```

The following figure is a visual representation of the residual error pattern:



*Figure 10.7: Residual error pattern*

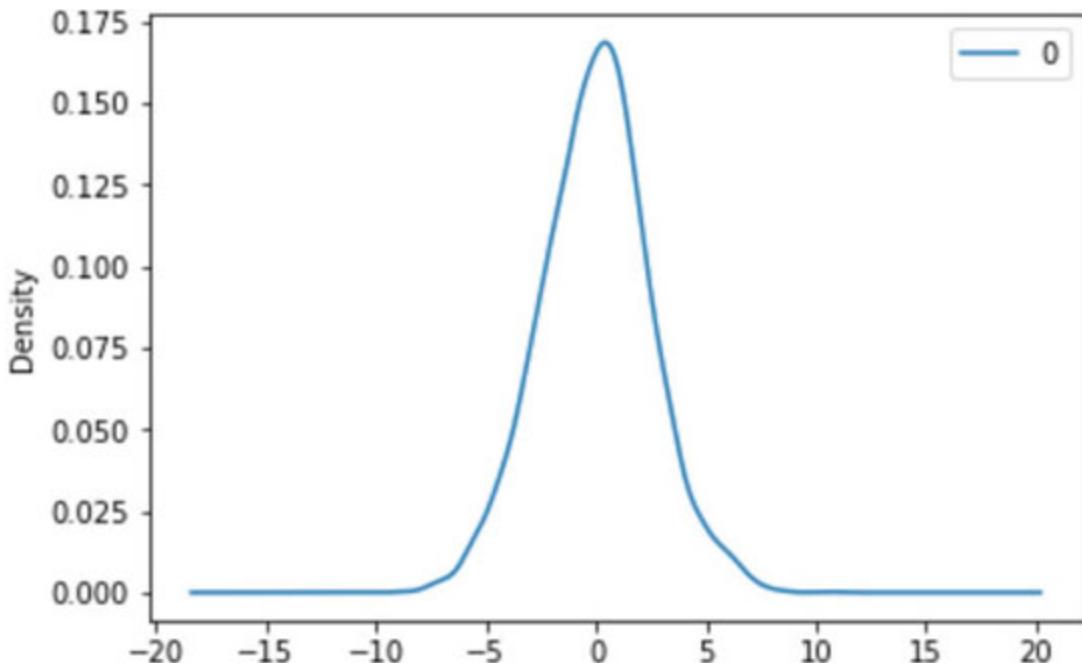
The next step shows us the distribution of residual errors.

10. Building on top of the above step, we try to describe the distribution of our residual errors.

We can see a bell curve as a distribution curve for our residual error plot. 'kde' is the kernel density estimation parameter:

```
residualerrors.plot(kind='kde')
plt.show()
print(residualerrors.describe())
```

The following figure is a visual representation of the residual error distribution:



```

0
count    3649.000000
mean     -0.001055
std      2.482981
min     -8.731153
25%     -1.637912
50%      0.063771
75%      1.562933
max     10.547800

```

*Figure 10.8: Residual errors distribution*

In the next step, we will use our current model to predict/forecast the temperature.

11. In this step, we are using our ARIMA model to predict temperatures by splitting the data into training and test datasets. The test dataset outputs are compared with the forecasting outputs by our model and we will calculate the Mean Squared Error in the next step to understand the deviation between the predicted values and the expected values:

```

x = df.values
size = int(len(x) * 0.66)

```

```

train_data, test_data = X[0:size], X[size:len(X)]
history = [x for x in train_data]
predictions = list()
for t in range(len(test_data)):
    model = ARIMA(history, order=(5,1,0))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast()
    y' = output[0]
    predictions.append(y')
    observation = test_data[t]
    history.append(observation)
    print('predicted=%f, expected=%f' % (y', observation))

```

12. We will print the mean squared error value in this step”

```

error = mean_squared_error(test, predictions)
print('Test MSE: %.3f' % error)

```

## Output

Test MSE: 5.707

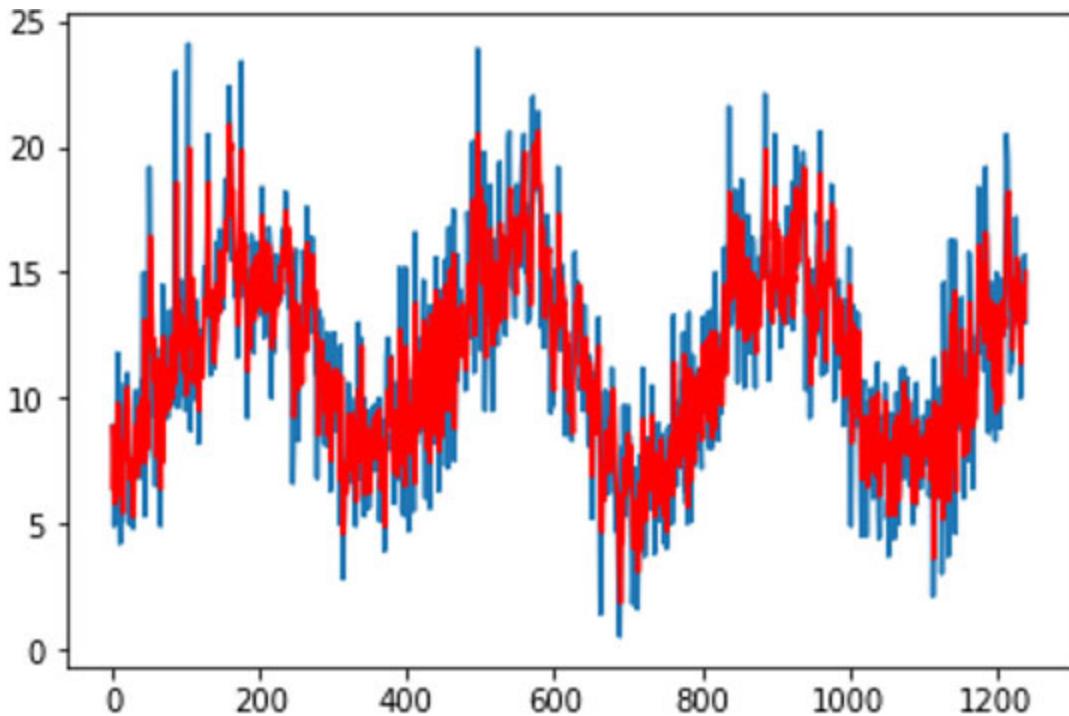
13. In this step, we will print the forecasting pattern and compare it to the original pattern produced by our data. You can notice that the forecasting pattern has the same trend and is very close in behavior with the original pattern:

```

plt.plot(test)
plt.plot(predictions, color='red')
plt.show()

```

The output is shown in the following figure, which is a visual representation of the pattern produced by the forecasted values:



*Figure 10.9: Pattern produced by the predicted values*

In the next section, we will take a look at the summary of our case study.

## Summary

In a time series forecasting problem, it is essential to visualize the predicted patterns and understand if it displays similar trends to your current data trends (in case your current data is already displaying some trends). It is not an easy task to validate your time series model since the prediction trends could vary based on your dataset and the trends in your dataset.

You can print the different metric scores for the above model and evaluate for yourself if the existing model it's a good fit for your data or not. And if not, then you may need to look at several characteristics like the amount of data, the data distribution intervals, and so on, to revamp your model.

## Conclusion

Time series are incredibly statistical and heavily rely on concepts like moving averages, stationarity, and many more. We need to understand the statistical basis underlying time series forecasting before jumping into the analysis head first.

Time series models are beneficial when it comes to analyzing the various trends. In this chapter, we have looked at the different types of patterns that one can observe in a time series data and models that can be applied to get insights out of these patterns. We have deep-dives into the working of the ARIMA model and implemented a case study for the same.

In the upcoming chapters, we will focus on the subject of deep learning. Deep learning models have changed the way artificial intelligence is being looked at in the industry and have created a pathway for developers to understand patterns more efficiently from the given dataset.

## **Quiz**

1. What is the time series forecasting? Can we forecast the type of users for a website based on a pattern?
2. What are the different components of a time series plot?
3. Are seasonal patterns or cyclical patterns easier to estimate in a time series?
4. What is autocorrelation?
5. Is stationarity a desirable property for time series modeling?

# CHAPTER 11

## Demystifying Neural Networks

### Introduction

Neural networks have been a subject of discussion for a very long time in the machine learning world. Artificial neural networks have significantly contributed to the deep learning way of working with data. Deep learning applications can infer patterns from a given set of data by creating highly efficient and generic machine learning models.

This chapter will help you understand what neural networks are and how they contribute to deep learning. We will look at neural networks from both a programmer's lens and a mathematician's lens. We will also look at the different ways to implement the neural network algorithms using the Python language.

### Structure

- What are neural networks and their importance
- Working of a neural network
- The mathematics associated with neural networks
- Types of activation functions
- Forward propagation and Backward propagation
- Shallow and deep neural networks
- A brief introduction to tensorflow
- Case Study in Python
- Conclusion

### Objectives

- Learn the end to end working of a neural network

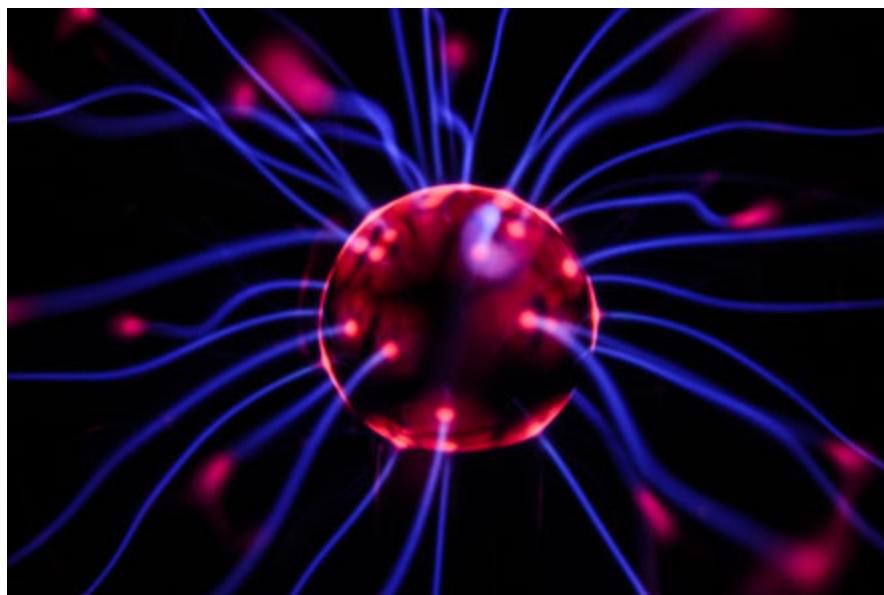
- Implement an end to end neural network from scratch in Python

## What are neural networks

The idea of neural networks is fairly ancient and has been a very familiar subject to the researchers. They have been working in the field of Computer Science as well as the intersection of Computer Science and Mathematics. Neural networks derive their name from the neurons in our brains and have been built to mimic the functioning of our brain.

The neural networks are similar to the various connections that are present between the neurons in our brain and carry the signals between several neurons. To understand neural networks in-depth and their purpose, let us take a look at the functionality of the neurons in our brain.

Here is an example image of a neuron and it's several connections:



*Figure 11.1: An example image of a neuron*

The human brain is one of the most complicated yet useful organs in the human body. Every action that we take, every decision that we make involves our brain performing some significant functions to aid the actions or the decision-making process. Neurons are nerve cells that make up the entire brain. There are an estimated 100 billion neurons in a single human brain.

Similar to how we use the logic gates and cables in a computer, the neurons in our brain can collect and transmit various electrochemical signals, thereby creating a message sending ability in our brain. The neural network design and functionality are aimed to duplicate the actions of a neuron so that computers can understand and function like a human brain.

## **Why neural networks/deep learning?**

As machine learning data is becoming increasingly complex to process and understand, we need more complex algorithms that can learn deeper and recognize the various patterns present in data.

Neural networks consist of a set of algorithms which are primarily modeled on the human brain and can be used to classify data or form clusters of data with similar patterns.

Deep learning is a sub-branch of machine learning which uses neural networks at the backend. It not only helps in machine learning but can also significantly contribute to areas like computer vision and natural language processing, hoping to move the artificial intelligence algorithms closer to passing the Turing Test.

There are several applications where deep learning is making its mark in the industry. Here are some examples of deep learning applications:

- Facial identification or image recognition
- Handwriting Recognition
- Autonomous driving
- Machine translation

Here is a figure that illustrates the several applications of deep learning:



*Figure 11.2: Use cases of deep learning*

Here are some more examples of real-time problems that are being solved by deep learning:

- Our emails that come to our inbox are filtered based on whether they are spam or not. The spam filter is a machine learning algorithm that uses its understanding of the words present in an email to filter it as spam or not spam. The algorithm also learns from the new classification (people rejecting more emails as spam or approving spam emails as not spam) and keeps re-learning. The backbone of this algorithm's functionality is neural networks.
- Several algorithms that categorize customer reviews on e-commerce websites use underlying neural networks for classification. Understanding customer sentiment from customer reviews requires sentiment analysis, and deep learning can play an important role in classifying if a customer is happy or unhappy with a particular product.

In the next section, we will take a look at how a generic neural network works.

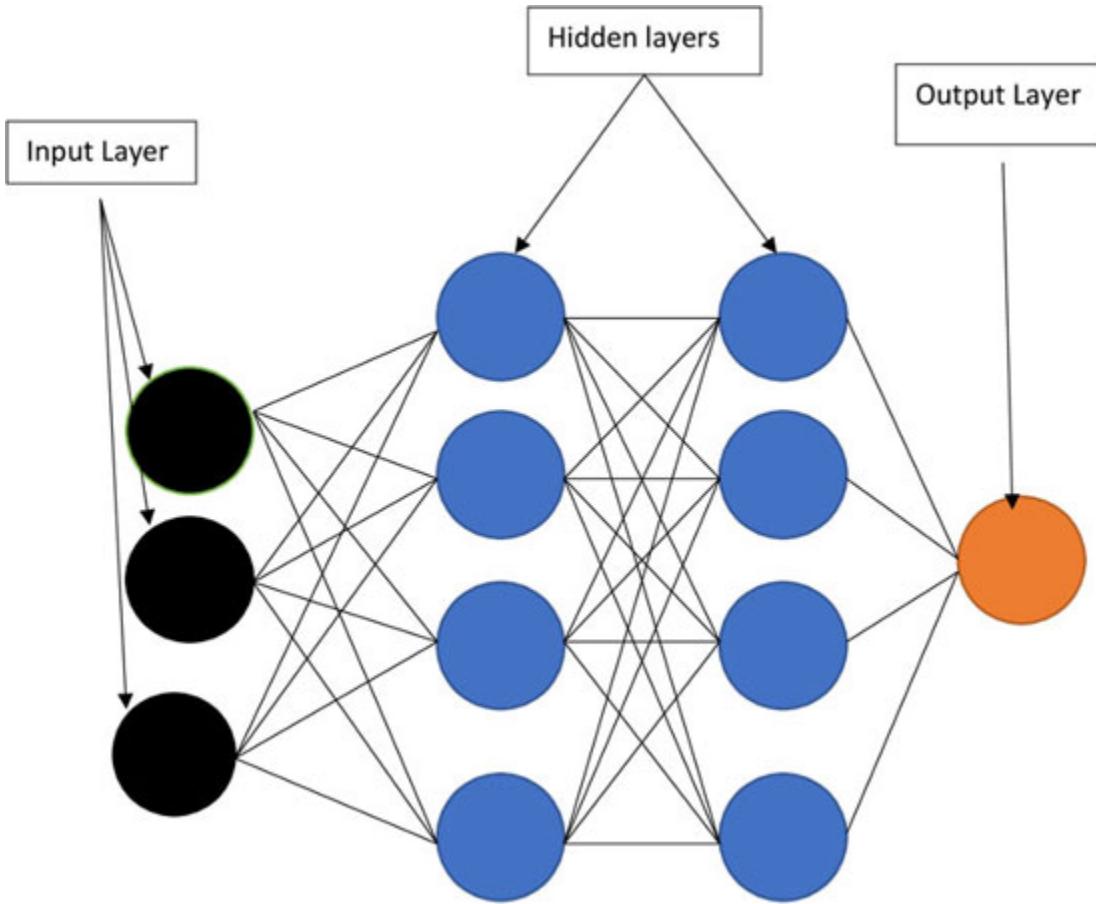
## **Working of the neural network**

A simple neural network consists of three layers:

- An input layer
- A hidden layer
- An output layer

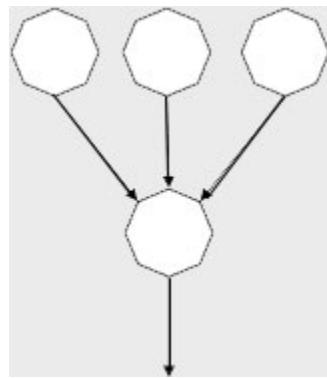
Each of these layers has a specific function to perform. The input layer receives the inputs to the algorithm, and the output layer is the one that gives the prediction/output after performing the required computation in the hidden layers. Each of these layers plays an important role in the final prediction, and we will look at the mathematics involved at each of these layers in the upcoming sections.

Here is an example image of how the various neural network layers look together:



**Figure 11.3:** An example of the various layers in neural networks

Here is another simple representation of neural networks:



**Figure 11.4:** An example of the simple neural network

To help you visualize the functionality of these layers, let's take a real-time example and look at how they can be modeled as neural networks. Our real-time example for this use case would be, *Predicting the House Prices in a certain locality*.

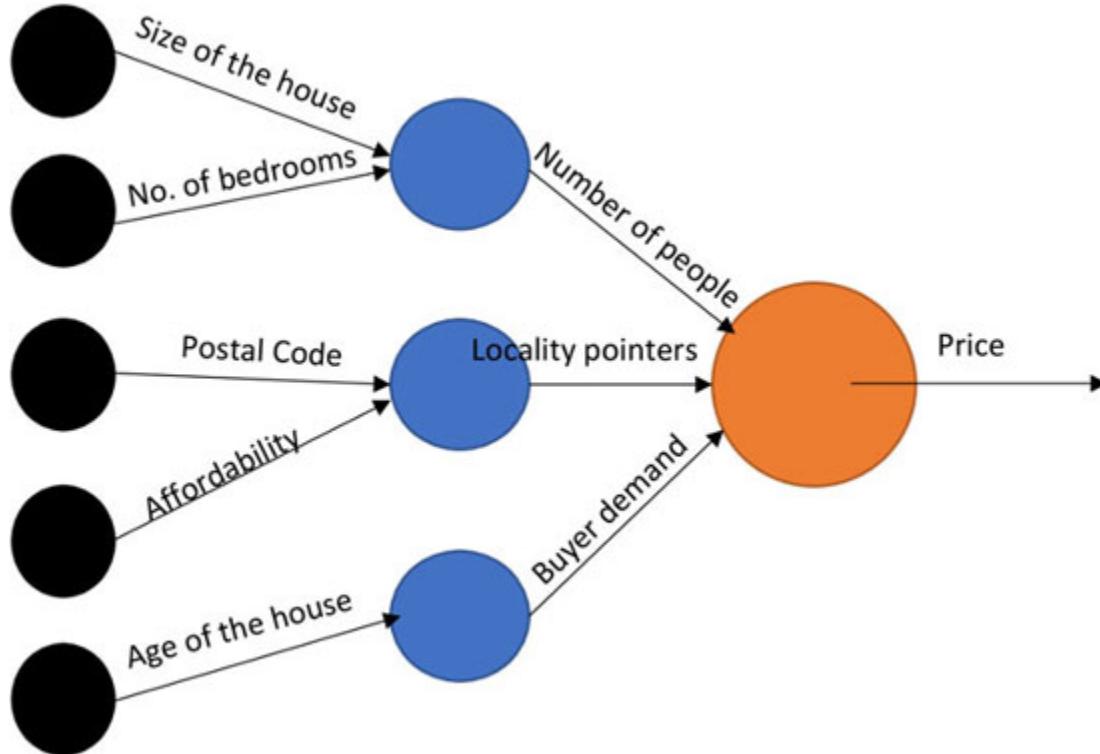
We will assume that the following characteristics/attributes are needed to be able to predict the house prices in a locality:

- Size of the house
- Number of bedrooms
- Postal code/Area code
- Affordability - how much can families afford in this area/community
- Age of the house

If we were to create a neural networks model using the above data, it would look like the following:

- Input layer - Given attributes
- Hidden layer - Attributes derived from the input attributes
- Output layer - Price

Let's model the above-given information in the form of a diagram to understand it better:



**Figure 11.5:** Modelling an example of a real-time neural network

In the above figure ([Figure 11.5](#)), you can see that we have used the given attributes as inputs, and the hidden layers have derived some more attributes/meaning out of the given attributes. The combination of it all would be used to predict the pricing at the output layer.

The figure ([Figure 11.5](#)) briefly explains the use case of hidden layers. However, the original functionality is slightly more complicated and based on mathematical computation, but this is a good picture for readers to understand the workings of the hidden layers.

The hidden layers use mathematical computation to derive relationships between the provided inputs and process these relationships to derive patterns that can be understood by the user. These layers re-train and re-learn the information propagating it in the network. In cases where there are several attributes or the relationships are more complex, we may have several hidden layers doing several rounds of computation and sending the final results to the output layer.

Now that we have had a look at the basic terminologies associated with a neural network let us deep-dive into the mathematical operations that are being performed at each layer in a neural network.

## The mathematics associated with neural networks

If we go back to the basics of machine learning, to chapters like *Linear Regression* and *Logistic Regression*, you would remember that the basic task of a machine learning algorithm is to optimize the coefficients/weights to create a model that can predict the new data points with as much accuracy as possible.

In the case of machine learning as well, we will be working with weights that will help assign importance/significance to each of the inputs that are coming to the neural networks. Let's start with some basic terminology associated with a neural network and then dive deep into the explanation for the terminologies.

Here are some basic neural network jargons (refer [Figure 11.5](#)):

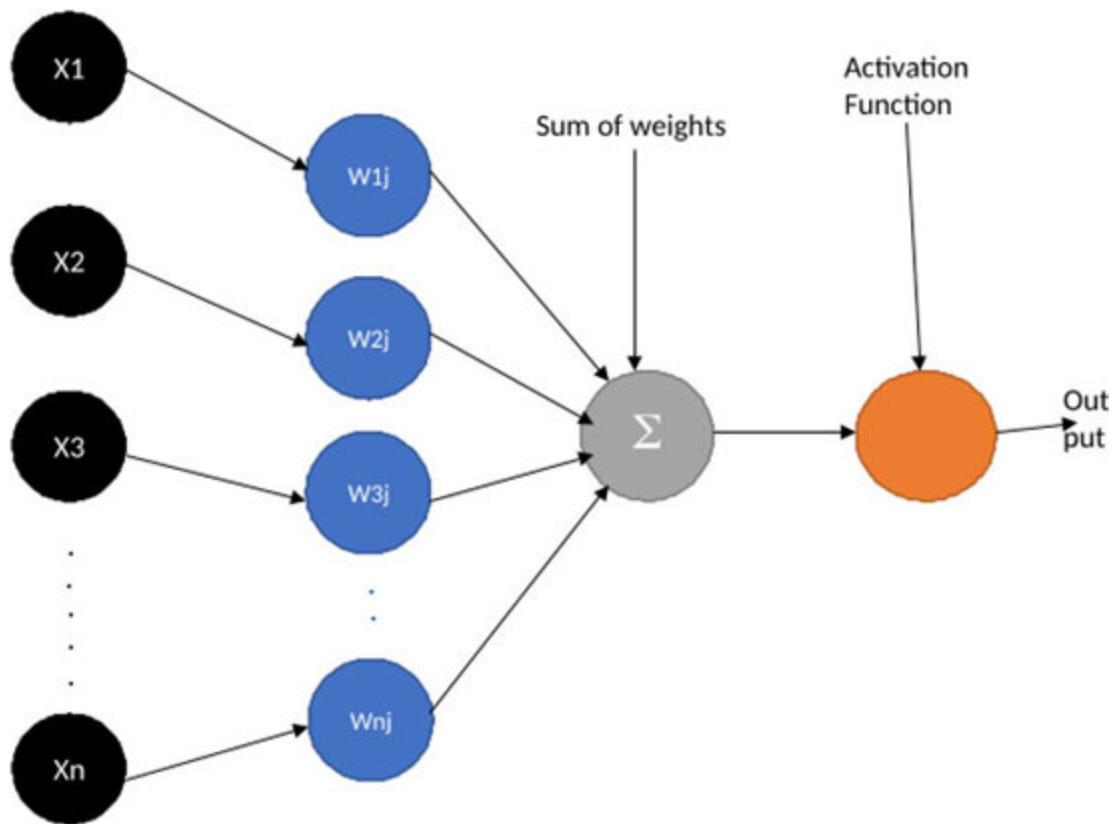
- **Inputs:** Inputs are attributes/characteristics present in our dataset (X<sub>1</sub>, X<sub>2</sub>, X<sub>3</sub>...X<sub>N</sub>)

- **Weights:** Coefficients that determine the importance of each feature ( $W_{1j}, W_{2j}, W_{3j} \dots W_{Nj}$ )

Transfer function/Activation function: A transfer function, also known as an activation function is a mathematical function applied to the weighted inputs to produce a final/net output.

In the Python code section of the book, we will understand more about how we can derive patterns from a dataset using the above neural network structure.

**NOTE: Transfer function and Activation function are used interchangeably in the machine learning world. To avoid confusion, we will use the term Activation function throughout the scope of this book.**



*Figure 11.6: Mathematical representation of functions in neural networks*

To understand more about the activation functions, let us look at the different activation functions and understand as to which ones can be used in which scenarios.

## Types of activation functions in neural networks

Activation functions are an important part of the neural networks. The base unit of a neural network is an artificial neuron that connects with other artificial neurons and creates a network mimicking the brain. Activation functions are mathematical functions applied to the sum of the products of the inputs and their corresponding inputs. They are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated (“fired”) or not, based on whether each neuron’s input is relevant for the model’s prediction.

It is the activation function that decides as to which values and their corresponding neuron parameters will go to the next layer. Each input has its weight and bias associated with it, whose values are updated throughout the neural network using back-propagation. Activation functions help us understand if a particular neuron/input should be activated based on its contribution to the output.

Here are several different types of activation functions.

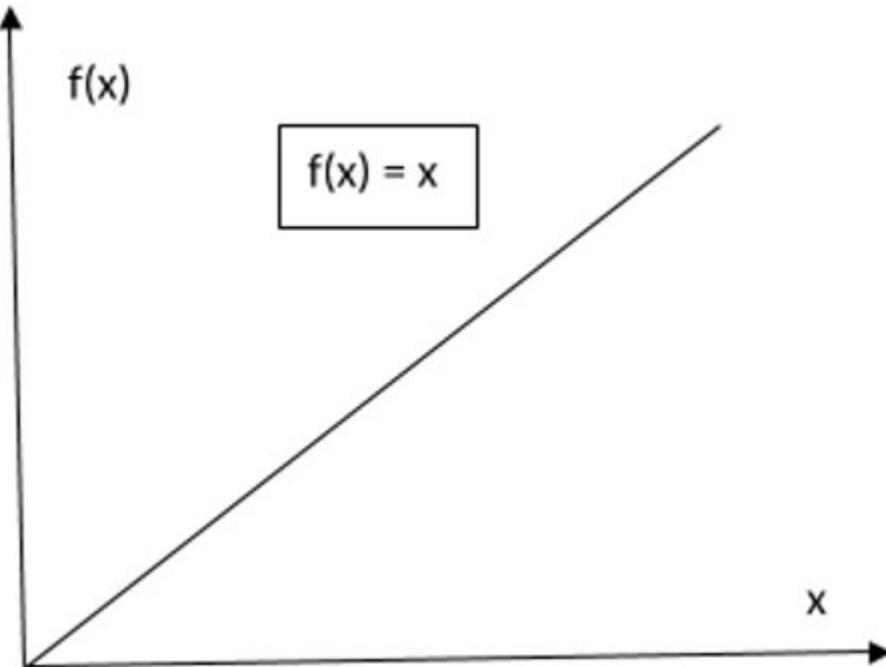
### Linear activation functions

Linear activation functions are functions where the variable and the function are linearly dependent. A linear activation function can be defined as follows:

$$f(x) = x$$

In this function, you can observe that the value of  $f(x)$  will increase/decrease linearly with the value of  $x$ . The input value  $x$  is nothing but the weighted sum of the weights and biases associated with the neurons. The output  $f(x)$  is not fixed in range and can take up any value based on the value of the input.

Here is a graphical representation of how a linear function would look like:



*Figure 11.7: Linear activation function*

Linear activation functions are very straightforward and do not have too many ways to activate the neurons since they have a constant derivative. As the derivative is constant, there is no concept of gradient descent happening in this function.

Our model is not involved in a constant learning process and improving the error rate, which is the purpose of the model. As the activation function is linear, there is no need for more than one hidden layer, thereby avoiding complex calculations when needed.

The linear activation function cannot be used for problems where complex mathematical operations are involved in deriving patterns or relationships out of the given inputs. In the next section, we will take a look at non-linear activation functions and the edge that they offer over linear activation functions.

### Non-linear activation functions

Non-linear activation functions add another layer of complexity to the linear activation functions as they are mathematically more advanced than the linear activation functions. In the upcoming sections, we will take a look at several commonly used non-linear activation functions.

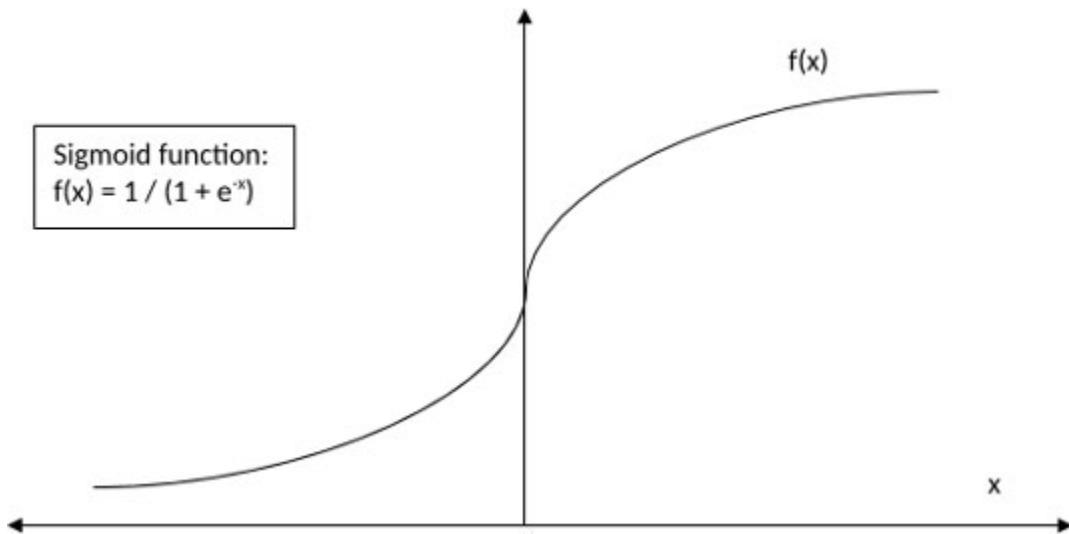
## Sigmoid/Logistic activation function

Sigmoid function or logistic function is one of the simplest non-linear activation functions. The equation for a sigmoid function is as follows:

$$f(x) = 1 / (1 + e^{-x})$$

As you can observe in the equation, whatever be the input, in this case, the output will always lie between 0 and 1. The graph, as seen in [Figure 11.8](#), is a curve and is differentiable at all points, thereby allowing a derivative to be obtained at all points in the graph. Assuming derivation and calculus part is explained in other parts of the book.

Here is a graph that represents the sigmoid curve:



*Figure 11.8: Sigmoid activation function*

The above graph shows us the sigmoid activation function curve. As you can see, the output ranges between 0 and 1 throughout the span of the input.

In a generic machine learning problem, neural networks that use the sigmoid function end up predicting outputs with lower accuracy during the training phase. As we also know that neural networks use hidden layers to compute the weights and biases, using the sigmoid activation function might produce values that are difficult to optimize.

Having said that, problem statements where the predictions are generally binary use the sigmoid activation function for achieving good accuracy.

## Tanh or Hyperbolic

As we saw in the above section, the range of output for the sigmoid function is 0 to 1, whereas the range of output for the tanh function is between -1 and 1. The graph of the function, as seen in [Figure 11.8](#), is a Hyperbolic tangent. As we all know, the tanh function is a combination of the sinh function and the cosh function; it is calculated as follows:

Here is how the tanh formula was calculated:

$$\sinh = (e^x - e^{-x})/2$$

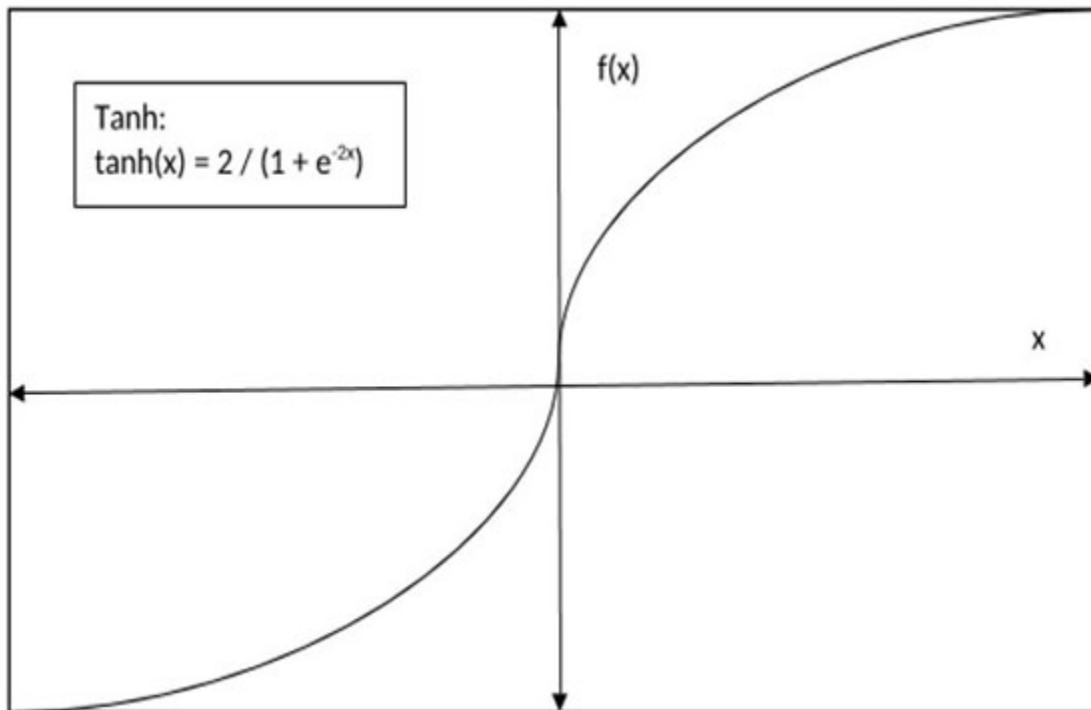
$$\cosh = (e^x + e^{-x})/2$$

$$\tanh = \sinh/\cosh$$

$$\tanh = 2 / (1 + e^{-2x})$$

Tanh and Sigmoid are quite similar except for the fact that tanh offers a wider range of outputs than the sigmoid function. The derivative of the *tanh* graph is slightly steeper than the sigmoid curve and, therefore, acts as the differentiating factor while choosing between the sigmoid and tanh activation function.

Here is a figure showing us the visual representation of the *tanh* function:



**Figure 11.9:** Tanh activation function

The above figure shows us that at 1 and -1, the edges of the graph are flatter in nature, thereby showing us the limits of the graph. In the next section, we will look at ReLU and Leaky ReLU activation functions, which are more frequently used in neural network problem statements.

## ReLU (Rectified Linear Unit)

**ReLU**, also known as **Rectified Linear Unit**, is one of the most widely used activation functions in deep learning models. As you can see, the function's formula from [Figure 11.9](#), the ReLU function value is positive if the input is positive, and for all negative inputs, the value of the function is 0.

The derivative for the ReLU function is 0 or 1, depending on the value of z. Here is the equation of ReLU and the derivative of ReLU for our reference:

$$R(z) = \max(0, z)$$

In other words:

$$R(z) = \{ 0 \} \text{ when } z \leq 0$$

$$R(z) = \{ z \} \text{ when } z > 0$$

Derivative of the above function will be:

$$d(R(z)) = \{ 0 \} \text{ when } z \leq 0$$

$$d(R(z)) = \{ 1 \} \text{ when } z > 0$$

We will now see why ReLU is more commonly used than the others and what problem does it solve.

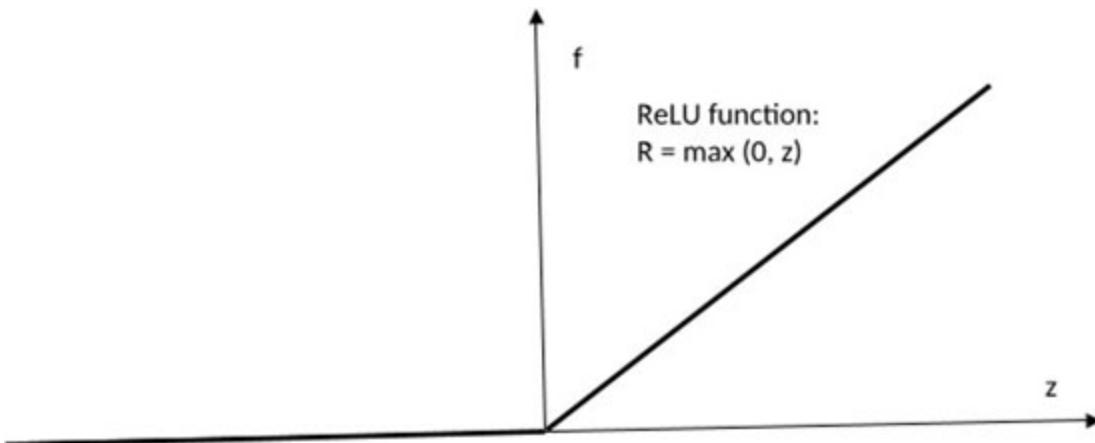
## What is the vanishing gradient problem, and how does ReLU solve this problem?

As the name suggests, vanishing gradient problem is a problem caused by some of the activation functions in neural networks where the gradients of the network's output become increasingly small in the early layers thereby making it extremely difficult to understand the contribution of a particular node(attribute) to the output of the network.

Several activation functions such as Sigmoid, Tanh, and so on, have a specific range to produce the output, for example, the range of output for Sigmoid is [0,1]. Because the range of output is too small, a huge change in the input would map to a small change in the output. As the number of layers increases, the output mapping becomes smaller and smaller, thereby reducing the effectiveness of the neural network.

Since ReLU functions have a derivative of either a 0 or a 1, there is no question of vanishing derivatives. The derivatives can be propagated throughout the network easily in the case of ReLU, therefore, making it more favorable for deep learning problems.

Here is a figure that describes the ReLU function graphically:



*Figure 11.10: ReLU activation function*

As you can notice, there is a 0 derivative in the ReLU function, which will result in the neurons not being able to learn anything since 0 is being passed in the network.

We can fix the above problem by using another algorithm called Leaky ReLU, which adds a leakage factor, thereby making the 0 derivative non-zero.

Several online resources cover Leaky ReLU in-depth, which might be useful for deep learning enthusiasts who want to dive deeper into the mathematics and statistics related to deep learning methodologies.

In the next section, we will look at two basic methods of propagating information in neural networks—forward propagation and backward propagation.

## Forward propagation and Back propagation

In this section, we will throw light on two types of propagation methods in neural networks—Forward propagation and Backward propagation. We will briefly introduce these methodologies so that you can understand the end to end functioning of neural networks.

### Forward propagation

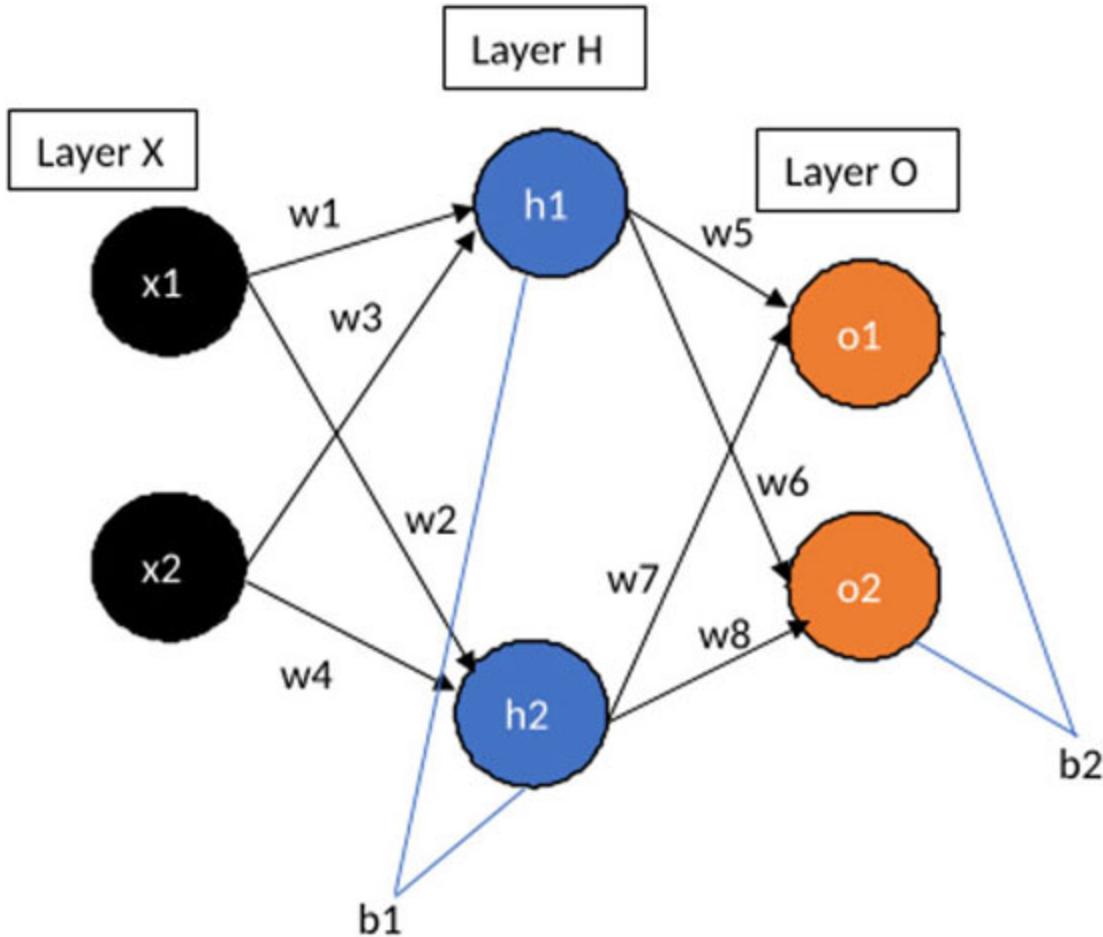
Forward propagation is the process of forwarding inputs from one layer to the next layer in a neural network until we reach the output layer. There are activation functions that act at each layer and forward the output to the next layer.

As we have already seen, in neural networks, there is an input layer, several hidden layers, and an output layer. We also calculate parameters width and depth in a network, where depth is the number of hidden layers, and width is the number of nodes on each hidden layer. The outputs produced at the layers are dependent on the activation function that is being used at that layer.

### Backward propagation

Backward propagation is a methodology where information in the neural network is sent backward, to calculate the error and update the weights accordingly.

Here is an image that we will use to explain back propagation in-depth:



*Figure 11.11: Backpropagation in neural networks*

Let's look at [Figure 11.11](#) and understand the different layers here: The first layer (**Layer X**) is the input layer, we have one hidden layer (**Layer H**), and one output layer (**Layer O**). Here is what happens in a backpropagation algorithm:

- Outputs from the input layer are sent to the hidden layer as inputs where an activation function is applied, and the outputs are sent to the output layer
- We will backpropagate the error calculated at the output layer back to the input layers to update the bias and the weights

In the upcoming section, we will take an example and walk through the example to understand the mechanism of backpropagation in depth.

In [Figure 11.11](#), our example figure,  $x_1$ , and  $x_2$ , are the inputs,  $h_1$ , and  $h_2$  are the nodes present in the hidden layer, and  $o_1$  and  $o_2$  are the output

nodes. The labels  $w1, w2 \dots w8$  are the weights associated with the inputs, and hidden layers, and  $b1$  and  $b2$  are the biases associated with the input layer and the hidden layer.

The output at  $h1$  is computed in the following manner:

$$h1 = x1 * w1 + x2 * w3 + b1$$

We will apply an activation function at  $h1$ , to get the output at  $h1$ . For easy understanding, we will pick the sigmoid activation function.

The equation for the sigmoid activation function is:

$$fx = 1/(1 + e^{-x})$$

Here, we will replace  $x$  by  $h1$ :

$$fh1 = 1/(1 + e^{-h1})$$

The function  $f$  calculated in the above step is the output at  $h1$ ; similarly, we will use the parameters  $x1, w2$ , and  $x2, w4$  for calculating  $h2$  and the output at  $h2$  using the sigmoid function. We will use these outputs generated as inputs to calculate the output at the output layer. The following equation is the output calculation at the output layer:

$$o1 = fh1 * w5 + fh2 * w7 + b2$$

Using the  $o1$  value, we will calculate the output at  $o1$ :

$$fo1 = 1/(1 + e^{-o1})$$

Now that we have our first set of outputs at  $o1$  and  $o2$ , we will calculate the deviation of the output from the expected target values, which will be the error produced using the current inputs and the activation function that we have chosen for the calculation.

Error-values will be calculated for both the outputs  $o1$  and  $o2$  as  $E_{total}$ .

After having received the error values, we will now start back-propagating the error to calculate the error at each of the weights using partial differentiation. Let's start with weight  $w5$ , which is the weight from  $h1$  to  $o1$ .

Here is an equation that will help us understand better:

$$\text{Error at } w5 = \partial E_{total} / \partial w5$$

We split the above function into several relevant terms:

$$\partial E_{total}/\partial w5 = \partial E_{total}/\partial o1 * \partial f_{o1}/\partial o1 * \partial o1/\partial w5$$

The above partial differentiation term will tell us the change in  $w5$  that needs to happen to reduce the error value. There is a learning factor that is added to the change to calculate the final  $w5$  value.

Similar to the process of updating  $w5$ , all the other weight parameters are updated as we backpropagate through the neural network. Once we reach the input layer, we will forward propagate and calculate the outputs.

As the weights keep getting updated, the error will keep getting reduced. When we reach a threshold error value, we can consider the model robust enough to be used for predicting the desired results for new and unknown upcoming inputs.

To summarize, backpropagation consists of the following steps:

1. Perform forward propagation using the value of weights and calculate the error at the output layer.
2. Perform partial differentiation of the error output values for the weights to update the weights.
3. Once you reach the input layer, use the updated weight values, and perform the action mentioned in Step 1 until the final error is less than the threshold value set by us.

## Shallow and deep neural networks

Neural networks are generally of two types—shallow neural networks and deep neural networks. Depending on the type of dataset and the number of attributes, you can figure out as to which neural network model will suit you the best. Here is some basic difference between the two kinds of neural networks:

Shallow neural networks	Deep neural networks
<ol style="list-style-type: none"> <li>1. Consists of a smaller number of hidden layers</li> <li>2. Requires a higher number of parameters to be present</li> <li>3. Is known to be able to fit any function</li> <li>4. Not computationally complex</li> </ol>	<ol style="list-style-type: none"> <li>1. Consists of a greater number of hidden layers</li> <li>2. Can manage with a smaller number of parameters</li> <li>3. Not all activation functions perform well with deep neural networks due to the vanishing gradient problem</li> </ol>

- |  |  |
|--|--|
|  | 4. Computationally more complex as it uses the neurons in the hidden layer to extract finer features from the given input features |
|--|--|

As the amount of data and the dataset attributes are evolving over some time, it has become essential to add computational complexity to neural networks.

In the next section, we will learn about a powerful library called tensorflow. We will learn about the basic commands that make up tensorflow to lay the foundations for tensorflow.

## A quick look at TensorFlow

In this section, we will take a look at what us TensorFlow and some basics of programming with TensorFlow to familiarize you with its functionality.

TensorFlow is a deep learning library released by Google to aid in machine learning and neural network research. It is a robust and versatile library built on the run on several CPUs and GPUs and has multiple wrapper languages like Python, C++, or Java.

A multi-dimensional array, also known as tensors, is sent as an input to the system, using which we can build a sort of flowchart of the operations, thereby attributing the name TensorFlow to the system. The smallest unit in TensorFlow is known as a tensor.

In this section, we will look at some programming basics of working with the tensorflow library:

1. We need to import the tensorflow library in our Jupyter notebook/Python code to consume the library. We can import it using the following command and use it as `tf` throughout our program:

Here is the import command for tensorflow:

```
import tensorflow as tf  
import numpy as np
```

While using the tensorflow library in our program, we need to start a session. A session executes the desired functions and stores the values of the intermediate variables and results.

2. The next step is to define a session for our TensorFlow operations. On successful creation of a session, we can print and let everyone know that the session has started:

```
sess = tf.Session()  
print ("Session has started")
```

### **Output:**

```
Session has started
```

In the next step, we will define a function to print the type and value of the various variables and constants defined in our program.

3. We will now define a function that will take an input argument and print the type and value of the input argument. In the following piece of code, we will input the string “HELLO WORLD” and notice the output.

The output gives us the type of the string as belonging to class tensor and the value as dtype string.

```
def print_tensorflow(x):  
    print("Type is\n %s" % (type(x)))  
    print("Value is\n %s" % (x))  
hello_world = tf.constant("HELLO WORLD")  
print_tensorflow(hello_world)
```

### **Output:**

```
Type is  
<class 'tensorflow.python.framework.ops.Tensor'>  
Value is  
Tensor("Const_8:0", shape=(), dtype=string)
```

4. In this step, we will define another function that will take an input argument but only print the value of the input argument. In the following piece of code, we can print the value of the variable as “HELLO WORLD” because we execute the `tf.constant` command using the session run function which makes sure that the variable is stored in the session memory and therefore, we can see the value while printing the variable in our current session:

```
def print_tensorflow_value(x):
    print("Value is\n %s" % (x))
    hello_out = sess.run(hello_world)
    print_tensorflow_value(hello_out)
```

### Output:

```
HELLO WORLD
```

5. In the following section, we will look at how Constants are being declared and used in TensorFlow. Here is the code snippet for constants:

In the first code snippet, we will just look at the data type that is being assigned to the constant.

```
c1 = tf.constant(4.5)
c2 = tf.constant(9.5)
print_tensorflow_value(c1)
print_tensorflow_value(c2)
```

### Output:

```
Value is
Tensor("Const_13:0", shape=(), dtype=float32)
Value is
Tensor("Const_14:0", shape=(), dtype=float32)
```

In the next code snippet, we will run them through `sess.run` and print the values assigned to them. We can see that the value of `c1_out` and `c2_out` have been printed as 4.5 and 9.5 respectively:

```
c1_out = sess.run(c1)
c2_out = sess.run(c2)
print_tensorflow_value(c1_out)
print_tensorflow_value(c2_out)
```

### Output:

```
Value is
4.5
Value is
9.5
```

6. In this section, we will look at the different operators and operations in TensorFlow. We will use the constants defined in the above section to perform operations on them and observe the results produced.

We will look at basic operations like sum and multiplication in TensorFlow. Here is a code snippet for the various operations in TensorFlow:

```
sum_two_numbers = tf.add(c1, c2)
print_tensorflow_value(sum_two_numbers)
```

**Output:**

```
Value is
Tensor("Add_4:0", shape=(), dtype=float32)

sum_two_numbers_op = sess.run(sum_two_numbers)
print_tensorflow_value(sum_two_numbers_op)
```

**Output:**

```
Value is
14.0
a_mul_b = tf.multiply(a, b)
a_mul_b_out = sess.run(a_mul_b)
print_tensorflow_value(a_mul_b_out)
```

**Output:**

```
Value is
3.75
```

7. In the next section, we will look at variables and variable initialization in TensorFlow. We will define a variable and try to print it using `sess.run`. We will notice some errors, and we will also understand how we can work on resolving those errors.

Here is the code snippet that deals with variables:

```
weight = tf.Variable(tf.random_normal([5, 2], stddev=0.1))
print_tensorflow_value(weight)
```

**Output:**

```
Value is
```

```
<tf.Variable 'Variable_2:0' shape=(5, 2)
dtype=float32_ref>

weight_out = sess.run(weight)
print_tensorflow_value(weight_out)

Output: ERROR

//Variables need to be initialized in tensorflow before
running them through the session

init = tf.initialize_all_variables()
sess.run(init)
print ("Initializing Variables")
```

### Output:

```
Initializing Variables

weight_out = sess.run(weight)
print_tensorflow_value(weight_out)
```

### Output:

```
Value is
[[ 0.02752653 -0.05753667]
 [-0.06099767  0.06995358]
 [-0.1159814  0.01317753]
 [ 0.15581843  0.09825547]
 [ 0.07575827  0.04244521]]
```

There are several other advanced concepts and tutorials on tensorflow that you can find online if you are interested to understand more and deep-dive into TensorFlow.

There are also other libraries like PyTorch and Keras that support deep learning and are commonly used in the developer ecosystem. We will not be diving deep into these libraries in this book, but we encourage readers to look them up and understand the basics of how they work.

The next section is one of the most important sections in the book, where we will be building an end-to-end hands-on neural network and use a dataset to predict the labels using our neural network model. There are

several terminologies that we will explain as we go along the step by step solution.

## Case study - I

This case study will take us through the various steps that are involved in building an artificial neural network model. In this section, we will be focusing on the MNIST Dataset.

We will go through the process of cleaning the data (if required), performing exploratory data analysis followed by model building and checking for accuracy of the built model. In the following case study, we will build a model to predict a handwritten digit based on its picture.

## About the data

The dataset that we are using for this case study is one of the most common datasets used in deep learning. The dataset is known as the MNIST dataset (Reference: <http://yann.lecun.com/exdb/mnist/>) and contains over 60,000 example images of handwritten digits for training and 10,000 example images for testing. All of the digits have been normalized and have been made fixed-sized images.

The attributes, in this case, are going to be the pixels of the image. Let's look at more information about the attributes in the upcoming section.

## Attribute information

In the case of an image, the image is converted into pixels which are assigned different mathematical values based on the color exhibited by the pixel. Each image is formed by several pixels, and we use the different pixel values to classify any new image that comes as an input to our model.

Since MNIST is a popular dataset, there is an in-built Python library called python-mnist that can be installed using the following command:

```
pip install python-mnist
```

Or

```
pip3 install python-mnist
```

We will import the MNIST dataset from the `mnist` library in python and use it in our code. We also need to download the MNIST dataset from the above-given reference link and store it in a folder titled MNIST on our system.

We will load the training dataset and convert it into a numpy array, which will be our dataset. This NumPy array will consist of the pixels of the images. The labels are the digits, each of which has a certain combination of pixels associated with it.

## Python code and step-by-step analysis

1. The first step is to import all the required machine learning libraries for this particular analysis:

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import numpy as np
import matplotlib.pyplot as plt
```

2. Read the data (which is stored in the ubyte format) using the MNIST library and store it as a variable. While reading the data, you must mention the location where the datasets would be stored.

In my case, I have mentioned the storage folder as MNIST.

Once you download the data, you will get two train gz files called `train-images-idx3-ubyte.gz` and `train-labels-idx3-ubyte.gz`. The test files are gz files called `t10k-images-idx3-ubyte.gz` and `t10k-labels-idx1-ubyte.gz`.

These training and test files will be unwrapped by the `input_data` function present in the `mnist` library as a part of the TensorFlow tutorials.

The original datasets can be downloaded from here (<http://yann.lecun.com/exdb/mnist/>). I have unpacked and stored the datasets under the `/MNIST` folder on my local system:

```
# Read the data from the tensorflow in-built mnist dataset
mnist = input_data.read_data_sets("MNIST/", one_hot=True)
```

**Output:**

```

Successfully downloaded train-images-idx3-ubyte.gz 9912422
bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz

Successfully downloaded train-labels-idx1-ubyte.gz 28881
bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz

Successfully downloaded t10k-images-idx3-ubyte.gz 1648877
bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz

Successfully downloaded t10k-labels-idx1-ubyte.gz 4542
bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz

```

3. In this step, we will define the various deep learning parameters like the learning rate, image size (the pixels), the number of labels, the number of steps and the batch size of the neural network:

```

#Define the various Deep Learning parameters as well as the
image and label size
#image_size = 28 refers to the pixel dimensions and
num_labels = 10 refers to the number of #digits (0-9)

image_size = 28
num_labels = 10
learning_rate = 0.05
number_of_steps = 1000
batch_size = 100

```

4. In this step, we will define a TensorFlow placeholder for the training dataset (`x_train`) and the labels (`y_train`). A placeholder is a basic TensorFlow structure where we can define a variable and assign the data to it later as it moves through different stages:

```

# Define placeholders
x_train = tf.placeholder(tf.float32, [None,
image_size*image_size])
y_train = tf.placeholder(tf.float32, [None, num_labels])

```

5. In this step, we will print the training set placeholder variable, to ensure that it has been assigned the value correctly:

```
print(x_train)
```

**Output:**

```
Tensor("Placeholder:0", shape=(?, 784), dtype=float32)
```

6. In this step, we will print the first few digits present in our dataset. As our image dimension is 28\*28 pixels, we will reshape the image into 28\*28 and plot it using the matplotlib library.

We initially use a 10\*10 figure size and then print all the digits by creating subplots from the existing training set:

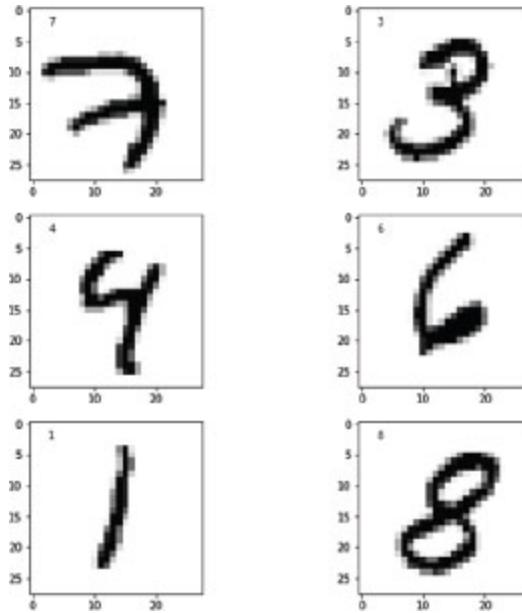
```
left= 2.5
top = 2.5

fig = plt.figure(figsize=(10,10))

for i in range(6):
    ax = fig.add_subplot(3,2,i+1)
    im = np.reshape(mnist.train.images[i,:], [28,28])

    label = np.argmax(mnist.train.labels[i,:])
    ax.imshow(im, cmap='Greys')
    ax.text(left, top, str(label))
```

Here is a series of output images produced by the above `ax.imshow()` command which shows the top six images of handwritten digits present in our dataset:



**Figure 11.12:** Images of digits present in the MNIST dataset

The above images are handwritten digits that are present in the training dataset along with their corresponding labels. In the next step, we will encode the labels to make it easier for processing the data.

7. We will now create the placeholders and variables for the input layer, the weights, the bias, and the outputs layer.

The input layer consists of the images, which are formed by 28\*28 pixels (784).

The weights are a matrix of zeros whose size is image pixels\* number of labels (784\*10).

The bias is a matrix of zeros of size number of labels (10).

```
# A placeholder for the data (inputs and outputs)
inputs = tf.placeholder(tf.float32, [None, 784])

# Weights: the weights for each pixel for each class
# bias: bias of each class
Weights = tf.Variable(tf.zeros([784, 10]))
bias = tf.Variable(tf.zeros([10]))
```

Once we create the input, weights, and bias placeholders and variables, we will use the above values to create the outputs layer by applying the softmax activation function on the above parameters.

**Softmax function:** The softmax function is an activation function that takes a series on numbers and converts it into probabilities, all of which together add up to 1.

Here is an equation that represents the softmax function:

$$\sigma(z)_i = e^{z_i} / \sum_{j=1}^K e^{z_j} \text{ for } i = 1, \dots, K \text{ and } z = (z_1, z_2, \dots, z_K) \in \mathbb{R}^K$$

Z: Input vector

K: Number of real numbers

```
# The model
outputs = tf.nn.softmax(tf.matmul(inputs, weights) + bias)
```

In the above step, we get the output layer results by applying the softmax function on the inputs, weights, and the given bias. The output layer information is stored in the outputs variable.

In the next step, we will create a placeholder to input the correct answers.

8. In this step, we are creating a TensorFlow placeholder where we can store the right answers.

We will use cross-entropy as a measure to understand the precision of the model that we have created.

Here is the formula for cross-entropy:

$$H(P, Q) = - \sum x \in X P(x) * \log(Q(x))$$

Where  $P$  and  $Q$  are the two events for which probability needs to be calculated.

```
# Create a placeholder to input the right answers
y_ = tf.placeholder(tf.float32, [None, 10])

# A measure of model precision using cross-entropy
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ *
tf.log(outputs), reduction_indices=[1]))
```

9. Once we obtain the cross-entropy from the previous step, in this step, we will try to minimize the cross-entropy by using the Gradient Descent Optimizer. There are several optimizers that we can use, but in this case, we chose Gradient Descent as it was the most suited for our problem.

We use a learning rate of 0.5 in this case to optimize the cross-entropy. The learning rate can vary in different cases, and you may have to try out various values of learning rates to arrive at the most optimized value for your algorithm.

```
# minimize the resulting cross_entropy using gradient
descent
# learning rate (also known as alpha) = 0.5
train_step =
tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

10. In the next step, we will execute our commands, which will call the various functions that we have defined above, to train our dataset. The training step is run for about 1000 times in our case:

```
init = tf.global_variables_initializer()

# the execution
sess = tf.Session()
sess.run(init)

# run training step 1000 times
for i in range(1000):

    # get random 100 data samples from the training set
    batch_xs, batch_ys = mnist.train.next_batch(100)

    # feed them to the model in place of the placeholders
    # defined above
    sess.run(train_step, feed_dict={inputs: batch_xs, y_:
        batch_ys})
```

11. In this step, we will print the accuracy of our model based on the predictions made by our neural network:

```
correct_prediction = tf.equal(tf.argmax(outputs,1),
tf.argmax(y_,1))

#this accuracy returns the mean value of an array of 1s and
0s.
```

```

accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))

# rerun the accuracy on the test set.
print("Accuracy: ", sess.run(accuracy, feed_dict={inputs:
mnist.test.images, y_: mnist.test.labels}))

Accuracy: 0.9182

```

In case you want to build and run your model without using TensorFlow, there is an alternative approach to do that as well. But when we are writing our training code, it might be prone to errors due to the huge sample size that we deal with in our dataset.

Therefore, we need to be careful while choosing to write our code from scratch or using the existing libraries.

In the next session, we will look at some small python functions that can be used for training and running a neural net.

## Alternative method – Programming from the scratch

In this section, we will define the various functions of the class NN, which can be useful to train and run the models.

We will write a function that will take the `input_vector` and the `target_vector` as arguments and then build the training arm of the neural network.

We have labeled the various steps under this function with a number associated with each of the steps, and we will explain briefly as to what's happening in each of these steps:

1. Our first step is to create the transpose of the input vector and the target vector to create new `input_vector` and `target_vector` variables. The `ndmin` parameters ensure that the minimum number of dimensions is 2 in this case.
2. Our second step is to calculate the output vector number 1, which is a dot product of the input weights and the input vector. We apply the

sigmoid activation function on this output vector to calculate the output at the hidden layer.

3. We will use the output calculated at the hidden layer as one of the inputs to the next output vector 2, which is a dot product of the output weights at the hidden layer multiplied by the output calculated at the hidden layer in the previous step. To this output vector 2, we will apply the sigmoid activation function and get the output of the network.
4. We will calculate the error at the output nodes by subtracting the calculated output at the final layer from the target vector values.
5. Using the process described in the *Backward propagation* section, we will now update the value of the actual weights to the new values.
6. *Step 6* and *Step 7* are similar to *Step 4* and *Step 5* where we re-calculate the errors and update the weights and the hidden layer and the input layers:

```
def train(self, input_vector, target_vector):  
    """  
        input_vector and target_vector can  
        be tuple, list or ndarray  
    """  
  
    #1  
    input_vector = np.array(input_vector, ndmin=2).T  
    target_vector = np.array(target_vector, ndmin=2).T  
  
    #2  
    output_vector1 = np.dot(self.wih, input_vector)  
    output_hidden = activation_function(output_vector1)  
  
    #3  
    output_vector2 = np.dot(self.who, output_hidden)  
    output_network = activation_function(output_vector2)  
  
    #4  
    output_errors = target_vector - output_network  
  
    #5  
    # update the weights:
```

```

tmp = output_errors * output_network \ * (1.0 -
output_network)
tmp = self.learning_rate * np.dot(tmp,
output_hidden.T)
self.who += tmp

#6
# calculate hidden errors:
hidden_errors = np.dot(self.who.T, output_errors)

#7
# update the weights:
tmp = hidden_errors * output_hidden * \ (1.0 -
output_hidden)
self.wih += self.learning_rate \ * np.dot(tmp,
input_vector.T)

```

In this section, we will define the function run, which will perform some functions on the input vector and return the output vector. We will use the weights and activation functions to arrive at the output layer in the following code snippet:

```

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray
    input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih, input_vector)
    output_vector = activation_function(output_vector)

    output_vector = np.dot(self.who, output_vector)
    output_vector = activation_function(output_vector)

    return output_vector

```

You can build a confusion matrix for the above code as well as play around with assigning the initial weights, which should be random.

## Case study summary

The accuracy of our model is ~92%, which is an extremely good percentage to achieve.

The above case study shows us how we can build a neural network and how the different elements of neural networks can be constructed individually in Python.

It is important to understand that **Exploratory Data Analysis** or **EDA** plays an important role in the machine learning process as it takes you through the various steps of cleaning and preprocessing the data before feeding it into a machine learning model.

You can print the different metric scores for the above model and evaluate for yourself if it's a good fit for your data or not. You can tweak anything from the number of hidden layers to the value of the learning rate to build the best model for your dataset.

## Conclusion

This chapter gives you an overall understanding of neural networks without delving too deep into the mathematics behind it but, at the same time, making sure that you understand the basic logic behind the different concepts in neural networks.

In this chapter, you would have had the opportunity to understand the various steps involved in constructing a neural network algorithm, the basic mathematics behind each of these steps, and an end-to-end example on how a neural network can be constructed in Python.

Moving forward, we will look at several types of neural networks like convolutional neural networks and recurrent neural networks and their applications in the industry in the upcoming chapters.

## Quiz

1. What is the difference between the weights and bias in machine learning? (Re-visit the old chapters to answer this question if necessary).
2. Give us an example of one application of a shallow neural network and one application of a deep neural network.
3. Which are the most frequently used non-linear activation functions used in neural networks?

4. What is the smallest unit of computation in TensorFlow?
5. What are the libraries used to implement the various activation functions in Python?

# CHAPTER 12

## Recurrent Neural Networks (RNN)

### Introduction

Recurrent Neural Network, also known as RNN are networks where the original structure is similar to neural networks but an additional state to store memory is added to them. Imagine a scenario where you are reading a book; your understanding of the book is based on the previous chapters that you have completed reading.

Similarly, many times in the machine learning world, we come across scenarios where we need to use this information from the previous iteration to apply to our new learnings, here is where recurrent neural networks can help.

### Structure

We will study the following topics in this chapter:

- What are feed-forward networks?
- What are RNNs?
- Applications of RNNs
- Types of RNN
- Step-by-step python code using Keras

### Objectives

- The primary objective of this chapter is to understand the working of a recurrent neural network
- Understand the different applications of RNN in the current world and learn to code RNN step-by-step in Python

## What are feed-forward networks?

A feed-forward neural network is an artificial neural network where the nodes do not connect to form a cycle. The network consists of several processing units like neurons, organized in several layers.

Imagine classifying an object in a feed-forward network. Here are the simple steps that will take place if you would want to classify a fruit:

- Design a feed-forward simple neural network
- Feed the picture of an apple at time  $t$
- The network classifies the picture as an apple and assigns the label Apple to it
- Feed the picture of a banana at time  $t+1$
- The network classifies the picture as a banana and assigns the label Banana to it

In the above-given scenario, the labelling at time  $t+1$  has nothing to do with the labelling at a time  $t$ , both of which are independent of each other.

Now imagine the scenario mentioned in the introduction, where you try to create an overall understanding of the book by parsing the contents of the book. Here is how a feed-forward neural network will treat the book:

- Feed the first chapter at time  $t$ .
- The network draws some conclusions from it
- Feed the second chapter at time  $t+1$
- The network now concludes individually from the second chapter, and so on

But the idea of creating a summary of the book is to be able to process the inputs provided at a time  $t$ ,  $t+1$ ,  $t+2$ , and so on, and create something based on all the inputs together.

It is why we cannot use a simple feed-forward network and need to think of strategies to improvise this process.

## What are Recurrent Neural Networks (RNNs)?

Recurrent Neural Networks are primarily used in speech recognition. Some of the issues which RNNs can address are:

- Can handle data which is sequential
- Considers both current and previous inputs
- Because of the memory state property, it can memorize previous inputs

As you can see in the below figure ([Figure 12.1](#)), recurrent neural networks have a memory state in the hidden layers due to which they can help connect the dots between current and previous inputs:



*Figure 12.1: Recurrent Neural Networks*

The above figure indicates the memory loop in the hidden layers. x and y are the inputs and the outputs of the machine learning model, respectively.

At a high level, a recurrent neural network (RNN) helps process sequences easily. Some of the examples could be stock prices on a day-to-day basis, sentences in a language etc. The term 'recurrent' signifies that the output at present later becomes the input to the upcoming step.

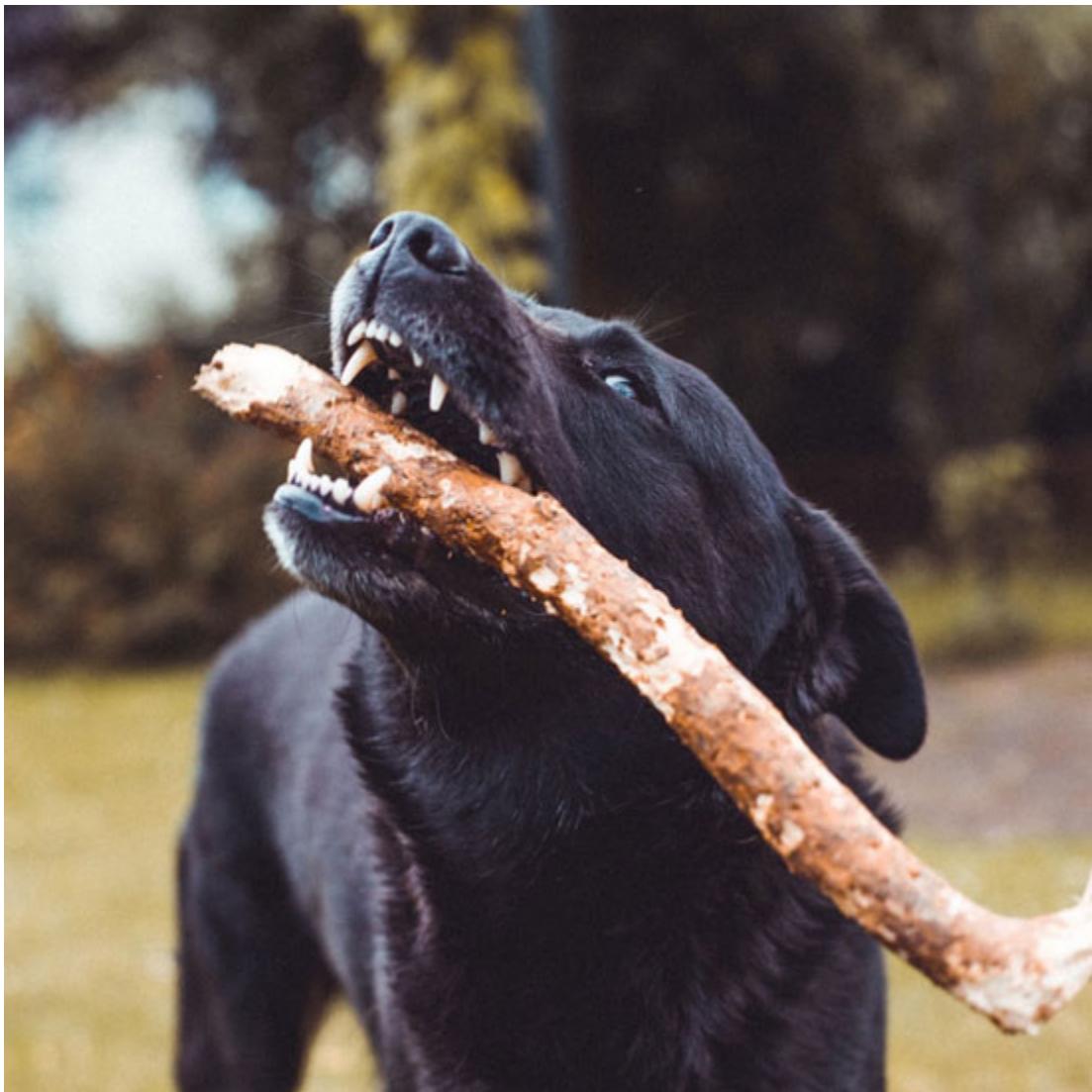
The RNN model has a unique power to remember. So, while processing it not only considers what it sees but also remembers the other elements to process the data efficiently.

## **Applications of Recurrent Neural Network**

Let us look at some of the applications of RNN in this section.

### **Application1 – Image Captioning**

One of the cool applications of the RNN is image captioning. Let us look at what that means. Look at the image ([Figure 12.2](#)) below and try to caption this image:



*Figure 12.2: Dog holding a stick*

The most common captions to the above image would be – 'A dog holding a stick' or 'A dog catching a stick'.

How did our brain arrive at this caption? Let us analyze the process a little bit:

- Our brain notices the image of the dog/the stick first and registers it.
- Post that, our brain registers the other entity in the picture. Let us assume that we noticed the dog first and then the stick.
- We notice that the dog is holding the stick.
- We connect both the objects and arrive at a caption.

Imagine the same steps that would need to be followed by a machine, where our model is helping it learn the different activities in the image and then correlate them by storing the previous information and helping it conclude on a caption. It is where RNN comes in as a strong player.

## Application 2 – Time series prediction

Time Series prediction is used in several industries to forecast future data based on current and past trends. Recurrent neural networks are quite capable of handling the sequence dependencies in time series. The Long-Short Term Memory or LSTM network is a type of recurrent neural network, frequently used in deep learning to train a huge corpus of data.

Using the basic principle of recurrent neural networks, the hidden layers can be programmed in such a way that they store the past data presented in a time series dataset to derive useful conclusions.

Let's look at an example where we have to predict the number of international passengers in units of 1000 given a year and a month.

To solve this problem:

- We will use a dataset that has collected information about international passengers in the past
- We will use the dataset and map trends that associate international passengers with the various months and years
- We will use these trends to predict the value given a year and a month

RNNs will play a significant role in the second step, where we are trying to derive the trends and patterns from the given data. Deep learning algorithms can especially be useful since the data to be analyzed would be huge:



*Figure 12.3: RNN in time series*

## Application 3 – Text mining and sentiment analysis

Analysis of text has always been on the radar for several organizations as well as the government in several regions as the content present on the internet is growing day by day.

Several text mining solutions fall under the RNN umbrella, where LSTM architecture is primarily used to create the model that performs the mining/analysis. An example could be – mining information from a movie dataset.

Here is an example of how we can extract the sentiment from the review for a given movie:

- Collect data regarding movies and their reviews
- Perform operations on the review text to clean it
- Create patterns to find out words that indicate the sentiment of the review
- Given a review for a movie, predict its a sentiment

LSTM architecture of RNNs can be really useful as it will try and connect the data stored in the memory to create the required patterns for prediction:

avans, Mummie, there wouldn't be any railway fare and we shan't have to pay for our meals. Oh, do let us go in a caravan."

Mrs. Russell shook her head. "I know it sounds lovely, darling; but how can we get a caravan? It would cost at least fifty pounds to buy one, even if we had one, Daddy couldn't get away this summer. No, we must make up our minds to do without a holiday this year; but I'll tell you what we'll do: we'll all go to Southend for the day, as we did last year, and have dinner and tea with us and have a splendid picnic."

"Then we can bathe again," said Bob; "but, oh! I do wish I could have a pony and ride," he added unexpectedly. "You don't know how I long for a pony," he continued, sighing deeply as he remembered the blissful holidays when a friend let him share his little Dartmoor pony and ride occasionally. "Southend is nothing but houses and people," cried Phyllis; "it's no better than this place; and oh! Mummie, I do so long for fields and flowers and animals," she added piteously; and she shook her long brown hair forward to hide the tears in her eyes.

"Never mind, darling, you shall have them one day," answered Mrs. Russell with easy vagueness.

This really was not very comforting, and it was the most fortunate thing that at that moment a car stopped at the door.

"Uncle Edward!" shouted Bob, rushing from the room. Phyllis brushed her tears so hastily from her eyes that she arrived at the front door almost as he did, and both flung themselves on the tall, kindly-looking man standing beside the car.

"Uncle Edward! Uncle Edward!" they cried. "You've come at last! We've been longing to see you. Oh, how glad we are you're here!"

Now the delightful thing was that their uncle seemed just as pleased to see them as they were to see him, and returned their hugs and greetings with most cordiality. They were just on the point of dragging him into the house, hanging one on each arm, when he said: "Stop, not so fast. There are things to fetch in from the car."

So saying he began diving into the back of it and bringing out, not only a suitcase, but various parcels, which he handed out one by one.

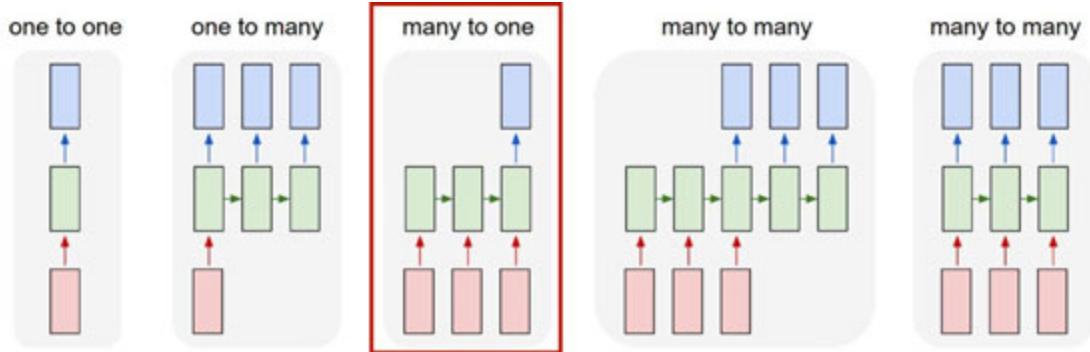
"That's the pair of chickens I've brought for your mother," said Uncle Edward.

Figure 12.4: Application of RNN in text mining and NLP

## Types of Recurrent Neural Network

In this section, we will cover the different types of recurrent neural networks. Each of these types is useful and can be used based on the inputs and outputs that are expected in your problem statement.

Here is an image which represents the different types of RNNs:



*Figure 12.5: Types of RNN Networks*

Let's understand a detailed description of each of these types.

## One to one

One to one single output RNN is also known as vanilla neural network and is used for basic machine learning problems. Each of these types of neural networks describes the shape of the input and the output and the classification associated with it.

One to one refers to a single input and a single output. Single output generally refers to the fact that the output can be scalar/binary – 0 or 1.

For example, single images/words are classified whether they belong to a category or not; that is, is this image a dog or not.

## Many to one

Many to one network take in multiple inputs which are in a sequence. But the final classification category is scalar or binary. It can be described as a sequence of images or words being classified in a single class.

For example, sentiment analysis where a review of a movie can be classified as negative or positive based on the sequence of words fed into the model in the form of a sentence.

## One to many

One too many networks take a single input and try to create an output with multiple classes. It can be described as a single input being tagged with multiple classes.

For example, captioning a picture will require one picture as an input, but the multiple outputs are the multiple words tagged on the picture.

## Many to many

Many to many take a sequence of inputs and gives us a sequence of outputs. In other words, multiple inputs are tagged into multiple classes.

For example, a review for a movie being classified into positive, negative or average.

## Python code and step-by-step RNN using Keras

The following code snippets give you an insight into the various steps of creating an RNN model using Keras. You can use the code and type it on your Jupyter Notebook/Google Colab or an editor on your local computer and match the outputs with the outputs shown here:

1. The first step is to import all the required machine learning libraries:

```
import collections  
import matplotlib.pyplot as plt  
import numpy as np  
  
import tensorflow as tf  
  
from tensorflow.keras import layers
```

2. The next step is to build a simple model using the above libraries. Here are the three built-in RNN layers in Keras:

- `tf.keras.layers.SimpleRNN` a fully-connected RNN where the output from the previous timestep is to be fed to next timestep.
- `tf.keras.layers.GRU`
- `tf.keras.layers.LSTM` for Long Short Term Memory.

In the following code snippet, there is a Sequential simple model that processes several sequences of integers into a 64-dimensional vector. Post that, it processes the resultant vector using an LSTM layer, which is one of the built-in layers mentioned above:

```

modelRNN = tf.keras.Sequential()
# Add an Embedding layer expecting input vocab of size
1000, and
# output embedding dimension of size 64.
modelRNN.add(layers.Embedding(input_dim=1000,
output_dim=64))
# Add a LSTM layer with 128 internal units.
modelRNN.add(layers.LSTM(128))

# Add a Dense layer with 10 units.
modelRNN.add(layers.Dense(10))

modelRNN.summary()

```

The model summary can be viewed in the output displayed here.

### **Output:**

Here is the output displayed in two parts:

Model: "model"			
Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[None, None]	0	
input_2 (InputLayer)	[None, None]	0	
embedding_2 (Embedding)	(None, None, 64)	64000	input_1[0][0]
embedding_3 (Embedding)	(None, None, 64)	128000	input_2[0][0]
encoder (LSTM)	[None, 64], (None, 33024	33024	embedding_2[0][0]
decoder (LSTM)	(None, 64)	33024	embedding_3[0][0] encoder[0][1] encoder[0][2]

*Figure 12.6: Summary of an RNN model part - 1*

dense_2 (Dense)	(None, 10)	650	decoder[0][0]
<hr/>			
Total params: 258,698			
Trainable params: 258,698			
Non-trainable params: 0			
<hr/>			

*Figure 12.7: Summary of an RNN model part - 2*

In the next step, we will take a closer look at outputs and the different states in RNN.

3. The default output of an RNN layer consists of one vector per sample. This vector is the RNN cell o/p of the last timestep, which in turn contains information about the whole input sequence.

The shape of this is (batch\_size, units) where units refer to the ‘number of units’ argument passed to the layer’s constructor. A layer can also return a sequence of outputs if we set return\_sequences = True.

```
modelRNN = tf.keras.Sequential()
modelRNN.add(layers.Embedding(input_dim=1000,
                               output_dim=64))

# The output of GRU will be a 3D tensor of shape
# (batch_size, timesteps, 256)
modelRNN.add(layers.GRU(256, return_sequences=True))

# The output of SimpleRNN will be a 2D tensor of shape
# (batch_size, 128)
modelRNN.add(layers.SimpleRNN(128))
modelRNN.add(layers.Dense(10))

modelRNN.summary()
```

The model summary resulting from the above code snippet can be viewed in the output displayed below.

### Output:

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
<hr/>		
bidirectional (Bidirectional (None, 5, 128))		38400
bidirectional_1 (Bidirection (None, 64))		41216
dense_3 (Dense)	(None, 10)	650
<hr/>		
Total params:	80,266	
Trainable params:	80,266	
Non-trainable params:	0	

---

*Figure 12.8: Summary of an RNN sequential\_2 model*

In the next step, let's take a look at how the RNN returns its states.

4. There could be an internal state(s) present in the RNN which store data from the processing. These final internal state(s) can be returned in an RNN layer. These returned states can be used to resume the RNN execution layer or to initialize another RNN.

Here are the steps that need to be followed to configure an RNN layer to return its internal state(s):

- Set `return_state = True` during layer creation.
- To configure the layer's initial state, use an additional keyword argument `initial_state` while calling the layer.

Here is an example code snippet to show you the above process:

```
encoder_vocab = 1000
decoder_vocab = 2000
encoder_input = layers.Input(shape=(None, ))
encoder_embedded =
    layers.Embedding(input_dim=encoder_vocab, output_dim=64)
    (encoder_input)
# Return states in addition to output
output, state_h, state_c = layers.LSTM(64,
    return_state=True, name='encoder')(encoder_embedded)
encoder_state = [state_h, state_c]
decoder_input = layers.Input(shape=(None, ))
decoder_embedded =
    layers.Embedding(input_dim=decoder_vocab, output_dim=64)
    (decoder_input)
# Pass the 2 states to a new LSTM layer, as initial state
decoder_output = layers.LSTM(64, name='decoder')
    (decoder_embedded, initial_state=encoder_state)
output = layers.Dense(10)(decoder_output)

modelRNN = tf.keras.Model([encoder_input, decoder_input],
    output)
modelRNN.summary()
```

The model summary from the above code snippet can be seen in the output display below:

### Output:

Here is the summary output in two parts:

```
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[ (None, None) ]	0	
input_2 (InputLayer)	[ (None, None) ]	0	
embedding_2 (Embedding)	(None, None, 64)	64000	input_1[0][0]
embedding_3 (Embedding)	(None, None, 64)	128000	input_2[0][0]
encoder (LSTM)	[ (None, 64), (None,	33024	embedding_2[0][0]
decoder (LSTM)	(None, 64)	33024	embedding_3[0][0] encoder[0][1] encoder[0][2]

**Figure 12.9:** Details about an RNN model part-1

```
dense_2 (Dense)           (None, 10)      650      decoder[0][0]
=====
Total params: 258,698
Trainable params: 258,698
Non-trainable params: 0
```

**Figure 12.10:** Details about an RNN model part-2

In the next section, we will talk about RNN layers and cells.

**5. RNN layers and RNN cells:** The RNN API provides cell-level APIs, which can be used to process a single timestep.

Each RNN layer has cells inside its for loop and wrapping a cell inside the `tf.keras.layers.RNN` layer gives you a layer that can process a batch of sequences.

Here are three built-in RNN cells, each of them corresponding to a matching RNN layer:

- `tf.keras.layers.SimpleRNNCell` corresponds to the `tf.keras.layers.SimpleRNN` layer.

- `tf.keras.layers.GRUCell` corresponds to the `tf.keras.layers.GRU` layer.
- `tf.keras.layers.LSTMCell` corresponds to the `tf.keras.layers.LSTM` layer.

Using cell abstraction, we can easily use RNN to build our custom architecture.

Whenever we have long sequences, it is typical behaviour to break them into smaller sequences and to feed these smaller values sequentially to the RNN layer, while maintaining the state of the layer.

It can be done by setting `stateful=True` in the constructor.

Here is a code snippet that shows us how to process the states and it also shows how to reset the state in the last line of the code:

```
paragraph1 = np.random.random((20, 10,
50)).astype(np.float32)
paragraph2 = np.random.random((20, 10,
50)).astype(np.float32)
paragraph3 = np.random.random((20, 10,
50)).astype(np.float32)

lstm_layer = layers.LSTM(64, stateful=True)
output = lstm_layer(paragraph1)
output = lstm_layer(paragraph2)
output = lstm_layer(paragraph3)

# reset_states() will reset the cached state to the
original initial_state.
# If no initial_state was provided, zero-states will be
used by default.
lstm_layer.reset_states()
```

In the next section, we will talk about reusing the RNN states.

**6. RNN State Reuse:** To reuse the state from an RNN layer, we can get the value of the states from `layer.states` and use it; however, you would like to, in your program. The `layer.weights()` variable does not contain the older states of the RNN layer.

The states from the `layer.states` can also be used as an initial state for another layer.

A sequential model cannot be used in the above case because it's a one-to-one model and supports layers with a single input and single output:

```
paragraph1 = np.random.random((20, 10,
50)).astype(np.float32)
paragraph2 = np.random.random((20, 10,
50)).astype(np.float32)
paragraph3 = np.random.random((20, 10,
50)).astype(np.float32)
lstm_layer = layers.LSTM(64, stateful=True)
output = lstm_layer(paragraph1)
output = lstm_layer(paragraph2)
existing_state = lstm_layer.states
new_lstm_layer = layers.LSTM(64)
new_output = new_lstm_layer(paragraph3,
initial_state=existing_state)
```

In the next step, we will look at using the CuDNN kernels in TensorFlow 2.0.

## 7. Using CuDNN kernels when available

Let's build a simple LSTM model to demonstrate the performance difference. We'll use as input sequences the sequence of rows of MNIST digits (treating each row of pixels as a timestep), and we'll predict the digit's label.

```
batch_size = 64
# Each MNIST image batch is a tensor of shape (batch_size,
28, 28).
# Each input sequence will be of size (28, 28) (height is
treated like time).
input_dim = 28

units = 64
output_size = 10 # labels are from 0 to 9

# Build the RNN model
```

```

def build_model(allow_cudnn_kernel=True):
    # CuDNN is only available at the layer level, and not at
    # the cell level.
    # This means 'LSTM(units)' will use the CuDNN kernel,
    # while RNN(LSTMCell(units)) will run on non-CuDNN
    # kernel.
    if allow_cudnn_kernel:
        # The LSTM layer with default options uses CuDNN.
        lstm_layer = tf.keras.layers.LSTM(units, input_shape=
            (None, input_dim))
    else:
        # Wrapping a LSTMCell in a RNN layer will not use CuDNN.
        lstm_layer = tf.keras.layers.RNN(
            tf.keras.layers.LSTMCell(units),
            input_shape=(None, input_dim))
    model = tf.keras.models.Sequential([
        lstm_layer,
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(output_size)])
)
return model

```

We will use the MNIST dataset for this:

```

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
sample, sample_label = x_train[0], y_train[0]

```

Here is the code snippet to call the above build\_model function:

```

model = build_model(allow_cudnn_kernel=True)

model.compile(loss=tf.keras.losses.SparseCategoricalCross
               entropy(from_logits=True),
               optimizer='sgd',
               metrics=['accuracy'])
model.fit(x_train, y_train,
          validation_data=(x_test, y_test),

```

```
batch_size=batch_size,  
epochs=5)
```

Here is the output of the model after training for several epochs:

### Output:

```
Train on 60000 samples, validate on 10000 samples  
Epoch 1/5  
60000/60000 [=====] - 28s  
475us/sample - loss: 1.0930 - acc: 0.6427 - val_loss:  
0.6855 - val_acc: 0.7798  
Epoch 2/5  
60000/60000 [=====] - 28s  
465us/sample - loss: 0.5405 - acc: 0.8307 - val_loss:  
0.4112 - val_acc: 0.8727  
Epoch 3/5  
60000/60000 [=====] - 28s  
469us/sample - loss: 0.3223 - acc: 0.9015 - val_loss:  
0.2804 - val_acc: 0.9133  
Epoch 4/5  
60000/60000 [=====] - 27s  
457us/sample - loss: 0.2412 - acc: 0.9261 - val_loss:  
0.1877 - val_acc: 0.9415  
Epoch 5/5  
60000/60000 [=====] - 28s  
464us/sample - loss: 0.2017 - acc: 0.9388 - val_loss:  
0.2115 - val_acc: 0.9332  
<tensorflow.python.keras.callbacks.History at 0x13728e410>
```

Now, let us take a look at a slower model, the code snippet of which is presented below:

```
slow_model = build_model(allow_cudnn_kernel=False)  
slow_model.set_weights(model.get_weights())  
slow_model.compile(loss=tf.keras.losses.SparseCategoricalCr  
ossentropy(from_logits=True),  
                    optimizer='sgd',  
                    metrics=['accuracy'])  
slow_model.fit(x_train, y_train,
```

```
validation_data=(x_test, y_test),  
batch_size=batch_size,  
epoches=1) # We only train for one epoch because  
it's slower.
```

Here is the output of the slower model:

### Output:

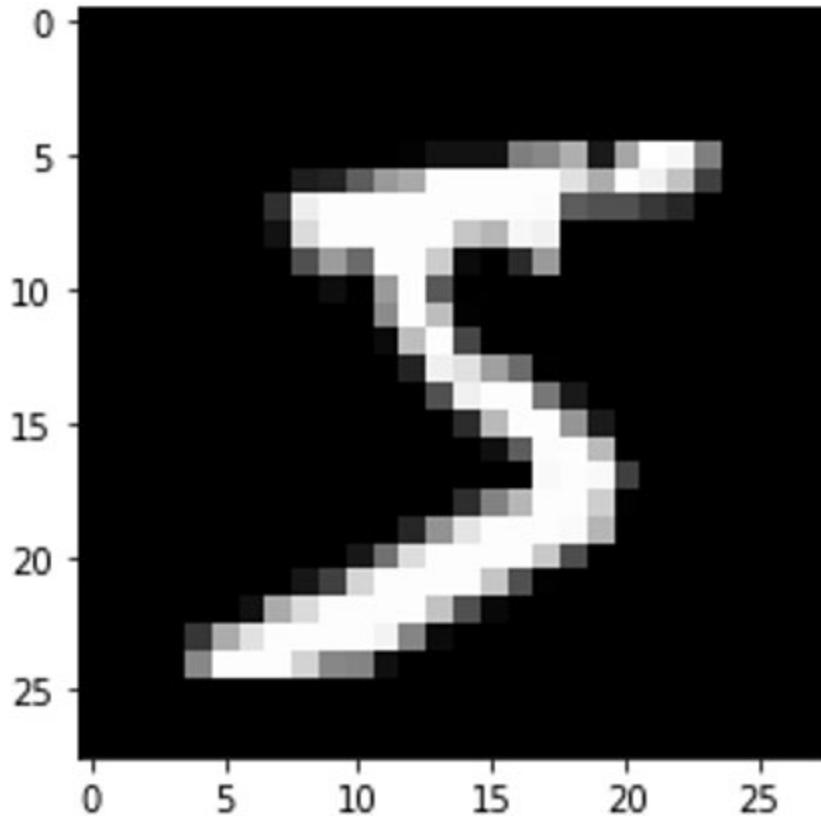
```
Train on 60000 samples, validate on 10000 samples  
60000/60000 [=====] - 28s  
461us/sample - loss: 0.1763 - acc: 0.9459 - val_loss:  
0.2206 - val_acc: 0.9285  
<tensorflow.python.keras.callbacks.History at 0x139665f90>
```

In case you would like to run the CuDNN-enabled model in a CPU driven environment, that is also possible by using Tensorflow 2.0.

The code snippet for the same is here:

```
with tf.device('CPU:0'):  
    cpu_model = build_model(allow_cudnn_kernel=True)  
    cpu_model.set_weights(model.get_weights())  
    result =  
        tf.argmax(cpu_model.predict_on_batch(tf.expand_dims(sample, 0)), axis=1)  
    #print('Predicted result is: %s, target result is: %s' %  
    #      (result.numpy(), sample_label))  
    plt.imshow(sample, cmap=plt.get_cmap('gray'))
```

The output image for the above code snippet can be found below:



*Figure 12.11: Output image from MNIST*

In the next section, we will look at how RNNs deal with nested inputs.  
8. If you want to send more information within a single timestep, then nested structures are the way to go. Here is an example code that helps you accept nested structures in an RNN:

```
class NestedCell(tf.keras.layers.Layer):  
  
    def __init__(self, unit_1, unit_2, unit_3, **kwargs):  
        self.unit_1 = unit_1  
        self.unit_2 = unit_2  
        self.unit_3 = unit_3  
        self.state_size = [tf.TensorShape([unit_1]),  
                          tf.TensorShape([unit_2, unit_3])]  
        self.output_size = [tf.TensorShape([unit_1]),  
                           tf.TensorShape([unit_2, unit_3])]  
        super(NestedCell, self).__init__(**kwargs)  
  
    def build(self, input_shapes):
```

```

# expect input_shape to contain 2 items, [(batch, i1),
(batch, i2, i3)]
i1 = input_shapes[0][1]
i2 = input_shapes[1][1]
i3 = input_shapes[1][2]
self.kernel_1 = self.add_weight(
    shape=(i1, self.unit_1), initializer='uniform',
    name='kernel_1')
self.kernel_2_3 = self.add_weight(
    shape=(i2, i3, self.unit_2, self.unit_3),
    initializer='uniform',
    name='kernel_2_3')

def call(self, inputs, states):
    # inputs should be in [(batch, input_1), (batch, input_2,
    input_3)]
    # state should be in shape [(batch, unit_1), (batch,
    unit_2, unit_3)]
    input_1, input_2 = tf.nest.flatten(inputs)
    s1, s2 = states

    output_1 = tf.matmul(input_1, self.kernel_1)
    output_2_3 = tf.einsum('bij,ijkl->bkl', input_2,
    self.kernel_2_3)
    state_1 = s1 + output_1
    state_2_3 = s2 + output_2_3

    output = (output_1, output_2_3)
    new_states = (state_1, state_2_3)

    return output, new_states

def get_config(self):
    return {'unit_1':self.unit_1, 'unit_2':unit_2,
    'unit_3':self.unit_3}

```

Let us now call the above declared functions by creating our own nested structures:

```

unit_1 = 10
unit_2 = 20

```

```

unit_3 = 30

i1 = 32
i2 = 64
i3 = 32
batch_size = 64
num_batches = 100
timestep = 50

cell = NestedCell(unit_1, unit_2, unit_3)
rnn = tf.keras.layers.RNN(cell)

input_1 = tf.keras.Input((None, i1))
input_2 = tf.keras.Input((None, i2, i3))

outputs = rnn((input_1, input_2))

modelRNN = tf.keras.models.Model([input_1, input_2],
outputs)

modelRNN.compile(optimizer='adam', loss='mse', metrics=
['accuracy'])

```

Here is the code snippet to fit the model:

```

input_1_data = np.random.random((batch_size * num_batches,
timestep, i1))
input_2_data = np.random.random((batch_size * num_batches,
timestep, i2, i3))
target_1_data = np.random.random((batch_size * num_batches,
unit_1))
target_2_data = np.random.random((batch_size * num_batches,
unit_2, unit_3))
input_data = [input_1_data, input_2_data]
target_data = [target_1_data, target_2_data]

modelRNN.fit(input_data, target_data,
batch_size=batch_size)

```

You can easily use the above RNN techniques to build your own RNN model for problem-solving.

In the next section, let us go through one of the applications of RNN using Keras.

**The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.**

## Application of RNN in real-time example

In this section, we will look at the application of RNN in an airline passengers problem.

We will use the dataset (`airline-passengers.csv`) and apply RNN on this dataset to predict the number of passengers given a month and a year.

The following code sample is extremely basic and introductory. You can find more advanced ones on the internet to obtain a deeper understanding of the working of RNN.

Here are the steps to implement the code:

1. Import all the necessary libraries.

Here is the code for that:

```
import numpy
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

In the next section, we will write a function that will convert the given set of arrays into a data matrix.

2. In this section, let us write a python function to create data matrices from the given inputs and read data from our existing dataset.

```

def create_dataset(data, look_back=1):
    X, Y = [], []
    for i in range(len(data)-look_back-1):
        a = data[i:(i+look_back), 0]
        X.append(a)
        Y.append(data[i + look_back, 0])
    return numpy.array(X), numpy.array(Y)

```

Here is the code snippet to read the data from the dataset:

```

# fix random seed for reproducibility
numpy.random.seed(7)
# load the dataset
df = read_csv('airline-passengers.csv', usecols=[1])
df.head()

```

In the next section, let's normalize the dataset and create the training and testing datasets.

3. In this section, we will normalize the dataset. We will create the training and testing datasets by using the `create_dataset` function.

Here is the code snippet:

```

data = df.values
print(data)
data = data.astype('float32')

# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
data = scaler.fit_transform(data)
# split into train and test sets
training_size = int(len(data) * 0.67)
testing_size = len(data) - training_size
train, test = data[0:training_size,:],
data[training_size:len(data),:]
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)

```

In the next section, let us create the LSTM network and fit it.

4. In this section, we will first re-shape the dataset to send it as inputs to our model. After re-shaping our dataset, we will create the LSTM network and fit our LSTM model with the data that we have.

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1,
trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1,
testX.shape[1]))

# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1,
verbose=2)

# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)

# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
```

5. Let us look at the root mean squared error generated.

```
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0],
testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))

# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(data)
trainPredictPlot[:, :] = numpy.nan
```

```

trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(data)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict) +
(look_back*2)+1:len(data)-1, :] = testPredict

```

### **Output:**

Train Score: 22.93 RMSE

Test Score: 47.60 RMSE

We can see from the scores that the error is ~23 out of 1000 for the training data and close to ~48 out of 1000 for the testing data. The metrics from the above calculation are not so bad for a basic RNN, and the error rate is fairly tolerable.

## **Conclusion**

Recurrent Neural networks have played a significant role in several of today's deep learning applications. As we juggle with sentiment analysis, image captioning, language translations and several other applications, where sequential processing input is essential, RNNs have stepped up and made our job easier.

In this chapter, you have learnt the working of RNN; it's applications and how to solve a real-time problem using RNN in Python.

Our next chapter will introduce you to another important class of deep neural networks, known as Convolutional Neural Network (CNN). CNNs are commonly applied for analyzing visual imagery and have interesting applications in the field of machine learning.

## **Quiz**

1. What is the one distinguishing factor present in RNN?
2. Does CuDNN increase the performance of a deep learning model?
3. What is the full form of LSTM, and how does it work?

4. What is the difference between one-to-one and many-to-one RNN model?
5. Research on any two applications of RNN in the real-time and share your findings.

# CHAPTER 13

## Convolutional Neural Networks

### Introduction

**Convolutional Neural Networks (CNN)** are becoming extremely popular, owing to the increasing need for analyzing visual images. Object detection and recognition is one of the common areas where CNN is being used extensively. A typical CNN gets its name from its multiple convolutional layers with filters, which forms a part of the CNN architecture. In this chapter, we will explore how CNN and its functionality in depth.

### Structure

- The first killer app of deep learning
- Representations of images
- Convolutions and pooling
- Step-by-step code walkthrough
- Advanced architecture and techniques

### Objective

- Understand the application of CNN in image recognition
- Learn to build a CNN model in Keras

### The first killer app of deep learning

Although perceptron existed in the pre-1970s, it did not regain popularity/dominance until AlexNet; a convolutional neural network outperformed the next best algorithm on the ImageNet competition in 2012 by more than 10%. It leads to a resurgence of interest in deep learning research and applications; and, this is what ended the AI winter. In many

ways, variants of the CNN architecture are still the killer applications of neural networks.

Here is an architecture diagram of AlexNet:

## AlexNet (Krizhevsky et al. 2012)

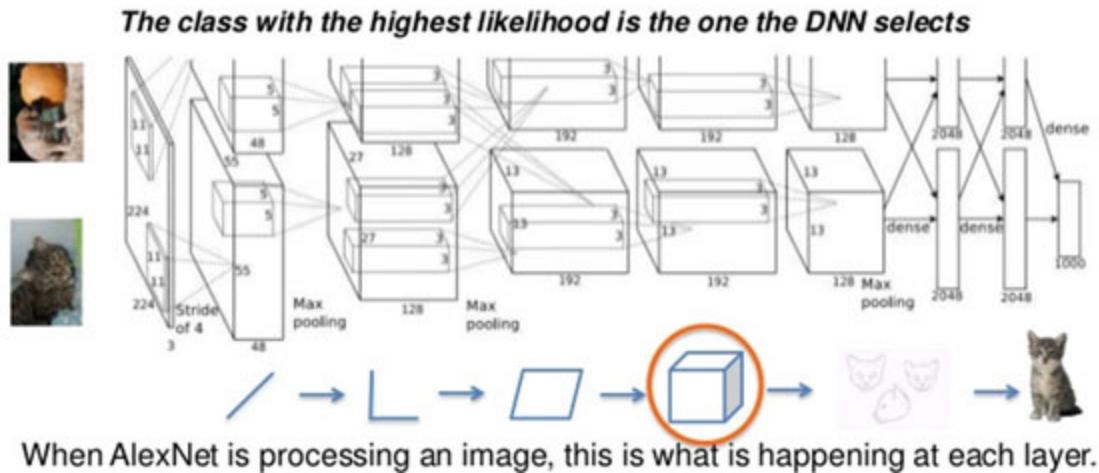


Figure 13.1: AlexNet

What makes CNNs differ from the neural network is the alternate layers of convolutional layers and then pooling layers reducing the pixels in an image into a feature vector, which acts as the input to a conventional dense or fully connected neural network with a SoftMax output layer.

The main advantage of convolution and then pooling layers is that it reduces the dimensions and the number of parameters needed versus if you were to use a fully connected neural network.

**NOTE:** Until ~2018, the advancements in computer vision and image related neural networks seems to have rapidly outpaced the other applications (say text classification for example, largely due to the natural and well-established *representation* of images as a matrix, later embedding became a well-adapted representation for text and progress accelerated in NLP. It perhaps illustrates the importance of representation to successful machine learning.

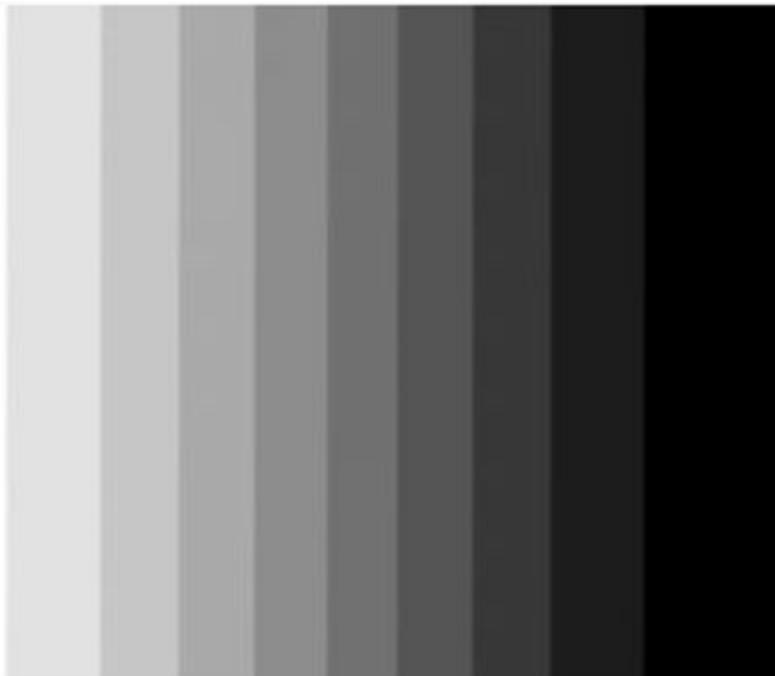
## How do we represent images?

First, let's explore how images are represented as numbers and then fed into a neural network as inputs. Each image is composed of pixels. Say you have a 4x4 (4 pixels wide and four long) image of a handwritten digit 1 (in grayscale).

It might look something like this:

200	0	0	255
255	200	0	255
255	255	0	255
255	100	0	100

**Table 13.1:** Image represented in pixels

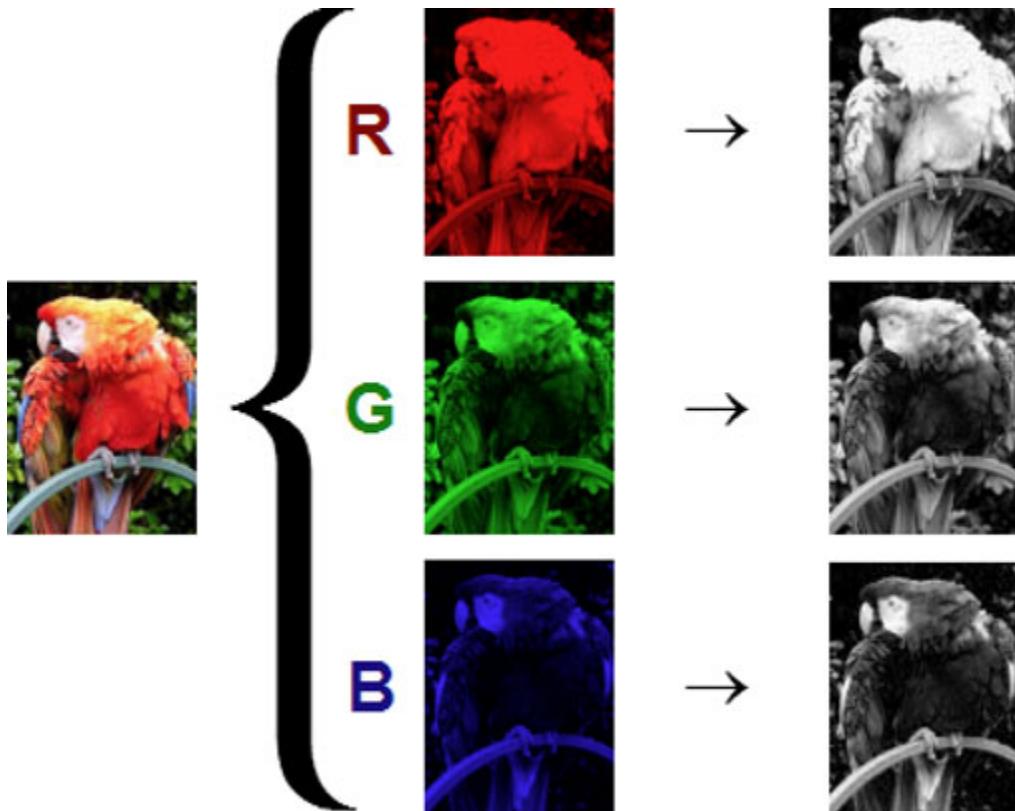


White (255) → Black (0)

**Figure 13.2:** Color gradient White -> Black

In the above table, each pixel value indicates *saturation* and ranges from 255 for pure white to 0 for pure black. Naturally, we can represent this as a matrix, just like we did.

But commonly, images are not black and white so that we would have three layers of "saturation" for Red, Green, and Blue, also known as the RGB channels:



**Figure 13.3:** Red, Green and Blue channels for an image

We can put the three different matrices into a nested array or a 3-D tensor. Each array is 2-D for *Width x Height*, and we add dimension for the dimension that indicates the RGB.

[R, G, B] where R, G, and B are 2D matrices similar to the above.

The image net dataset contains photos of size  $227 \times 227$  pixels and in color, so if we wanted to represent that as a flattened array to feed into a dense or fully connected NN as input, the input layer would need to be of size  $227 \times 227 \times 3 = 154,587$ .

**NOTE:** What is flatten?

Flatten is a process where the first element in the array would be the saturation of pixel at position 1,1 for the red channel, and the last element would be the saturation of the pixel at position 227, 227 for the blue channel, hence going from 3-D to 1-D.

Let's say the immediate next hidden layer has a size of 1/4 of the input layer (38647) we would have a weight matrix of  $38,647 \times 154,587$  = almost 5 billion parameters. As we know, the amount of data required to train a model in most cases scales exponentially with the number of dimensions we would need to have a lot of images and perform a large amount of computation.

In the next section, let's see how we can perform this computation with the help of CNN and where does the convolution neural network shine.

## Convolutions

Each convolution layer consists of sets of *Kernels*, sometimes called filters, although filters are typically stacked - like depth kernels. And those kernels learn to identify patterns in the input data.

In the example above (our greyscale image 1), we could learn a  $3 \times 3$  filter that identifies vertical lines (as a feature for the next layer).

The  $3 \times 3$  filter looks like the following:

-1	0	1
-1	0	1
-1	0	1

*Table 13.2:  $3 \times 3$  filter that identifies vertical lines*

When the filter is overlaid at the top left corner of the  $4 \times 4$  greyscale image matrix, do an element-wise product and sum the result. We get the below result:

200	0 x -1	0 x 0	255 x 1
255	200 x -1	0 x 0	255 x 1
255	255 x -1	0 x 0	255 x 1
255	100	0	100

*Table 13.3:  $4 \times 4$  greyscale image matrix operations*

If we sum up the values in the blue cells (or pixels), we get a value of 310.

If we move the filter to the top right corner, the pixel matrix will look like the following:

200	0	0	255
255	200	0	255
255	255	0	255
255	100	0	100

*Table 13.4: 4 x 4 greyscale image on changing the filter to top right corner*

Applying the same process as above for the filter placed on the top right corner, we get a sum of -710.

Therefore, it is likely that a vertical line feature exists in the first blue highlighted area (and similarly if we move the filter down 1 to the bottom right corner) and less so for the second highlighted area. It extends into the RBG channels, for which each filter you have a 3-tuple output for.

Now that we understand the underlying idea of convolution, we need to discuss terms that specify how the filters are to be applied. Each filter will learn a different set of patterns for each channel. For example, the filter may learn to identify vertical lines for the red channel but may learn to identify diagonal lines for the blue channel, and so on:

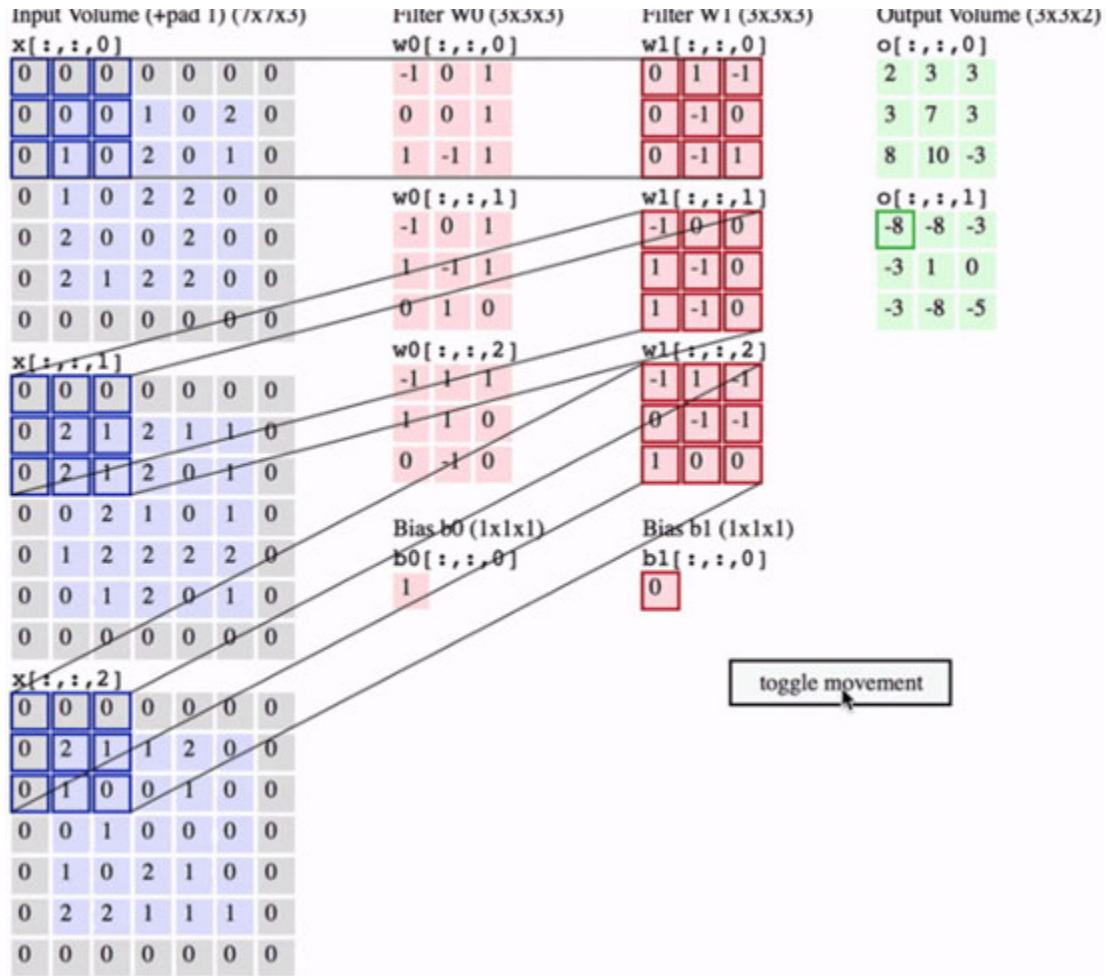


Figure 13.4: Convolutions in deep learning

## Application of filters across channels

There are a few key terms that describe how the filters should be applied across a channel (filters have a smaller size than the underlying image).

A key thing to note is that the number of channels for the image/depth and the number of channels for the filter must match. (If you are only interested in 1 of the channels, you could preprocess the image and only pass in a grayscale version for that channel or use all zero to initialize the filters for the irrelevant channels if you pass in a 3-channel image).

The result is a 2-D matrix with dimensions following the formula below:

- Stride(S): How many cells you move across and down at a time
- Padding(P):

- **Valid:** No padding, only the valid (real) data is used, and the output size of the convolution operation will be smaller
- **Same:** Add padding such that when the stride size is 1, the output and input will have the same dimension (pad with zeros)

The reason we may want to have padding is that we want to be able to learn from patterns at the edge of the image, that is, if images have a border of some sort, or if there is a circle at the very corner of the image. A snake along the very edge of the image is still a snake, and our classifier should learn to be able to handle that.

**Note:** Images with borders tend to be difficult to learn/cause reduced accuracy; it is a good idea to remove borders as a preprocessing step.

These are considered hyper-parameters of the model (not model weights themselves) and help determine how quickly you summarize information from the input and reduce parameters to feed into the next layer.

The relationship is described as:

$$\text{floor\_of}(\{(D + 2P - F) / S\} + 1)$$

Where D is the dimension of the image, P is the padding, and F is the dimension of the filter.

As you can see, the more padding and smaller stride size lead to less reduction in output size.

If you want to summarize information more quickly with each convolution layer, you should not pad the image, but do increase stride size and use a bigger filter.

For an image of size  $D \times D$ , Padding = P, Stride S and filter size of  $F \times F$ , the resulting output is:

$$\text{floor\_of}(\{(D + 2P - F) / S\} + 1) \times \text{floor\_of}(\{(D + 2P - F) / S\} + 1)$$

In dimension, which is a 2-D array/matrix.

If you had N filters, then there would be an additional dimension (depth) such that the resulting tensor (T) is sized:

$$(\text{height}) \times (\text{width}) \times (\text{depth})$$

$$\text{floor\_of}(\{(D + 2P - F) / S\} + 1) \times \text{floor\_of}(\{(D + 2P - F) / S\} + 1) \times N$$

This output matrix it's then fed into an activation function, for example, RELU, similar to the dense neural networks.

## Pooling

Pooling typically follows a convolution layer, which gives a similar effect —helps in the summarization of information. However, the mechanism in which it does so is somewhat unclear. We will see how it is implemented first.

Say your output from the previous convolution layer is a  $4 \times 4$  matrix as follows:

0	0	1	0
9	0	22	11
4	4	0	0
4	4	1	0

*Table 13.5:  $4 \times 4$  matrix from the previous convolutional layer*

If we have a filter size of  $2 \times 2$  and stride equal to 2, we can partition the above into four regions as follows:

- Upper left quadrant (in green)
- Lower left
- Upper right
- Lower right

We take the  $\max$  of each region and output a matrix of dimension  $2 \times 2$ :

9	22
4	1

*Table 13.6:  $2 \times 2$  matrix - max pooling*

It is max pooling.

If we apply the average operation instead of max, then it is average pooling.

One way to intuitively understand this is that we are condensing the spatial information of the features, which the filters have identified. Consider that

we have applied the vertical line or edge detection filter on the features.

## What is max pooling?

- Indicates that this feature is very prominent in the upper right quadrant
- It is not identified in the lower right quadrant
- It is less prominent in the green highlighted region
- Within each region, we don't really care as much about exactly which pixels show this pattern

## What is average pooling?

- Indicates that this feature is dominant in the upper right quadrant
- It is less dominant in the lower left quadrant
- Within each region, we don't really care as much about exactly which pixels show this pattern

If we want to learn a happy face pattern, we would want to see two circles in the upper quadrants and a curve or horizontal edge in the lower quadrants. It preserves some spatial information but at a lower fidelity. And this composition of features learned in the lower layers becomes an input into convolution and pooling layers deep in the network. It is leading to a compositional effect of features.

The lower layers, which are closer to input, learn simple patterns, and the higher layers compose them, eventually learning that two triangles on top of a circle might be a cat. It is mostly for intuition, the actuality of how or why this works is less clear:

## Deep Learning learns layers of features

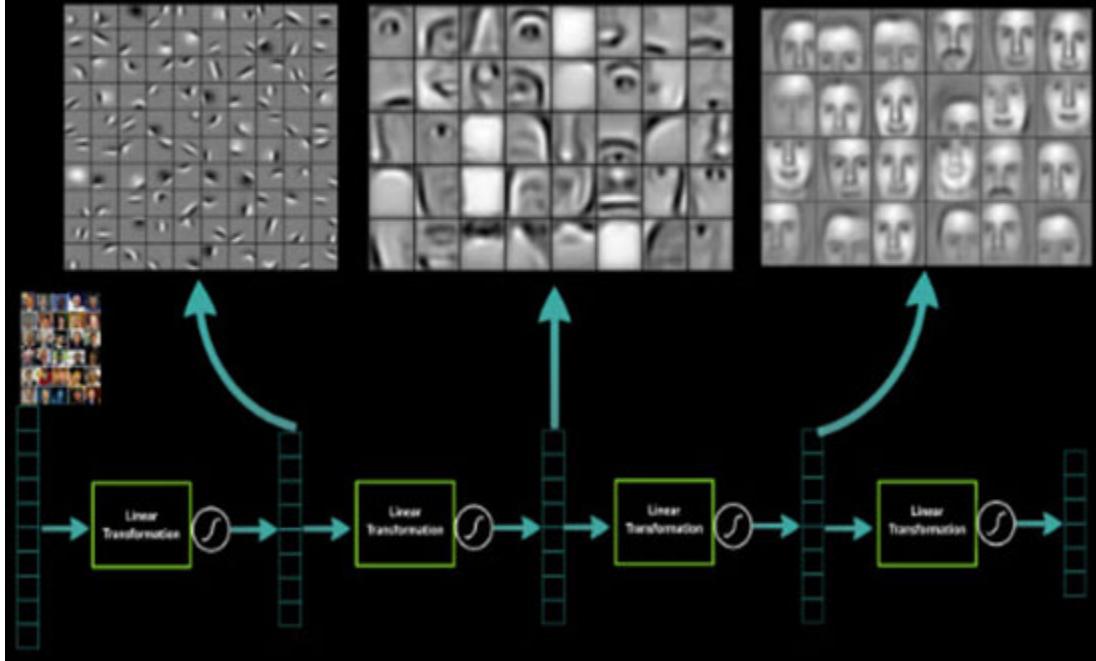


Figure 13.5: Deep Learning feature layers

Finally, after multiple layers of convolutions, followed by pooling layers, we end up with a high dimensional tensor as a feature map. We then use the flatten operation again to turn it into a 1-d array; and, this is what we feed into the "*classifier*" part of our network—the fully connected layers that had a soft-max output layer.

## Step-by-step code walkthrough

In the upcoming section, we present the code to build a simple classifier in Keras.

In this example, we will be using the in-built cifar10 dataset from Keras.

1. Import the dataset and print training data-related information.

Here is the code snippet:

```
from keras.datasets import cifar10
(x_train, Y_train), (x_test, Y_test) = cifar10.load_data()

# inspect the data
```

```
# print(x_train[0])
print(X_train.shape)
Y_train[:5]
```

Here is the output image for the above code snippet:

```
(50000, 32, 32, 3)
array([[6,
       [9],
       [9],
       [4],
       [1]], dtype=uint8)
```

**Figure 13.6:** Output describing the training data

In the next step, let's import the needed elements to train our model.

2. As you can see in the above image, we have 50k images, 32 x 32 pixels each, and three channels for RGB.

In the below code, we import the needed elements to construct and train the model:

```
import numpy as np

from keras.callbacks import EarlyStopping
from keras.datasets import cifar10
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Flatten
from keras.layers import LeakyReLU
from keras.layers.convolutional import Conv2D
from keras.optimizers import Adam
from keras.layers.pooling import MaxPooling2D
from keras.utils import to_categorical

np.random.seed(42)
# set random seed for drop out etc... otherwise results are
not reproducible / unreliable
```

```
# define model name and initialize using the Sequential API  
(which allows us to easily build our network layer by  
layer)  
classier = Sequential()
```

**Keras provides two APIs for model construction: Functional and Sequential. For most use cases, the Sequential API is preferred because it's more intuitive to use. However, the Functional API is more flexible and allows for layer sharing and even acyclic graphs networks and other advanced architecture. It is important to keep track of the input and output dimensions properly for the Functional API.**

3. In the below section of code, we specify the initial convolutional block of our CNN, which typically follows the pattern:

*convolution => max pool => activation => (dropout)*

Here is the code snippet:

```
'''  
  
- adding a 32 filters with dimensions of 3x3  
- padding to learn features at the edge  
- input shape is 32x32 pixel img, depth of 3 for RGB  
channels  
'''  
  
classier.add(Conv2D(32, kernel_size=(3, 3), padding='same',  
input_shape=(32, 32, 3)))  
'''  
  
- leakyReLU tend to perform better than RELU because there  
is a different slope (flatter slope, default = 0.3) for the  
negative values  
- standard ReLU propagates less information to the next  
layers because gradient is zero for all negative values  
- LeakyReLU is one solution (ELU also) that addresses the  
Dying RELU problem where a large gradient update causes the  
RELU neuron unit to never activate again  
'''
```

```
        classifier.add(LeakyReLU(alpha=0.3))
        classifier.add(Conv2D(64, padding='same', kernel_size=(3,
3)))
        classifier.add(LeakyReLU(alpha=0.3))
        classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

4. In the next section, we add the dropout layer for regularization. To understand what dropout does in a better manner, please see the intro to NN and the discussion on regularization techniques for NN.

Here is the code snippet:

```
"""
- drop out is used to regularize neural networks
- adding dropout after pooling layer tend to lead to better
performance
"""
classifier.add(Dropout(0.25))
```

5. In the next step, we will add another convolutional block for CNN:

```
"""
the following block is similar in structure to previous
blocks but with more filters
"""

classifier.add(Conv2D(128, kernel_size=(3, 3)))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
classifier.add(Conv2D(128, kernel_size=(3, 3)))
classifier.add(LeakyReLU(alpha=0.3))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
classifier.add(Dropout(0.25))
classifier.add(Flatten()) # Flatten the feature map tensor to
1D array
```

6. In the next step, we will define dense layers that consumes the feature array and produces a classification:

```
"""
below is the classification Dense Neural Net
```

```
typically a shallow DNN performs sufficiently and is faster  
to train  
'''  
  
classier.add(Dense(1024))  
classier.add(LeakyReLU(alpha=0.3))  
classier.add(Dropout(0.5))  
classier.add(Dense(10, activation='softmax'))
```

## 7. At this point, our model is built, we move on to compilation and training:

```
'''  
  
after the model architecture has been specified (roughly  
equivalent to specifying the computational graph)  
we can compile the model by providing the loss function and  
optimizer along with metrics you want to track  
it is appropriate to use categorical_crossentropy as a loss  
function because the softmax layer is size > 2 (if 2, use  
binary_crossentropy) and classes are mutually exclusive, if  
not use binary_crossentropy  
ADAM optimizer with a lower learning rate and low decay is  
a safe option especially because we are using early stop  
callback (see below), we are not concerned about training  
time for a relatively simple dataset especially when  
training on GPU  
'''  
  
classier.compile(loss='categorical_crossentropy',  
                  optimizer=Adam(lr=0.0001, decay=1e-6),  
                  metrics=['accuracy'])
```

## 8. Now we will start training the model.

```
'''  
  
Train the model:  
- the X_train / 255 is a way to normalize the data since  
the pixel values go from 0-255  
- not normalizing the data, especially for images, will  
lead to substantial degradation in model performance
```

```

- we need to convert the labels (integer from 0-9) into a
one-hot encoded vector of size 10 for the loss function
- the early stop callback reduces overfitting, when the (by
default) validation loss has not improved more than 0.01 in
3 epochs, we terminate training
- another practical advantage of using early stop is it
removes the need to tune number of epochs as a hyper
parameter (set too low, model could have performed better,
set too high overfit and waste compute / $)
"""

history = classier.fit(X_train / 255.0,
to_categorical(Y_train),
batch_size=128,
shuffle=True,
epochs=250,
validation_data=(X_test / 255.0,
to_categorical(Y_test)),
callbacks=[EarlyStopping(min_delta=0.01,
patience=4)])

```

## **Output:**

```

Train on 50000 samples, validate on 10000 samples
Epoch 1/250
50000/50000 [=====] - 7s
149us/step - loss: 0.7181 - acc: 0.7510 - val_loss: 0.7807
- val_acc: 0.7319
Epoch 2/250
50000/50000 [=====] - 7s
136us/step - loss: 0.7121 - acc: 0.7527 - val_loss: 0.7939
- val_acc: 0.7272
Epoch 3/250
50000/50000 [=====] - 7s
137us/step - loss: 0.7028 - acc: 0.7564 - val_loss: 0.7750
- val_acc: 0.7338
Epoch 4/250
50000/50000 [=====] - 7s
136us/step - loss: 0.6961 - acc: 0.7564 - val_loss: 0.7561

```

```

- val_acc: 0.7392
Epoch 5/250
50000/50000 [=====] - 7s
138us/step - loss: 0.6861 - acc: 0.7612 - val_loss: 0.7548
- val_acc: 0.7393
Epoch 6/250
50000/50000 [=====] - 7s
139us/step - loss: 0.6768 - acc: 0.7654 - val_loss: 0.7819
- val_acc: 0.7338
Epoch 7/250
50000/50000 [=====] - 7s
137us/step - loss: 0.6671 - acc: 0.7678 - val_loss: 0.7553
- val_acc: 0.7430
Epoch 8/250
50000/50000 [=====] - 7s
136us/step - loss: 0.6597 - acc: 0.7692 - val_loss: 0.7532
- val_acc: 0.7419

```

The early stop was triggered, we see the training progress as expected, and we see that the training and test accuracy is relatively close. It does not raise concerns but lets us further evaluate the model.

## 9. The next step is the performance evaluation of our model:

```

# calculate the accuracy on the test set, it is important
# to normalize your training and test data the same way
scores = classifier.evaluate(X_test / 255.0,
                             to_categorical(Y_test))

print('Loss: %.3f' % scores[0])
print('Accuracy: %.3f' % scores[1])

```

### **Output:**

```

10000/10000 [=====] - 1s
114us/step
Loss: 0.753
Accuracy: 0.742

```

For a relatively small model that was quick to train (7 seconds per epoch on a GPU, eight epochs, approximately 1 minute), and the

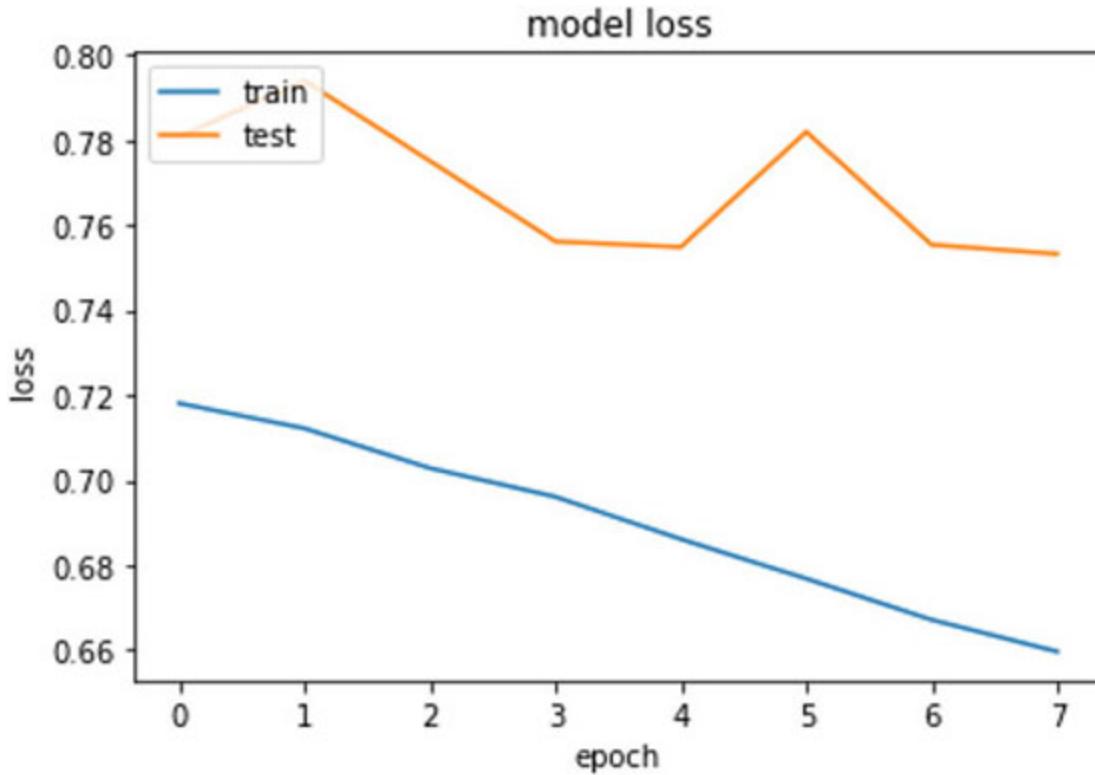
accuracy is sufficient. The model does not indicate any bugs or errors.

**Usually, the first epoch takes longer due to the extra parameter initialization required, but that was not shown in the output above.**

## 10. Check for overfitting:

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Here is the figure that shows the output for the above code snippet:



*Figure 13.7: Overfitting graph for our CNN model*

We see that the gap between the training and test loss remained relatively consistent.

It does not indicate overfitting.

## 11. Predict using our built model.

Let's select a random image from the test set and see how the model performs. The code snippet is below:

```
sample_id = 108  
  
plt.figure(figsize=(4, 4))  
plt.imshow(X_test[sample_id])  
plt.axis('off')  
print(Y_test[sample_id])
```

### **Output:**

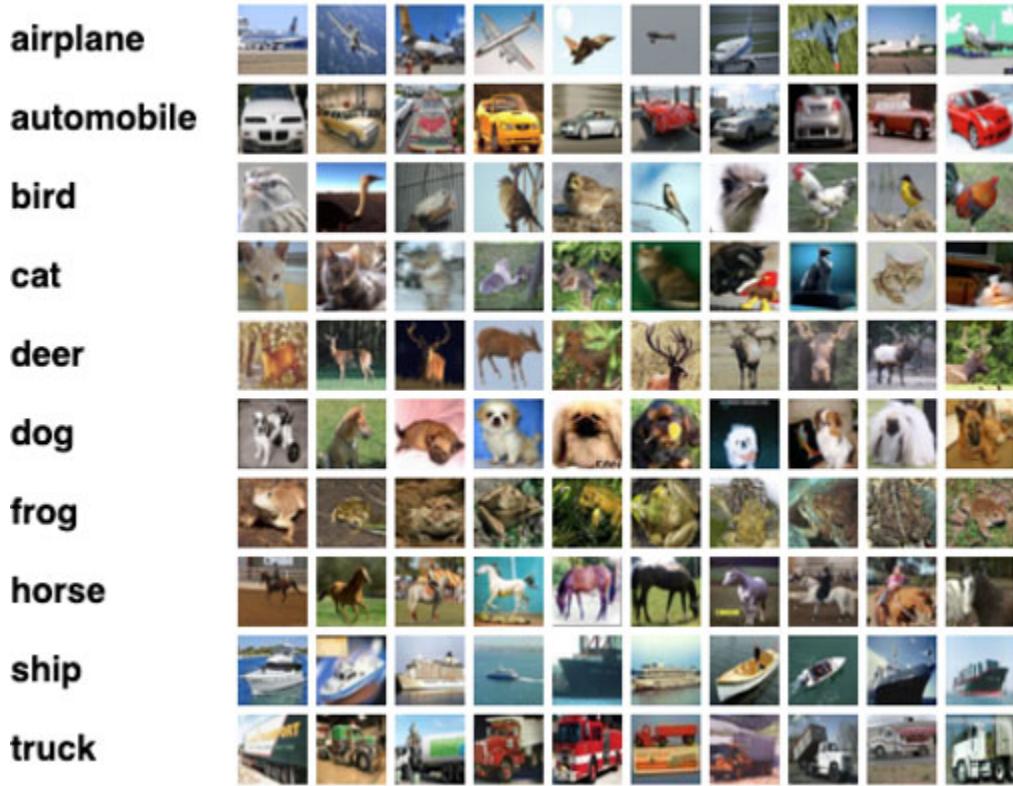
```
[8]
```

Here is the image output:



*Figure 13.8: Output of sample prediction*

It looks like a ship. And if we reference the data source below class 9 (labeled eight because the indices start at 0), this is indeed a ship:



*Figure 13.9: Image collection of CIFAR-10 dataset*

**The images in CFAR-10 are very small, so it looks pixelated to human eyes. The benefit of using small images (especially for a benchmark) is that training takes less time/we will not have to spend too much money and energy on computing. It is interesting to consider the fact that the model does a reasonable job on such images where even as a human, it's difficult to be certain.**

## 12. Continuing to make a prediction using our model.

Here is the code snippet:

```
'''  
make a prediction using the model:  
- notice that the dimensions of the test data is the same  
as training data, it is 4D by default [index, height,  
width, channel]  
- we must reshape the sampled image to a shape the model  
expects by removing the outer dimension (remove the index
```

```
dimension). This is the same reason why the label output
from above is [8] instead of 8 by default.
- we also need to normalize our data the same way we did
during training
'''

softmax_output = classifier.predict(X_test[108].reshape((1,))
+ X_test[108].shape)/255.)
softmax_output
```

## Output:

```
array([[7.6884049e-04, 4.3487982e-04, 3.0618946e-03,
7.0881476e-03, 1.7217095e-05, 2.1384763e-05, 2.6435079e-03,
9.3263647e-05, 9.8584425e-01, 2.6502352e-05]],  
dtype=float32)
```

By visual inspection, we notice array element 8 is the highest value by a few orders of magnitude. The second best guess, according to our model, would be Cat.

A quick way to just get the prediction is to use `argmax`:  
Here is the code snippet:

```
np.argmax(softmax_output)
```

## Output:

```
8
```

Let's also evaluate which classes our model tends to perform well on and which ones does it get wrong (and which ones tend to get confused):

```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(Y_test,
Y_preds.argmax(axis=1))

confusion_matrix
```

## Output:

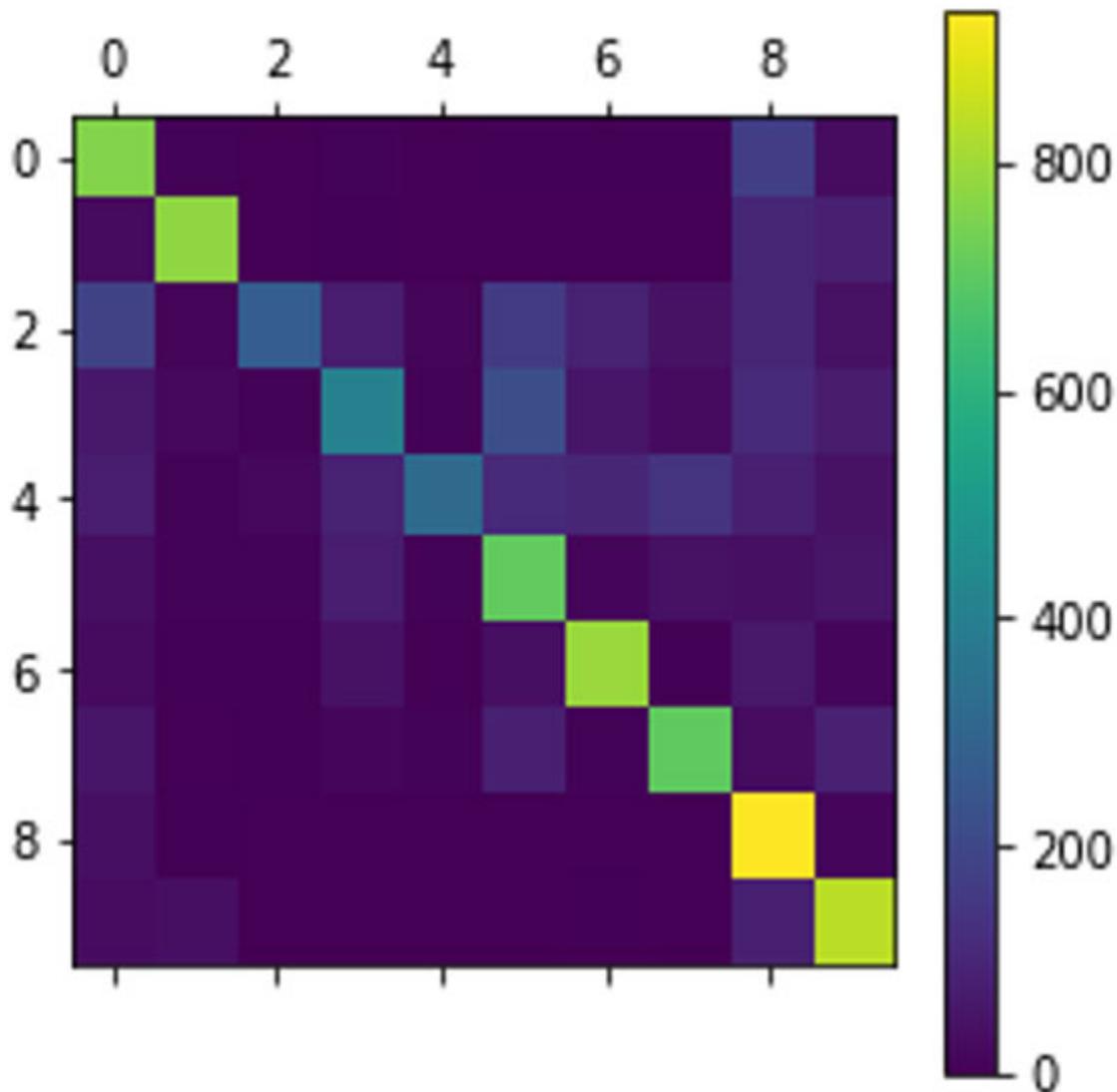
```
array([[762,    9,    2,    9,    1,    6,    5,    4, 172,   30],  
       [ 26, 781,    0,    5,    0,    2,    3,    0, 100,   83],  
       [185,   11, 280,   76,   11, 162,   88,   44, 101,   42],  
       [ 62,   19,   10, 411,   10, 223,   57,   27, 110,   71],  
       [ 76,    9,   20,   90, 325, 111,   99, 141,   83,   46],  
       [ 38,    8,    9,   76,    9, 710,   18,   41,   38,   53],  
       [ 26,    5,    4,   46,    2,   40, 794,    4,   61,   18],  
       [ 55,    2,    5,   17,    8,   81,    9, 705,   31,   87],  
       [ 37,    6,    0,    1,    0,    2,    1,    1, 935,   17],  
       [ 32,   37,    1,    1,    0,    1,    4,    2,   84, 838]])
```

The confusion matrix in matrix form is hard to evaluate with so many classes. We can plot this matrix using the below code snippet for a quick visualization:

```
plt.matshow(confusion_matrix)  
plt.colorbar()
```

Here is the output image for the confusion matrix.

**Output:**



*Figure 13.10: Confusion matrix*

Notice that class 8 (ships) have the best performance, followed by class 9 (truck).

Notice that class 2 and 4 have the worst performance. In particular, class 2 (bird) tends to get misclassified as class 0 (airplane) or 5 (deer). The confusion between birds and airplanes is perhaps expected. The confusion between bird and deer could be due to the shape of the antlers on deer that looks similar to wings on birds.

There is a significant speed difference when training on the CPU.

For reference, here is an example screenshot of training progress on a CPU (same model):

```

Train on 50000 samples, validate on 10000 samples
Epoch 1/250
50000/50000 [=====] - 278s 6ms/step - loss: 1.9444 - acc: 0.2832 - val_loss: 1.7105 - val_acc: 0.3735
Epoch 2/250
50000/50000 [=====] - 277s 6ms/step - loss: 1.6258 - acc: 0.4070 - val_loss: 1.5407 - val_acc: 0.4369
Epoch 3/250
50000/50000 [=====] - 278s 6ms/step - loss: 1.4828 - acc: 0.4617 - val_loss: 1.3924 - val_acc: 0.4961
Epoch 4/250
50000/50000 [=====] - 278s 6ms/step - loss: 1.3932 - acc: 0.4990 - val_loss: 1.3270 - val_acc: 0.5219
Epoch 5/250
50000/50000 [=====] - 278s 6ms/step - loss: 1.3267 - acc: 0.5267 - val_loss: 1.2870 - val_acc: 0.5354
Epoch 6/250
50000/50000 [=====] - 279s 6ms/step - loss: 1.2650 - acc: 0.5529 - val_loss: 1.2248 - val_acc: 0.5613
Epoch 7/250
50000/50000 [=====] - 278s 6ms/step - loss: 1.2154 - acc: 0.5683 - val_loss: 1.1501 - val_acc: 0.5912
Epoch 8/250
50000/50000 [=====] - 279s 6ms/step - loss: 1.1679 - acc: 0.5899 - val_loss: 1.1454 - val_acc: 0.5920
Epoch 9/250
50000/50000 [=====] - 277s 6ms/step - loss: 1.1288 - acc: 0.6019 - val_loss: 1.0985 - val_acc: 0.6107
Epoch 10/250

```

**Figure 13.11: Training time on a CPU**

Notice the difference in wall-clock time for each epoch. (This will vary widely depending on what hardware you are using.)

In the next section, let's take a look at some of the advanced architecture concepts and techniques.

## Advanced architecture and techniques

Here are some advanced techniques that we can apply in the space of CNN to make our models better and derive better insights.

## Data augmentation

As mentioned in Performance Evaluation. One way to improve both accuracies and reduce overfitting is by getting more data. One way to approach this is by generating synthetic data that is based on training data but has a slightly different distribution.

Consider you have an image of a cat facing left - the classifier should still predict cat if you provided a photo of the same cat facing right. Similarly, a cat photo taken in a darker light / more lighting should still be predicted as a cat.

What this indicates is that we can artificially take one image and apply a variety of preprocessing steps to artificially generate additional but slightly varied training data, which will help the model achieve higher accuracy and also generalize better.

Here is some example code that accomplishes the above:

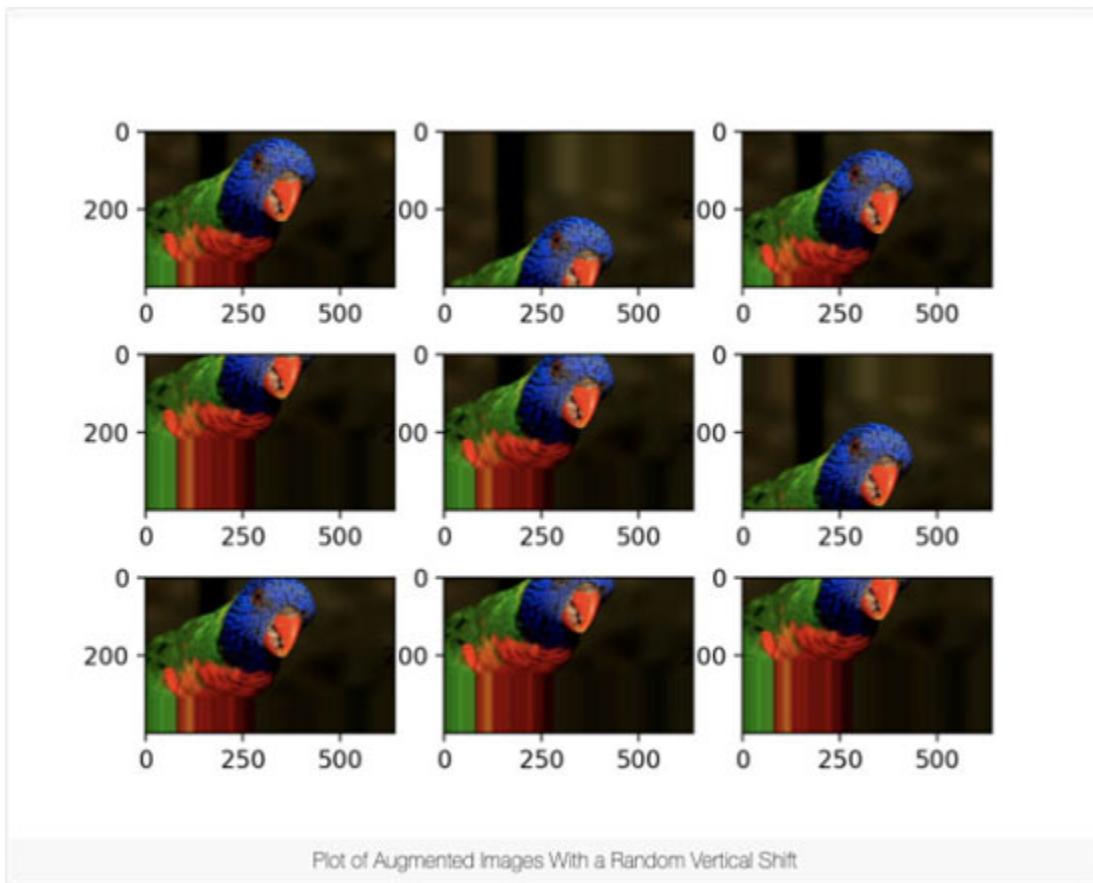
```
from keras.preprocessing.image import ImageDataGenerator
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

datagen = ImageDataGenerator(
    featurewise_center=True, # centers to zero mean
    featurewise_std_normalization=True, # scales according to std
    (over dataset)
    rotation_range=20, # max degree of random rotation
    width_shift_range=0.2, # max amount of random shift
    (horizontal)
    height_shift_range=0.2, # max amount of random shift
    (vertical)
    horizontal_flip=True) # horizontal flip allowed?

datagen.fit(x_train) # fit the ImageDataGenerator on your
dataset
```

Here is an image demonstrating what `height_shift_range` does:



*Figure 13.12: Augmented Images with a vertical shift*

## Practical notes

Here are some practical notes that can help you debug your model and create a better performing model based on your dataset:

- Know what augmentation method is appropriate: For example, if working on OCR, flipping words and letters around would lead to worse results.
- Transform training and test/inference data the same way. Whatever normalization/preprocessing/transformation you applied to your training data should be applied to your test set and also during inference (application). Otherwise, your model will behave unpredictably.

## Transfer learning

Major advances in computer vision applications were driven by transfer learning techniques. Transfer learning generally entails:

- Taking a pre-trained network (either downloaded model with pre-trained weights or less commonly you trained it on a different/generic dataset)
- Keeping the lower level features (read filters/kernels) learned on previous training data
- Freezing the weights in these borrowed layers/making the weights in earlier layers untrainable
- Adding a new custom network (layers) on top of the borrowed layers
- Fine-tune / train the weights in the new custom layers
- (Optionally) iteratively unfreezing the borrowed layers for additional fine-tuning

The key advantage is that we are frequently able to achieve very high accuracy on a custom dataset that would be impossible to achieve if you started to train a network from scratch. (Because there isn't enough training data specific to your use case.) The reason transfer learning works so well in computer vision applications is that the earlier features like

horizontal/vertical edges and curves are common and useful in nearly all other CNN tasks.

The key is to freeze the borrowed/transferred layers initially (until the custom / new layers) have converged because:

- The randomly initialized weights in the new/custom layers are expected to lead to large losses initially
- In the back-propagation process, the errors caused by randomly initialized/untrained weights will cause weights to update in the transferred layers
- Hence destroying much of the learned features and benefits of transfer learning

Below is a small code sample that shows how to download a model with trained weights from Keras and how you can freeze layers to complete transfer learning:

```
from keras import applications
from keras.models import Model
# use the functional API instead of sequential

vgg16 = applications.VGG16(weights='imagenet',
include_top=False)
# don't need the last layer we are not using it anyways

vgg16.summary()
```

## **Output:**

```
Model: "vgg16"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_4 (InputLayer)	(None, None, None, 3)	0
block1_conv1 (Conv2D)	(None, None, None, 64)	1792
block1_conv2 (Conv2D)	(None, None, None, 64)	36928
block1_pool (MaxPooling2D)	(None, None, None, 64)	0
block2_conv1 (Conv2D)	(None, None, None, 128)	73856
block2_conv2 (Conv2D)	(None, None, None, 128)	147584
block2_pool (MaxPooling2D)	(None, None, None, 128)	0
block3_conv1 (Conv2D)	(None, None, None, 256)	295168
 ...		
block5_conv2 (Conv2D)	(None, None, None, 512)	2359808
block5_conv3 (Conv2D)	(None, None, None, 512)	2359808
block5_pool (MaxPooling2D)	(None, None, None, 512)	0
<hr/>		

```
Total params: 14,714,688
```

```
Trainable params: 14,714,688
```

```
Non-trainable params: 0
```

Let's say the two datasets are very different, so we only expect to make use of the very bottom of the first block: (up to `block1_pool`).

Here is the modified code snippet for the above:

```
vgg16.layers  
Vgg16.layers[3].output.shape
```

## Output:

```
TensorShape([Dimension(None), Dimension(112), Dimension(112),
Dimension(64)])
```

Here is the code snippet for the output and the next steps:

```
inp = vgg16.layers[3].output
# start with the output of the last borrowed layer
'''

define a custom network on top of borrowed layer, the layers
below will start with untrained weight

notice how the functional API takes the previous layer as input
of the next layer. (keras takes care of most of the dimension
calculations automatically)
'''

inp = Conv2D(filters=128, kernel_size=(3, 3),
activation='relu')(inp)
inp = MaxPooling2D(pool_size=(3, 3))(inp)
inp = Conv2D(filters=128, kernel_size=(3, 3),
activation='relu')(inp)
inp = MaxPooling2D(pool_size=(3, 3))(inp)
inp = Flatten()(inp)
inp = Dense(256, activation='relu')(inp)
inp = Dropout(0.5)(inp)
inp = Dense(10, activation='softmax')(inp)
'''

define new model by specifying the i/o of the new model
'''

new_model = Model(input=vgg16.input, output=inp)
'''
```

Remember to freeze the already trained layers (freeze up to not including layer 4).

You can set trainable = True later (optional).

Here is the last series of steps to finish compilation and training:

```
'''

for layer in new_model.layers[:4]:
```

```

layer.trainable = False
"""
compile and train as usual
"""

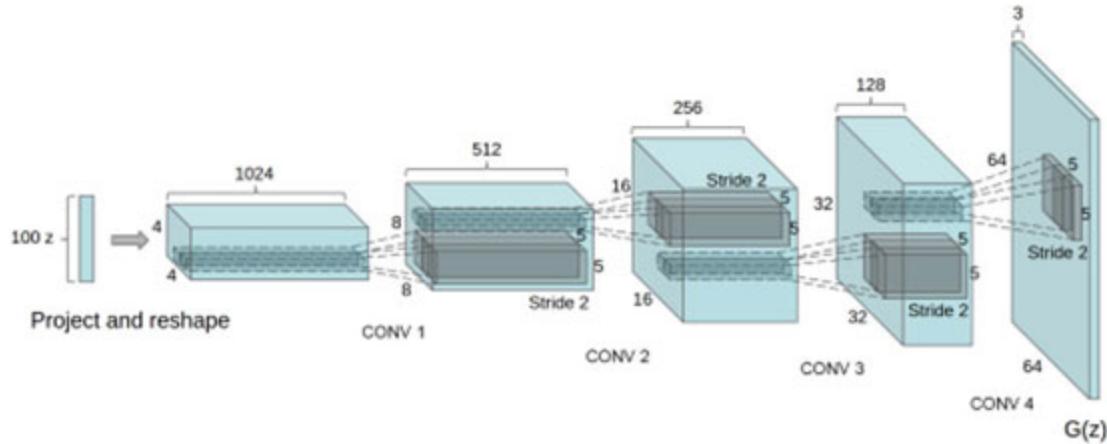
```

## Examples of advanced CNN task and architecture

In this section, let us look at some advanced CNN related tasks and the architectures required to perform these tasks.

### DCGANs (Deep Convolutional Generative Adversarial Networks)

GANs, also known as Generative Adversarial Networks, play an important role in the convolutional neural network models. DCGAN is a popular network design for GAN. Convolutional stride and transposed convolution are used for the down-sampling and up-sampling of the data present:



*Figure 13.13: DCGAN*

Here is an image that represents the Generative Adversarial Network:

# Generative Adversarial Network

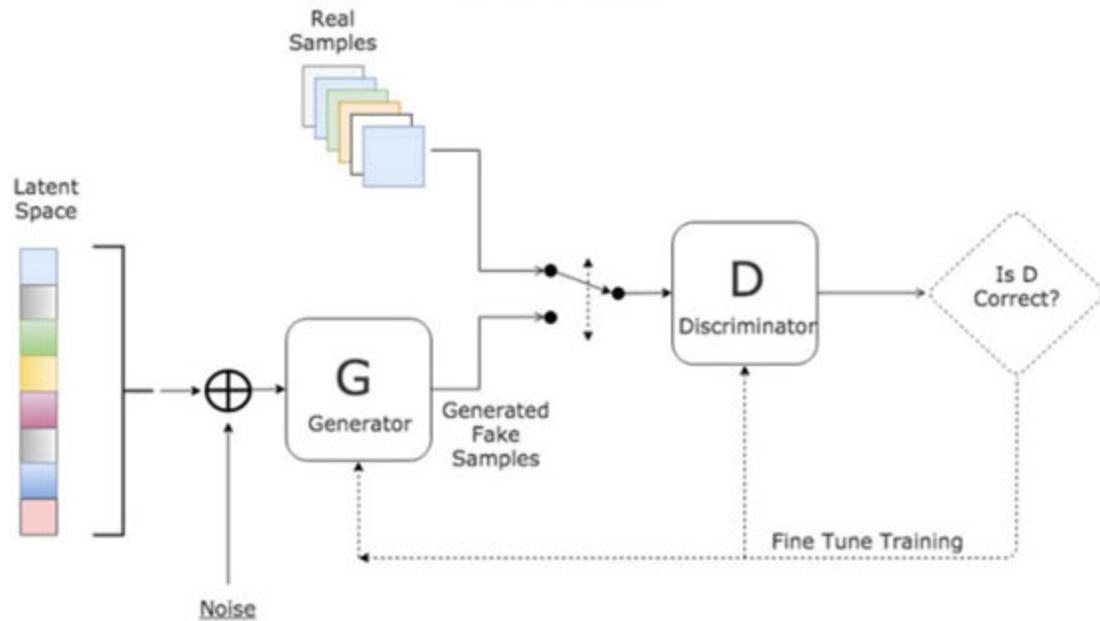


Figure 13.14: Generative Adversarial Network (GAN)

The objective of a Generative Adversarial Network (GAN) is to train a neural network to take a random noise vector ( $Z$ ), usually sampled from a gaussian distribution, and output a realistic image of the training data.

There are two parts to a GAN:

- The discriminator and the generator (the generator architecture is shown in the first image)
- The discriminator is provided with some generated images and also some real images sampled from data

The generator acts as a binary classifier: it will predict one if the image is real or sampled from the training set; and, it will predict 0 if the image is generated. The real and fake images are labeled accordingly as well. Notice that we can generate the label for the discriminator easily. Then after one round of weight update on the discriminator, the weights are frozen to allow the error to update the generator network. Note that the generator is provided with label one always. It is because if the generator succeeds in *fooling* the discriminator, then the outcome should be one not 0. We then

generate more improved fake images, unfreeze the weights on the discriminator, and repeat.

While the idea is simple: we expect the discriminator to get better at telling fake from real images, and this forces the generator to generate increasingly realistic versions. (See sample results below – [Figure 13.15](#)). In reality, GANs are notoriously difficult to train due to problems, like mode collapse, instability, and so on:



*Figure 13.15: Generated bedroom images after five epochs of training*

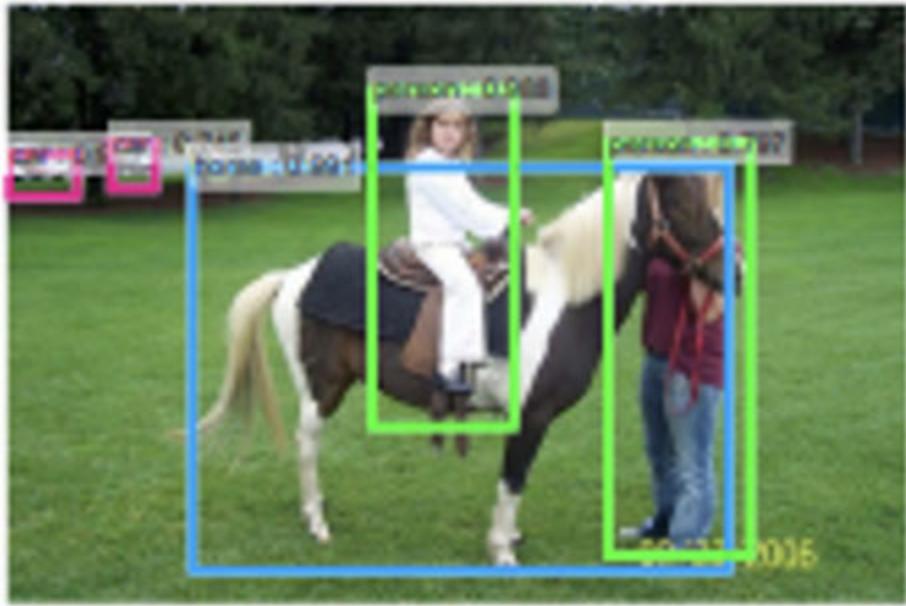
Subsequent papers have studied and proposed improvements to GANs (such as <https://arxiv.org/abs/1606.03498>) such as WGAN, Conditional GAN, and so on.

## Object localization

Classify the object and draw a box around where it is.

Notice there is a new element to the problem, a bounding box.

Here is an image to describe the above scenario:



*Figure 13.16: Object localization example*

Almost all modern object localization involves 2+ losses (multi-task loss):

1. Classify the object correctly
2. Specify the bounding box as closely as possible to the labeled data

The images are pre-labeled with a box and a class around each object.

We will discuss one of the more advanced architecture for this problem, **YOLO (You Only Look Once, one-shot learning)**. This network is named so and also has higher performance versus previous architecture because it allows for learning the classification and the bounding at the same time using five different loss functions.

It works as follows:

- Each image is split into smaller grids (regions)
- Each grid will predict two bounding boxes specified as  $(x, y, H, W, c)$  ( $x, y$  being the center of the bounding box, the  $H \times W$  are the height and width of the bounding box with  $x, y$  at its center,  $c$  is the confidence for the object in the bounding box)

When training:

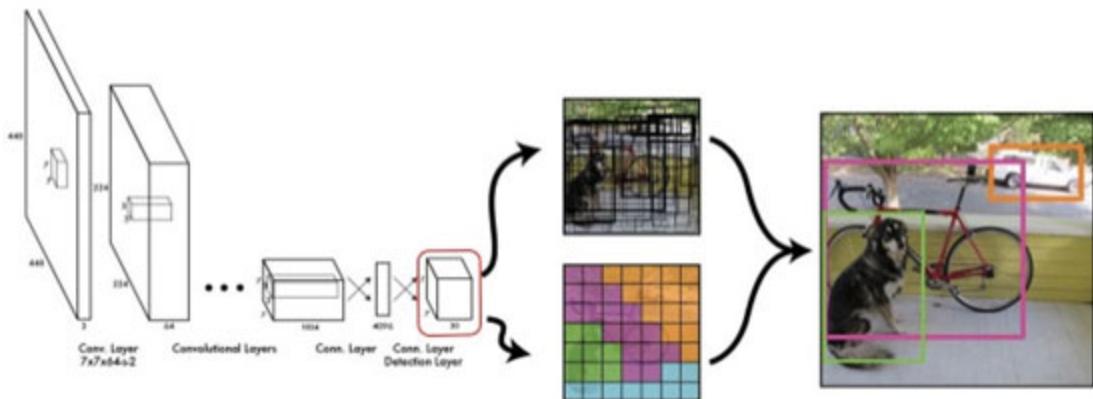
- Only the box with the highest overlap with the object goes through backprop

- There is the addition of an Object/no Object binary classifier each bounding box (this also makes learning fast, because it only updates the useful boxes)
- The boxes with more overlap (measured by IOU-intersection over union) should update such that it gets a higher confidence score and the x, y, H, W parameters are updated similar to a regression problem
- The classification loss is categorical cross-entropy (note: Binary cross-entropy in YOLO V3 with anchors [bounding box prior] as each box could have multiple classes => multi-label):

$$\begin{aligned}
& \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
& + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \right] \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left( C_i - \hat{C}_i \right)^2 \\
& + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} \left( C_i - \hat{C}_i \right)^2 \\
& + \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}$$

**Figure 13.17:** YOLO – Single-shot detectors

## YOLO: You Only Look Once

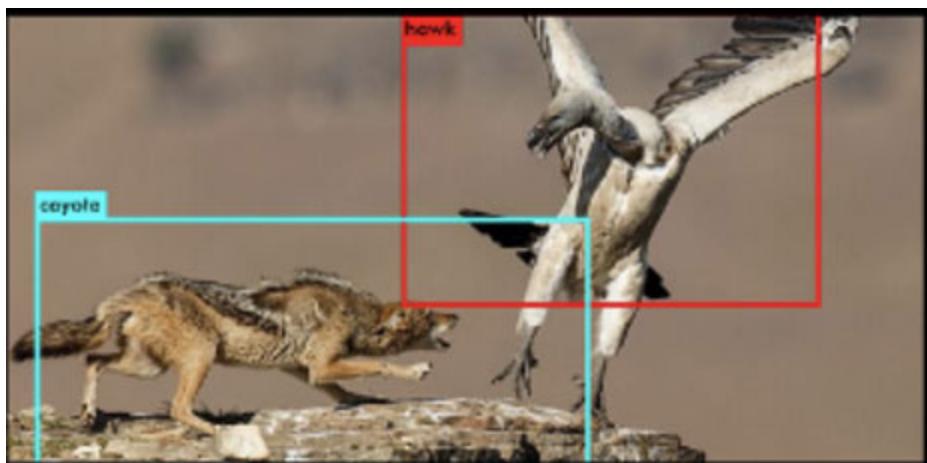


Redmon et al. [You Only Look Once: Unified, Real-Time Object Detection](#). CVPR 2016

30

*Figure 13.18: Working of YOLO*

Here is another example of how objects can be localized and identified:



*Figure 13.19: YOLO Example*

During prediction:

- After the model is trained, you end up with a lot of overlapping boxes with varying confidence scores
- A filtering algorithm is applied to combine bounding boxes with the highest confidence and make one final bounding box with object classification (Non-Max Suppression)

## Conclusion

Convolutional neural networks play an important role in the world of machine learning, especially when it comes to applications like object detection, face recognition, and so on.

In this chapter, we saw how we could represent an image in the matrix format and apply convolutional neural networks to build an image recognition model. We also went through some advanced techniques and architectures, which can help in building models if the images are not easy to recognize or distinguish.

We also saw some CNN related code that can be written in Keras, to understand better about CNN implementations in real-time.

In the next chapter, we will take a look at the different performance metrics that can help improve a machine learning model.

## Quiz

1. What is the biggest advantage of using CNN?
2. Share any two use cases of CNN in the industry.
3. What are the two types of pooling?
4. What is the difference between CNN and ANN?
5. What is Flattening?

## Image reference links

- *Figure 13.1* source: <https://image.slidesharecdn.com/pydatatalk-150729202131-lva1-app6892/95/deep-learning-with-python-pydata-seattle-2015-35-638.jpg?cb=1438315555>
- *Figure 13.2* reference: <https://learnlearn.uk/gcsecs/grayscale-image-representation/>
- *Figure 13.3* reference: [https://upload.wikimedia.org/wikipedia/commons/5/56/RGB\\_channels\\_separation.png](https://upload.wikimedia.org/wikipedia/commons/5/56/RGB_channels_separation.png)
- *Figure 13.4* reference: <https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d>

- Figure 13.5 source: <https://www.kdnuggets.com/wp-content/uploads/deep-learning-feature-layers.jpg>
- Figure 13.12 source: <https://3eqpr26caki16dnhd19sv6by6v-wpengine.netdna-ssl.com/wp-content/uploads/2019/01/Plot-of-Augmented-Images-with-a-Vertical-Shift.png>
- Figure 13.13 reference: <https://arxiv.org/pdf/1511.06434.pdf>
- Figure 13.14 reference: [https://cdn-images-1.medium.com/max/1600/1\\*5rMmuXmAquGTT-odw-bOpw.jpeg](https://cdn-images-1.medium.com/max/1600/1*5rMmuXmAquGTT-odw-bOpw.jpeg)
- Figure 13.16 reference: [https://miro.medium.com/max/781/1\\*hLM5gqM7l4fXBvXDjM6khw.png](https://miro.medium.com/max/781/1*hLM5gqM7l4fXBvXDjM6khw.png)
- Figure 13.17 reference: <https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/single-shot-detectors/yolo.html>
- Figure 13.18 reference: <https://i.stack.imgur.com/OPGDq.jpg>
- Figure 13.19 reference: <https://arxiv.org/pdf/1612.08242.pdf>

# CHAPTER 14

## Performance Metrics

### Introduction

Solving case studies is a great way to understand a concept end-to-end. In this chapter, we will pick up once the case study and look at how we can design the solution for a particular problem using the principles of machine learning. We will explore the different steps of creating a simple prototype solution - starting from data collection to ML modeling.

### Structure

- The usefulness of a model
- Metrics to use
- Overfitting, underfitting, variance, and bias
- How to make your model useful
- Other practical considerations

### Objective

- Understand the different characteristics that contribute to the usefulness and performance of a model
- Learn techniques to make your model useful by observing some performance metrics

### What makes a model useful?

Before we dive into formulas and technical discussions, it's helpful to ground the discussion with a small example: We need to identify the pirates (classification) and also predict how much they looted (regression) so we can give them a fine.

## Classification

- **Recall:** If our model says everyone is a pirate, we are sure to catch 100% of them. (High false positive, high recall and low precision)
- **Precision:** If we say everyone is a pirate, we are sure to mislabel some innocent citizens, so it's often wrong. (High false positive, high recall and low precision). What if our model only classifies an individual as a pirate if we caught them in the act three times, and we have a confession on tape? It'll be very precise, those our model classifies as a pirate are almost definitely pirate, but we won't catch many pirates. (High false negative, low recall, high precision).

The key question is: what kind of mistake is more acceptable? And relatively, how bad are these mistakes:

1. In a more authoritarian society, it may be more preferable to err on the safe side to catch more pirates and accept the fact that some innocent people will be jailed
2. In a more liberal society, it is unacceptable to mistakenly jail the innocent so it may be more acceptable for a few pirates to roam loose and loot
3. In the real world, it depends on the end goal/business case. The helpful question is to ask, how many ways can we be wrong, and what if we were wrong?
  - **For example (when the cost of false-negative outweighs false positive):** Letting childhood cancer go undiagnosed is bad, especially if you can easily repeat the test to verify a way to rule out a false positive, and it only costs the family some anxiety until the second test comes back negative.
  - **For example (when the cost of false-positive outweighs false negative):** We need only have ten vaccines, and if we give the vaccine to someone who does not need it, someone else might die from a preventable disease.

## Formalizing concepts

The simplest classification is binary. *It either is \_\_\_\_\_ or is not.* There are no other choices. And it's easy to extend a binary classifier into 3+ classes by asking the question, is it class1 or not, is it class2 or not, is it class3 or not, and so on. (One versus all classification). So, we will proceed with binary classification.

### You have a hypothesis/question:

- **Whether someone is a pirate or not?**

The label/or the outcome is valued 1 if the individual is in the pirate class, 0 otherwise.

The null hypothesis is: a given individual is not a pirate. The alternative hypothesis is: this given individual is a pirate.

- **False Positive (type 1 error):**

Our model classified an individual as a pirate, but they were not. (falsely rejected a true null hypothesis).

- **False Negative (type 2 error):**

Our model classified an individual as innocent (not a pirate), but they were. (failed to reject a false null hypothesis).

- **True Positive:**

A person was correctly classified as a pirate

- **True Negative:**

A person was correctly classified as innocent.

To explain the above, we have set this up in a grid, also known as the confusion matrix. Let's take a look at how it functions:

<b>Predicted (v) vs. actual =&gt;</b>	<b>Pirate (1)</b>	<b>Not Pirate or Innocent (0)</b>
Pirate (1)	True Positive (TP)	False Positive (FP)
Innocent (0)	False Negative (FN)	True Negative (TN)

*Table 14.1: Confusion matrix for a single class*

**Precision:** Out of all the items that were labeled 1 by the model ( $TP + FP$ ), how many should be labeled 1?

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

**Recall:** Out of all the items that should be labeled 1 ( $TP+FN$ ), how many were labeled 1 by the model as True Positives:

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

Let's say we have 10 data points/individuals, and here is the break down in results:

Predicted (v) vs. actual =>	Pirate (1)	Not Pirate or Innocent (0)
Pirate (1)	3	1
Innocent (0)	2	4

*Table 14.2: Confusion matrix single class break-down*

Let us see how we can extend the same scenario to the multi-class case? Let us add a class for cops:

Predicted (vertical) vs. actual =>	Pirate	Innocent	Cops
Pirate	3	1	2
Innocent	1	4	3
Cops	2	3	1

*Table 14.3: Confusion matrix for multi-class*

Only the diagonal counts are truly correct.

For example: Compute the precision for the pirate class for the model as follows:

$$Precision = [ TP / (TP + FP) ] = 3 / (3 + 1 + 2)$$

$$Recall = [ TP / (TP + FN) ] = 3 / (3 + 1 + 2)$$

We can use the harmonic average of both to summarize our model performance in 1 metric:

*Formula for f1 score*

$$F_1 = \left( \frac{2}{recall^{-1} + precision^{-1}} \right) = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In production,  $f1$  is particularly useful when there is an imbalance in the data. More specifically, the distribution of population labels is not uniform;

that is, in the pirate versus innocent example, one would expect that the majority of the population will be innocent, and only a few are pirates.

The mechanism of why  $f1$  is appropriate is two folds:

- When there is less data for a certain class, the performance is likely to be inferior.
- When there is a large imbalance, the model is likely to over classify to the majority class (if I took a random citizen and asked you to predict if this citizen is a pirate or innocent, I would guess you would say innocently.)

Therefore, using a (harmonic) average of precision and recall helps keeps the modeler honest.

We can see that the classifier performs worst in the class of cops. Now when there are only a few classes, we can discuss the performance of the model on each class individually. But imagine if we had ten or even 100 different classes, how can we combine the overall accuracy of the model?

We would have to average overall classes.

There are three most common ways to do this:

- Simple (aka macro) average / macro-f1: Average over classes, each class has equal weight
- Weighted average: Each class has a weight corresponding to the distribution in the label
- Micro-average (rarely used in practice): Treat the dataset as one class, this breaks down to being equal to the accuracy

## What metrics should you use (rule of thumb guide)?

In this section, we will discuss the different metrics that we will use in the various machine learning algorithms.

### Classification problems

Here are some of the questions that you might come up with while dealing with several datasets and applying machine learning algorithms on these

datasets:

- **Question:** Is data imbalanced?  
**Answer:** Use f1
- **Question:** Is label binary (you have only two exclusive classes)?  
**Answer:** Use binary entropy
- **Question:** Is data multi-label (you can have multiple classes for the same data point, think tagging)?  
**Answer:** Use binary entropy
- **Question:** Is label multi-class (more than two labels but exclusive)?  
**Answer:** Use categorical cross-entropy

## Regression

Given a model is wrong, how far off was it? In this model, here are some of the pointers that you might come up with while dealing with several datasets and applying machine learning algorithms on these datasets.

The key here is what distance do we want to use to measure *how wrong*.

Some examples of different settings are:

- **Euclidean distance:** Works for most cases
- **L2 distance or square Euclidean distance:** Penalize large errors and ignores small errors
- **Edit distance:** Minimum number of operations to convert one string into another
- **Keyboard distance:** How likely you are to make a certain typo due to *fat finger* errors
- **KL divergence:** How similar are two distributions?

Spearman's correlation or Kendall's tau is distance measures that are used in ranking, (given a ranked list, how far from the true position did our model predict).

## Most commonly used error metrics for regression

Here are some commonly used error metrics to keep in mind while calculating the performance of your regression model:

- **MSE:** Mean squared loss, more sensitive to outliers/extreme errors
- **MAE:** Mean absolute loss, not sensitive to outliers/extreme errors
- **Huber Loss:** In the middle of the two above

The intuition for regression loss functions is best illustrated with a visual.

Using the following visual, we will build up MSE and extend it to the other losses:

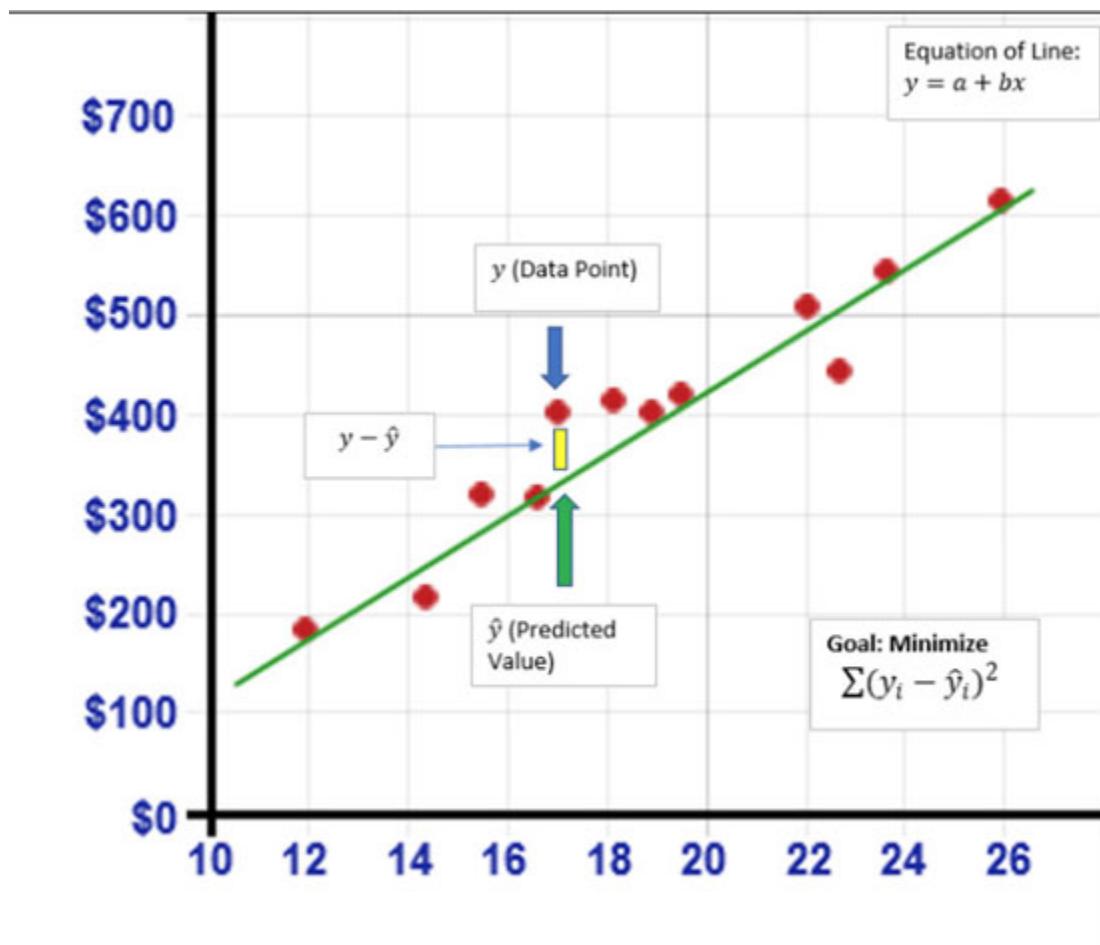


Figure 14.1: Predicting stock prices with linear regression

Explanation of the above figure:

- The green line is our model prediction, called  $y^{\wedge}$  (read *y hat*)
- The red dots are real data points (in this case, data is 2d ( $x,y$ ))

- The distance/error for each data point is  $y(i) - \hat{y}(i)$  where  $i$  is a value of  $x$

We want to get a sense of how our model performs across all our data points. So, naturally, we sum up all our distance/error (Sum (y(i) -  $\hat{y}(i)$ )) for all data points, over  $i = 0$  to a number of data points, that is,  $n$ ).

## But now we notice a problem

Say we only have 2 data points  $(i, j)$ . For data point  $i$ , our model was 1 too high, and for  $j$ , our model was 1 too low. Using the above deduction, we would have an error of zero. So, to avoid this “*cancelling*” out, we use either absolute distance/error (AE), or we square the distance/error (SE).

So now, we have an error metric that reflects the overall distance between what we predicted and the actual value. Still, the error will be bigger if we have more data points (the smallest error we can have is zero; therefore, as we add more data points, the overall error cannot decrease).

How do we compare model performance on datasets of varying lengths?

We can average all the data points we have. So, we can sum  $(y(i) - \hat{y}(i))$  for all data points, over  $i = 0$  to a number of data points, that is,  $n$ , over  $n$ ).

We have derived MSE, whose formula is:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

MAE is another variation that uses absolute distance instead of square distance.

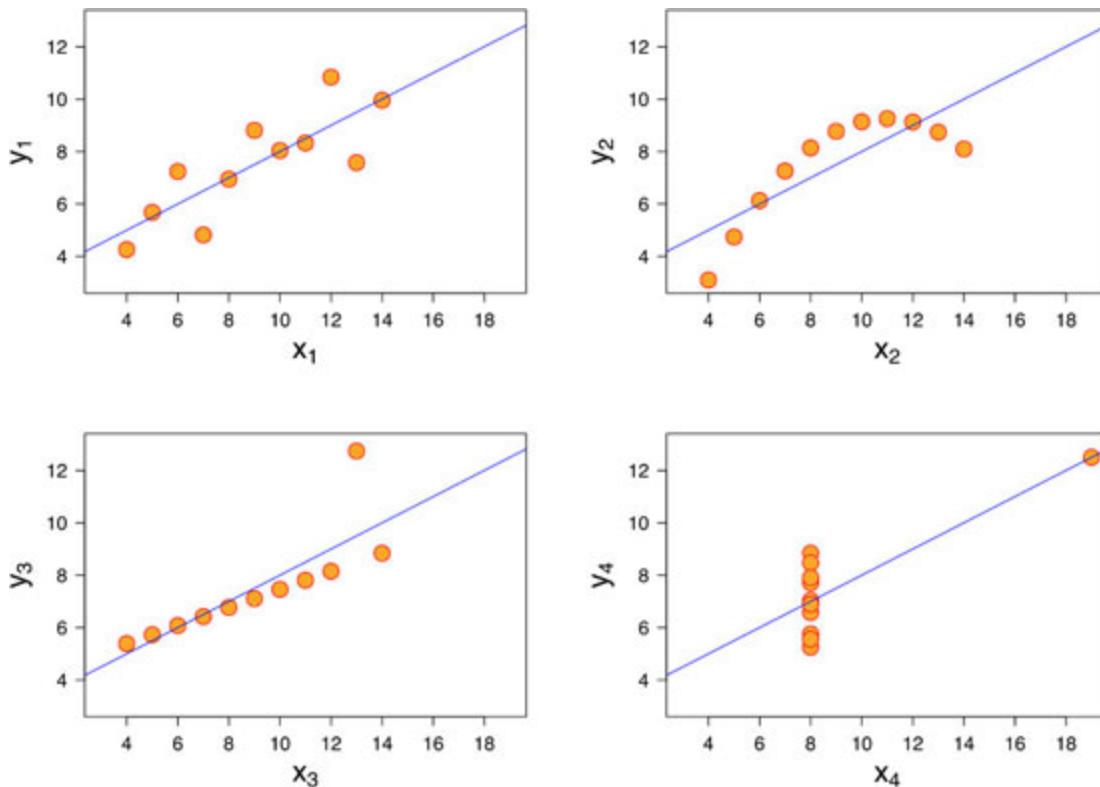
**Huber loss:** Define a hyper parameter delta, for small errors (distance less than delta) it is quadratic, for large errors (values greater than delta), it scales linearly with hence being more robust to extreme errors or outliers:

$$L_\delta = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

Quadratic  
Linear

*Figure 14.2: Huber loss equation*

**Note:** MSE should only be used to compare on the same dataset because it is not scale-independent the same for **RMSE (Root Mean Squared Error)**. While on the subject, R squared is a particularly bad metric (see below: Anscombe's quartet). Two favorite metrics from the forecasting community are **GMRAE (Geometric Mean of Relative Absolute Error)** and Percent Better (against a baseline benchmark or existing model):



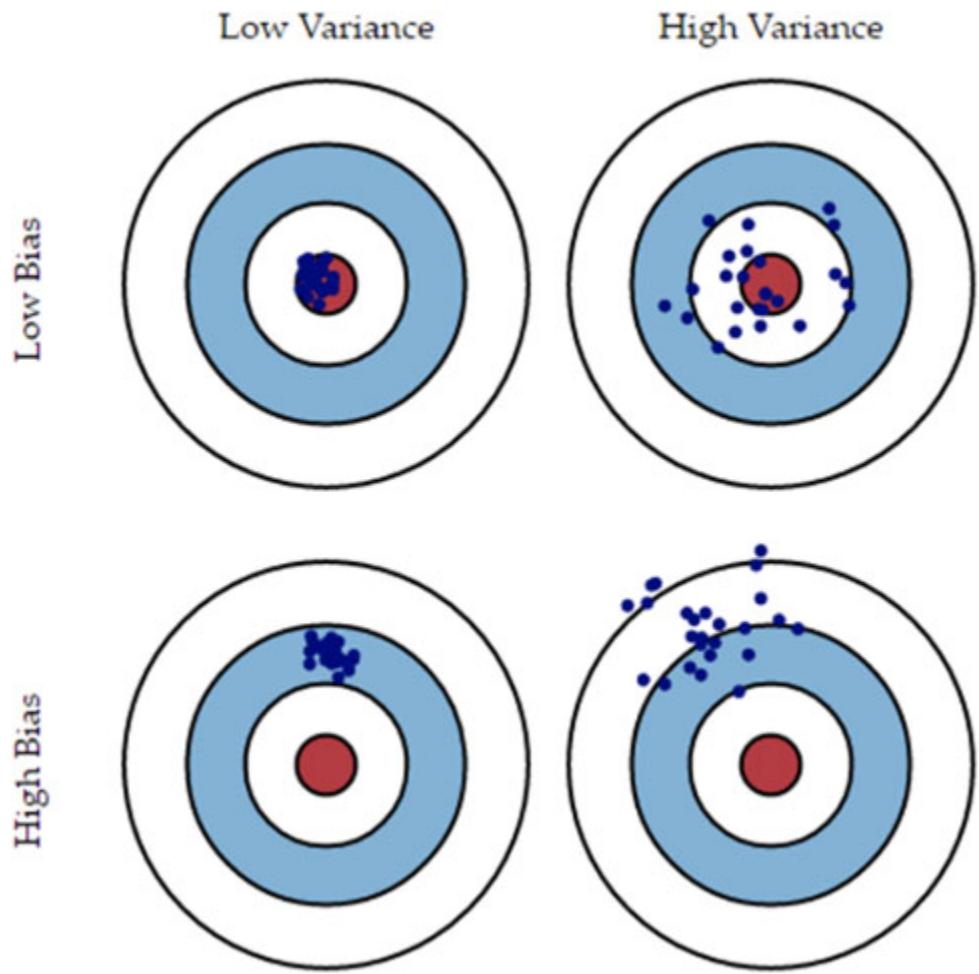
*Figure 14.3: Anscombe's Quartet*

Once a certain distance metric is selected: the model that has the lowest error is the better one.

## **Overfitting, underfitting, bias, and variance**

The performance dimensions overfitting, underfitting, bias, and variance tell us the ability of our model to generalize on unseen population data. The data we have is presumed to be a sample. If we had the data for the whole population, we would not need a model; we could calculate or query whatever attributes we wanted. If the sample is not representative of the population, there is a sample bias, which is defined as: Some elements of the population have a smaller chance of being sampled, which in reality is always the case to some degree, despite our best efforts. Our model built on the sample data may not do well when we apply it to new data from the general population. In some cases, we may not know if the model performs terribly on the new data until later.

Therefore, we would like to select the model that is not only performant on the training data (not underfitting) but also is most likely to generalize well (not overfitting):



*Figure 14.4: Representation of bias versus variance*

	<b>Underfitting</b>	<b>Just right</b>	<b>Overfitting</b>
<b>Symptoms</b>	<ul style="list-style-type: none"> <li>- High training error</li> <li>- Training error close to test error</li> <li>- High bias</li> </ul>	<ul style="list-style-type: none"> <li>- Training error slightly lower than test error</li> </ul>	<ul style="list-style-type: none"> <li>- Low training error</li> <li>- Training error much lower than test error</li> <li>- High variance</li> </ul>
<b>Regression</b>			
<b>Classification</b>			
<b>Deep learning</b>			
<b>Remedies</b>	<ul style="list-style-type: none"> <li>- Complexify model</li> <li>- Add more features</li> <li>- Train longer</li> </ul>		<ul style="list-style-type: none"> <li>- Regularize</li> <li>- Get more data</li> </ul>

*Table 14.4: Underfitting/Overfitting for various ML algorithms*

We want to evaluate our model on both its ability to reflect the training data inadequate details (use a metrics previously discussed) and the ability to generalize on the test set, which we use as a proxy of unseen data from the general population. We want a model that performs ideally equally well on the test set, but in reality, we will settle for the one that performs not much worse.

Given the above data sample, if I were to ask you to make a model that generates the least amount of MSE, you might do so by drawing a curve with many bends, say a polynomial with very high power, essentially increasing the number of parameters in the model, that connects one data point to the next. Instead, if I were to show you ten citizens and tell you that numbers 3 and 4 are the pirates, you could easily remember that and also have no error. But these models are “remembering the training data” or overfitting to the idiosyncrasies of the training data. When this model is

applied to the unseen data, there is a high chance it will perform poorly because no other data point is likely to be exactly like the one in your training set.

Say we only use one parameter to represent our model: the mean or the proportion of the pirate to innocent civilian ratio. Our model will likely perform just as good (more likely, just as insufficiently) on the training set versus the testing set. So, we would need a happy medium where the model has enough complexity and parameters to represent the relevant dimensions of the data well but not so many that it applies distinctive relations to the general population, where relations do not exist.

## Formalizing concepts

In this section, we will reflect on and formalize some of the concepts that we have discussed in the previous sections.

### High bias (underfit)

Models have large errors and low accuracy due to bad assumptions. The model is unable to pick-up/learn the relevant relationships in the data, which are typically evaluated on the training set. If the model is not performing well on the test set, it could also be due to overfitting.

### Variance (overfit)

The model has unreliable performance on unseen data from the population; the model has learned noise in the training data.

These are fundamentally two types of errors that we can and should reduce in the model.

Let's look at the MSE, for example:

$$\text{MSE} \left( f(x), \hat{f}(x) \right) = \mathbb{E}_{\mathcal{D}} \left[ \left( f(x) - \hat{f}(x) \right)^2 \right] = \underbrace{\left( f(x) - \mathbb{E} \left[ \hat{f}(x) \right] \right)^2}_{\text{bias}^2(f(x))} + \underbrace{\mathbb{E} \left[ \left( \hat{f}(x) - \mathbb{E} \left[ \hat{f}(x) \right] \right)^2 \right]}_{\text{var}(\hat{f}(x))}$$

Here are some of the techniques we can implement in practice:

- Use performance measures to narrow down to a small family of functions/models (if your model does not perform well even on the

training set, there is no point in worrying about generalization). To improve performance, you can:

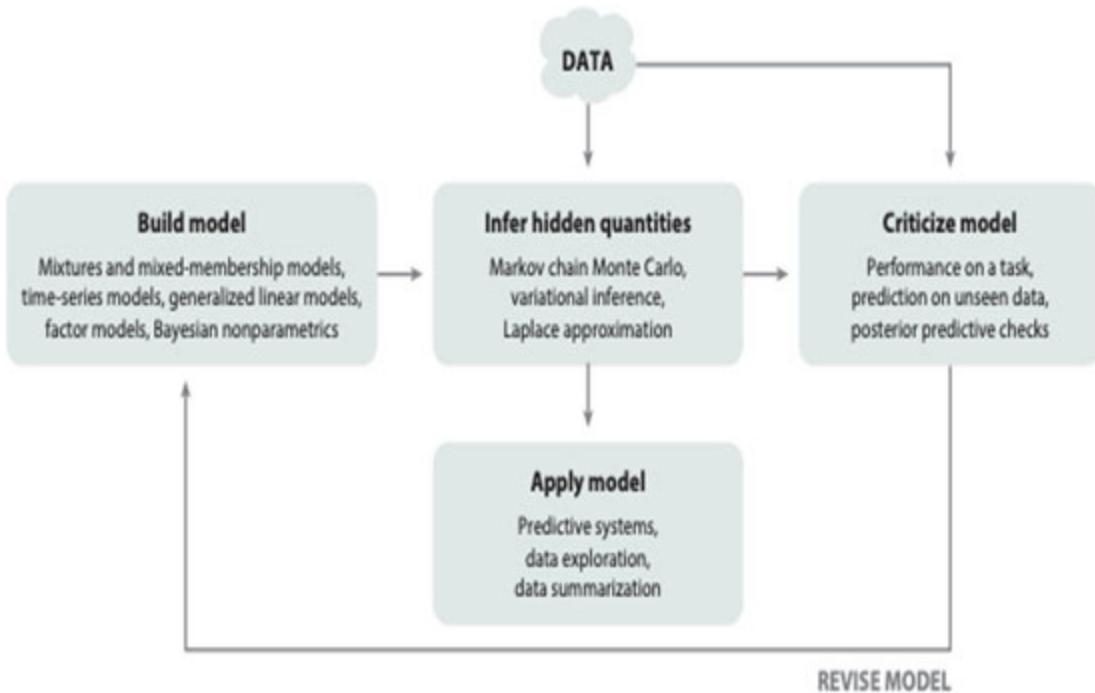
- Try feature engineering, aka doing more in-depth data exploration, such as talking to domain experts, regardless of their knowledge (or lack thereof) surrounding high ROI activity within ML. You may ask them questions like, “if you were asked to perform this task and you can ask 1 question/ask for one piece of information, what would you ask and why, what would you ask next?” (And so on...) You can ask them to use human performance as a benchmark.
- Try getting more data, try getting more informative data (that is, the minority class).
- Try different models or increase model capacity.
- Perform model or hyperparameter tuning to achieve higher performance (via grid search or even evolutionary algorithms)
- Then use regularization techniques and tune hyper-parameters to reduce the difference in performance on the training and test set (regularization techniques vary based on the model you are using, that is, if using regression L1 and L2 regularization is common if tree based = limit tree depth, or limit node splitting criterion if neural networks, dropout or L1/L2 or early stop are common methods)

## **How to make your model more useful?**

In this section, we will learn about the different metrics that will help derive more insights from our model and make it more useful.

Metrics in relation to the model development workflow: Box’s loop

The following figure represents the box’s loop, which is an iterative process to make your models better during data-analysis:



*Figure 14.5: Box's loop*

Here are the steps in box's loop that will help you make a better model:

1. Acknowledge that we approach any problems with a set of questions, prior knowledge, and assumptions that may or may not be productive (or correct).
2. Split the data into sets for training, tuning (of model parameters), and validation/testing (model criticism).
3. Explore the data and find relationships, feature engineering (this is done after splitting the data to reduce the chance of data leak via the modeler, it is considered best practice where possible to never look at the validation data).
4. Combine 2 and 3 into a model (s) and find optimal model parameters [via closed-form solution or approximations, that is, gradient descend).
5. Tune the model(s) hyperparameter using the tuning set [at this point, you may decide to drop some models and tune a smaller set of models you started with]; this is typically done with grid search or occasionally evolutionary algorithms or auto-ML.
6. Evaluate the model(s) on the validation set.

7. Return to Step 3, if needed, repeat until validation criteria (business-driven metrics) are met.
8. Deploy and apply the model (ideally with some mechanism for monitoring performance).

In this process, the metric helps you decide what models to fine-tune and also how you should fine-tune your models.

## **Efficiency**

All other things being equal, models that are fast and memory-efficient are considered more performant. It is similar to the analysis of time and memory complexity in computer science. Furthermore, there may be a trade-off between performance during training and performance during inference.

### **Example scenarios of model efficiency**

- Deep learning models are typically trained or updated with batched data. Once the model has been trained, inference time is relatively insignificant because we simply need to apply the saved weights, that is, the object classification in a self-driving car needs to be fast during inference (so you do not mistake a pet cat crossing the street for a *balloon*), we would rather have a model that takes longer to train but classifies quickly for the same level of accuracy.
- Bandit models are often trained online, with streaming data, and if the model takes a long time to train, it will lead to a bottleneck and also bad inference outcome.

## **Other practical considerations**

Let us look at some practical considerations that will contribute to the performance of a model.

### **Data needed**

The amount of data required roughly scales exponentially with the number of dimensions or loose parameters if most parameters are independent (the

curse of dimensionality).

## Parametric models versus non-parametric models

Let us take a quick look at the difference between parametric and non-parametric models.

Parametric models: assumes there is a finite set of parameters, given the parameters (and model) the future predictions are independent of observed data.  $P(x| \text{params}, D) = P(x| \text{params})$ , and therefore, the parameters fully represent the relationships in the data. It makes the model complexity bounded and scales well even if the amount of data is large or unbounded. These models have less flexibility, and therefore your model may not represent the data if you assumed the wrong set of params (model capacity) (think linear regression).

**Non-parametric model:** Data/or data generating distribution cannot be defined/represented by a finite set of parameters. These models are more adaptable but scale less well with more data (think K-means or Spline Regression).

## Transparency and interpretability

Transparency and interpretability are particularly important in high-risk situations with many other stakeholders (that is, medical diagnostics with regulatory stakeholders). A model can be highly performant, but you may not get it deployed if the results cannot be explained or *trusted*. Models like neural networks frequently suffer from this, which is why regression and tree models are more frequently deployed despite being less accurate.

## Robustness

Robustness is how well the model handles noise, or the performance does not suffer much given a small perturbation of parameters. It is similar in concept to generalization.

## Flexibility

In this section, let's refer to the non-parametric models mentioned above. In practice, we would like not to have to retrain a new model when the underlying data distribution shifts, or if new classification categories are added.

## Conclusion

In this chapter, we have explored the various performance metrics that will help make our model much better and useful. We have analyzed the role of data behavior like underfitting, overfitting, variance, and bias to create a better performing data model.

In the upcoming chapter, we will look at design thinking when it comes to designing a machine learning model and what are the different steps to consider while building an end-to-end solution starting from data collection to model deployment.

## Reference links:

- *Figure* 14.1 Source:  
<https://programmingforfinance.com/2018/01/predicting-stock-prices-with-linear-regression/>
- *Figure* 14.2 source:  
<https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/>
- *Figure* 14.3 source:  
[https://en.wikipedia.org/wiki/Anscombe%27s\\_quartet#/media/File:Anscombe's\\_quartet\\_3.svg](https://en.wikipedia.org/wiki/Anscombe%27s_quartet#/media/File:Anscombe's_quartet_3.svg)
- *Table* 14.4 source:  
<https://br.pinterest.com/pin/672232681855858689/>
- Variance Formula source:  
<https://daviddalpiaz.github.io/r4sl/biasvariance-tradeoff.html>
- *Figure* 14.5 source:  
<http://www.cs.columbia.edu/~blei/fogm/2019F/readings/Blei2014.pdf>

# CHAPTER 15

## Design Thinking for ML

### Introduction

The field of software engineering and design, also known sometimes as product management are relatively mature. Yet almost all of the existing toolsets and processes for software engineering and design do not work via a quick "lift and apply." To ensure that we are building quality products, it is important to think about ML from a design-thinking perspective. As ML is rapidly involving, it helps create more innovation in this space and structure a solution in an easy manner.

Design thinking is a problem-solving framework which ensures that solutions are created with empathy, optimism and integrative thinking.

### Structure

- Answering some important questions in the field of product development
- Introduction to Design Thinking
- Steps to follow in the design thinking process
- Practical advice for ML prototyping

### Objectives

- The primary objective of this chapter is to explore the process of design thinking
- Understand the different ways to build an AI solution, keeping in mind the various parameters of the design

## Answering some important questions in the field of product development

Several approaches are being taken today, in various industries, to design a product and to create a framework that would help everyone grow together while creating solutions. Let us look at answering some important questions in the field of design.

### What is agile?

There have been several different ways of product designing in the software industry. Starting from following the waterfall model to following the agile model nowadays, we have come a long way in the way we think about working on creating products.

Many of the same principles apply in the field of machine learning as well. Agile is a software development approach where it primarily focuses on collaborative development through self-organizing and creating cross-functional teams.

### How well do estimates work in ML?

One can perhaps estimate how long it will take for a model to converge but is that result going to be good enough for the feature to ship?

The high level of uncertainty causes frictions in teams because ML tends to view the "standard process" as more overhead than value add (and they are frequently correct). Research also tend to resent product from placing constraints on how much time/freedom they have to explore, and the performance of the final solution.

Engineering/Product tend to find that it's hard to measure productivity/velocity of machine learning. The key contribution only took 50 lines of code, but it took reading five papers and a couple of hundred GPU hrs to get there. It is even harder to plan around timelines.

### What is a design sprint?

A highly successful framework, Google ventures, popularized to build prototypes in 1 week. On prototype day (Thursday in the "sprint week")

likely, ML does not have enough, or the right data to prototype models or de-risk.

Or worse, the rest of the team started to work on something that is only feasible as a product if ML produces an infeasible model.

Yes, you can hack it for a while with "human in the loop" or further develop the product "Wizard of OZ" style, but inevitably the team ends up waiting for ML to ship their model to make the product work. A key selling point of ML is the ability to reduce costs via automation, right?

## Why is there a chapter on design for ML?

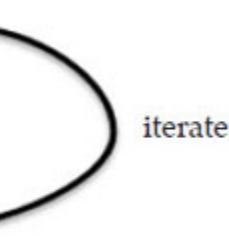
Helping ML professionals think more holistically, more like a designer is potentially a faster way to bridge the gap than having the engineering/design teams go through this book. Furthermore, because of the potential impact of the technology we work on, we must be conscientious about "**doing the right thing**".

We hope to summarize key ideas from the field of design and agile to help ML professionals work on things that have an impact. Even if you are an academic, working on research that benefit end-users helps with funding.

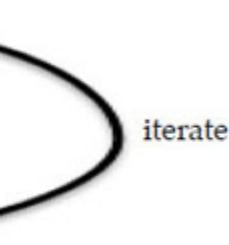
## Introduction to design thinking

The base framework we will discuss and adapt is called "Design Thinking" developed by D School at Stanford (<https://dschool.stanford.edu/resources/design-thinking-bootleg>). There are many versions, and you should develop your own.

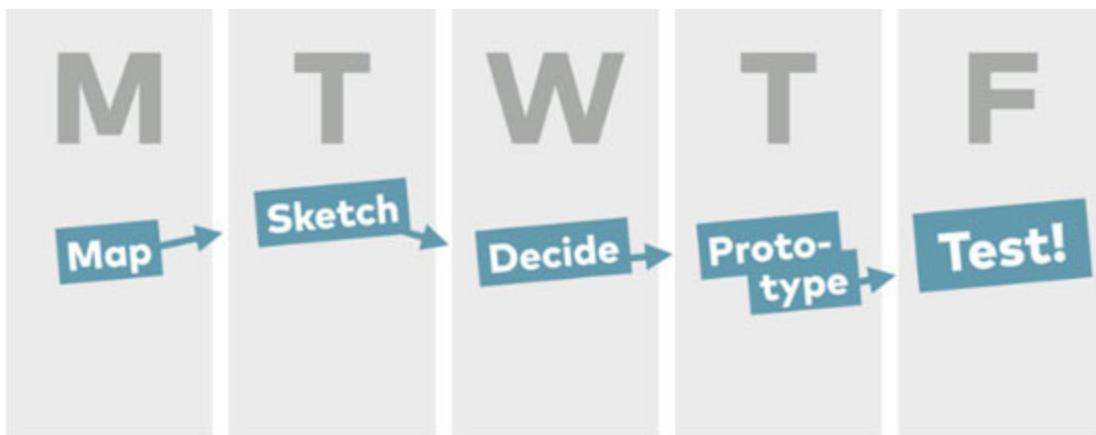
The main stages in design thinking are:

- Empathize
  - Define
  - Ideate
  - Prototype
  - Test
- 
- The diagram shows a horizontal sequence of five stages: Empathize, Define, Ideate, Prototype, and Test. A thick black curved arrow starts at the end of 'Test' and loops back to the beginning of 'Empathize', labeled 'iterate'.

For reference, the sprint method is:

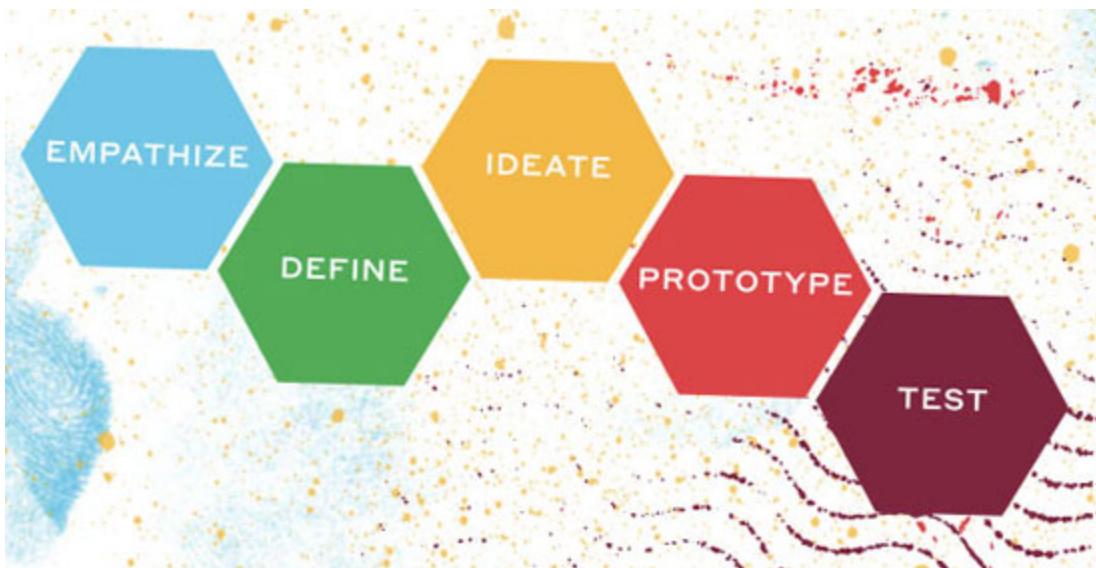
- Map
  - Sketch
  - Decide
  - Prototype
  - Test
- 
- The diagram shows a horizontal sequence of five stages: Map, Sketch, Decide, Prototype, and Test. A thick black curved arrow starts at the end of 'Test' and loops back to the beginning of 'Map', labeled 'iterate'.

Here is an image that depicts the sprint method:



*Figure 15.1: Sprint method*

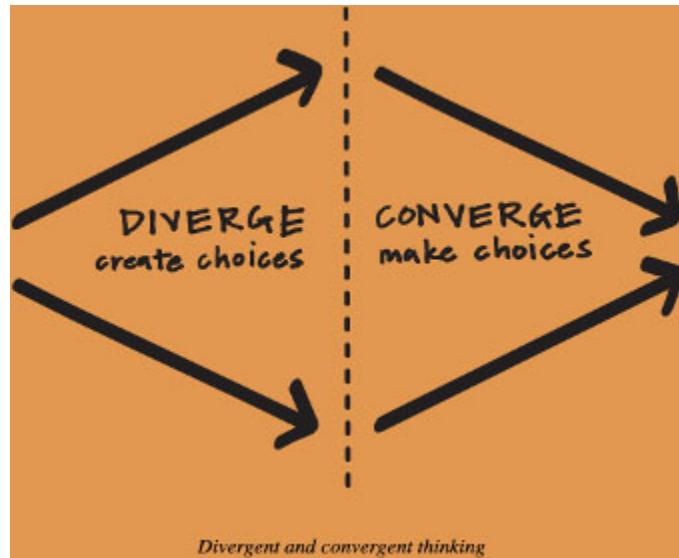
Here is another image that shows us how a sprint should be tackled:



*Figure 15.2: Sprint methodology*

Here are the different observations from the sprint principles, translated into the ML domain:

- All of these start with developing a deep understanding of the problem (empathize, map)
- All of them have a diverge and converge (explore => exploit) pattern where the ideate or the decide step is the turning point. Here is an image that depicts the diverge and converge pattern:



*Figure 15.3: Design thinking today*

- All of them have a small portion allocated to the prototype stage ( $\frac{1}{5}$  steps,  $\frac{1}{5}$  days for design sprint)
- And all of them end with test and iterate

## Why does design thinking matter?

- To succeed as an ML professional, we must understand the data; to succeed in a larger scope, to contribute to a successful machine learning-based product, we must understand the problem in all its intricacies.
- Premature optimization leads to regret. We all have our favorite techniques or approaches. There is a tendency for Machine Learners and Data Scientists to gravitate towards complicated solutions, because of their inherent curiosity and because it would be impressive. But these are commonly at odds with project or product success.
- It's important to approach with an open mind and explore other techniques, sometimes simplistic ones so that we have a chance to do a fair evaluation. Then it's much more likely we will narrow down to the highest ROI approaches. And we design experiments in a way such that if our initial plan fails, we can quickly assimilate learnings and move to viable alternatives. It works for getting published too.

- Why should you limit your build time? How do artificial constraints help?

Models will train forever as long as you can afford to pay for compute. The search space for optimal parameters is infinite.

- Testing is fairly well covered in machine learning, but testing needs to go beyond doing a train-test split. What do you need to consider for the model to generalize it in the real world? If things go badly and your name and the model get mentioned in the papers, what probably went wrong? How can you prevent it before it happens?

Let's look at the next section, which will help answer some of the above-stated questions by showing us the process of design thinking in a detailed manner.

## Design thinking in select details

A simple way to think about design thinking is to follow the below-mentioned flow:

- Observe
- Engage
- Immerse

Try to go with your product team when they make a discovery or meet with clients. Then you will have the opportunity to do the above.

- **Observe** the users or the human you are hoping to assist or replace in their natural context.
- **Engage** and ask domain experts about the tips and tricks that lead them to build their expertise.
- **Immerse** and try to do what they do or if not possible, shadow them closely.

It is an excellent time to do some benchmarking or feature engineering where you can ask questions such as:

**Q:** "On average, what is your speed or accuracy on this task?"

If your model is slower and less accurate then hopefully it costs less than the human's salary

**Q:** "What are some hacks you have learned that you would tell a friend who is new to the job"?

These are potentially important features

Let's say we have some usual data or if I gave you this, which is a representative set of data, what would be the first thing you look for? And why? What about the next thing? Iterate these questions over and over again to get the right answers.

You understand the human's workflow and also making a list of key features in descending order of importance, are the humans, always right? No, but this is a good place to "warm start".

## **Questions during the process of design thinking**

In this section, we will explore some questions that come to our mind during the process of design thinking and what can we gain by answering these questions. Some of these questions will throw insight into building better models or solutions for our product.

- What is the most annoying or difficult kind of situation? How long does it take, and what is your typical accuracy?"
  - You are finding the highest values add aspects of this project
  - In most cases, by making the user's life easier / helping them do the things that they are unhappy doing, this will be a key success measure for ML products... If you can't get any users => no data => can't improvement => project likely goes nowhere. Often user/ internal adoption is a key metric for decision-makers when evaluating a pilot solution.
  - As much as ML / Data Scientists / AI hungry executives would love for ML to replace humans, it's rather unlikely that it will happen in 1 step. There is typically a human in the loop stage if for nothing else than to build trust. (Run the two systems in parallel or have the humans check/correct the result if you can't get people to use your product, it'll be very hard to proceed).

- What kinds of errors should I watch out for in the data, what parts of the data are useless?
  - You are learning about the pre-processing steps and also building a test suite to catch potential problems

Lastly, you should maintain good relationships with your human compadres, especially on the client's side. It's highly likely and advised that you sit down with them and go over the dataset in details after you have done your first round of EDA (exploratory data analysis). It allows you to proceed without any bad assumptions.

## A quick peek into the challenges in current ML tools and technologies

Implementing ML solutions have their fair share of challenges, and when we think about design, we should also keep some of these challenges in mind and design a holistic solution.

Listed below are some challenges to keep in mind while designing an ML-based solution:

- For your algorithm to yield the best results, you need to ensure that there is sufficient amount of training data available. Sometimes having relevant training data which is big enough can pose a big challenge to designing a solution.
- It is important to understand that the results given by your model can be explained. It is also known as Model Explainability and can be a fairly tough challenge to solve.
- If your model performs well on the training data, it might not necessarily perform the same way on the test data. It is one of the biggest challenges to keep in mind, and you need to create a design in such a way that you have a solution to fall back to.

It would be essential to keep in mind that ML technologies have their evolution journey, and there is a constant evolution happening in this domain.

## Steps to follow in the design thinking process

There is no defined set of steps to follow in design thinking as every product and every project is different from each other. But here are some generic steps that can help you get started.

### Step 1 – Define

*"What is the problem we are trying to solve really?"*

A good definition of the problem:

- Incorporates your learnings and understanding of the problem and all the relevant stakeholders
- Has a key insight (a hypothesis/thesis/typically came from a "surprise")
- Uses strong and clear language
- Generates lots of possibilities (we explore first and exploit later)

Here is an example of defining the result in a wrong way:

"We want a high accuracy model that eliminates the headcount for XYZ department by half":

- **Does not incorporate learnings:** why half, why this department, why is accuracy the thing that matters? Any speed limits? Why does headcount matter?
- **There is no insight:** Almost any business executive would agree that that would be good. Why is specifically the goal for this project? What is your hypothesis as to why this problem exists?
- **Clear and strong language:** this does not get the team/customer excited—it is also not clear what high accuracy means
- **Generates possibilities:** this statement is both vague enough not to be inspiring/actionable but also so restrictive in the expected outcome that makes it hard to ideate in the next step. (How would I know if my model will help cut the headcount?... what if we managed 48% reduction?)

Let us look at an example of how we can illustrate the process of design thinking in real-time.

Help Edna, a freelancer, achieve financial sustainability by reducing an overdraft fee (the problem clearly states who it is about, and why it is important) because 80% of them experience stress from living paycheck to paycheck due to the feast and famine cycle and finances are not top of mind (incorporates learning and key insight). This solution also generates possibilities because it could help them keep finances "top of mind" by showing them their statements at a convenient time, helping them self-educate (recommender systems), predicting potential overdraft transactions and blocking it, or using time series predictions to remind them that they are likely to hit overdraft this month, and so on.

It is typically done with the team. It ensures you end up with a problem statement that helps you make progress on the ML front.

## Step 2 – Ideate

Go wide and explore, diversity of ideas is key.

The goal is to generate as many ideas that span as much of the solution space as possible:

- It is important to evaluate ideas at a later stage, do not judge ideas, treat all ideas equally.
- Do not start discussing implementation or its feasibility or how to approach it, your time is better spent generating more ideas.
- Do not get rabbit holed into iterating on the same base idea, random forest vs a mixture of experts, instead, focus on spanning the potential solution space.
- Reward different or out of the box ideas—different ideas are more effortful to think of and have a higher potential for outsized ROI.
- Having a diverse team in terms of background, area of expertise, demographics increases your *solution space*.

Watch out for things that prevent ideas from being fairly shared and evaluated, avoid groupthink. It is often done asynchronously and anonymously, and ideas are evaluated in a different step. You can take a

look at the ***Sketch and Decide*** approach in the design sprint for process details that help with this aspect.

In parallel to or after flushing out the product ideas, you may find that you need to do ML specific ideation and research to come up with ML specific approaches.

## Step 3 – Prototype

What is the fastest, easiest, lowest-risk thing that will help us get data and learn?

The answer is building a prototype.

Why limit prototype time constraints?

If there are no constraints, it's easy to say, let's kick-off a batch of 10 models run them over the weekend. Then you may come back to discover none of your models did well because of a data cleaning problem, or you may go down the rabbit hole.

With time and budget constraints, you default to simpler, faster and more interpretable models and also force yourself to ask the right questions:

- Which one is likely to overfit?
- Do I have enough data to train a deep RNN?
- What parameters are most important to perform grid search on?

The prototype to de-risk and decide:

There is likely key information that prevents you from moving forward on prototyping, for example:

- This model worked on images; does it work on frames from a video?
- Is it possible to do successful transfer learning with the small amount of data we have? If we use a deep LSTM, how bad is the overfitting?
- Are gradients being propagated sufficiently, if not does a CNN capture enough sequential relations for this task?
- Is the memory efficiency of this model going to work on this edge device?
- Is network latency going to prevent successful parallelization?

At this point, you have what you might call a decision tree as to what approaches are high ROI, and you should try first. But there is key data missing so you can't make use of this decision tree.

A definition of ROI in the ML setting is roughly defined as:

- Expected lift in metrics \* Expected probability it will work / How much time it takes to get it to work

You should set up small experiments to gain data or de-risk. Or do grid-search along one dimension (that is, how training data affects transfer learning).

In general, it's highly advisable to have a de-risking road-map laid out in the order of lowest risk, easiest to implement, and delivers immediate or incremental value to the users.

An example might look like:

- **If you have to ship tomorrow:** Use the average, call an existing API
- **If you have to ship at the end of the week:** Use moving average or simple regression, expected accuracy ~65%
- **If you have a month or 2:** Try a benchmark set of models (Neural networks, Random Forest, ARIMA etc.) and tune them, expected accuracy: 72%
- **Try learned transfer embeddings:** Expected accuracy improvement 4%
- **You bought yourself and your team a lot more time:** Do a literature review and try newer approaches. At this stage, try papers with code or at least provided detailed training procedures and description of the data first or find already implemented versions to start with. In my experience, this has been a "key feature" successfully borrowing research for production.

You might be surprised how many commercially successful projects never need to move beyond the first two stages. Most of the deployed ML models are Random Forest (that is, GBDT) or Regression (Logistic Regression, Linear Regression).

The beauty of having a de-risk ordered road map is that you build trust with your team and customers, and you buy yourself more time to work on the

impressive solutions with low stress. It's fine to say: *I tried this cool architecture, but it didn't work, you will have to deploy our previous random forest model, which the client is happy with. It's a lot more stressful when you say: I spent the last three weeks trying to get this to work, and I did not succeed... I will start working on a hack now, and hopefully, it'll be ready by the end of the week. In your next project, your PM will be checking in with you a lot more if you do the latter.*

Always remember, a successful prototype is not a solution with the highest accuracy, but a solution that adds value to the business.

## Step 4 – Test

There are four clusters of questions to consider when evaluating models.

First, ask "do you really need a better model"?

Use reasonable alternatives and compare them. The status quo (current model) and the simplest model (like the average) should be a part of this set.

The 4 clusters of questions include:

- **Testing assumptions** (The wrong assumption may prevent your model from learning)
- **Testing your data and methods** (Garbage in, garbage out)
- **Replicating outputs** (Were there easy mistakes? how robust and generalizable is the result)
- **Assessing outputs** (define what matters, how to measure better—this was covered fairly extensively in the performance evaluation chapter)

Let us look at each of these clusters of questions in the upcoming sections.

### **Testing assumptions:**

- **Use objective test of assumptions:** Use objective data to test assumptions, if not possible, verify with inside and outside subject matter experts.
- **Test assumptions for construct validity:** Trying different approaches to estimate or validate your assumptions should provide similar results.

- **Describe conditions of the problem:** Define a bounding box for how much or when the model is expected to generalize.
- **Tailor analysis to the decision:** Don't do useless work. Know what range of results will change decisions and grow your customer base.

### **Test data and methods:**

- **Describe potential bias:** The ML Engineer/Data Scientist should be able to list and disclose their influences and bias. Not disclosing is a red flag and unbiased predictions are empirically better.
- **Disclose details of the methods:** Help others help you catch problems and improve, noting deficiencies in methods are correlated with better predictions.
- **Verify the users understand the methods:** Ideally, the consumers of the model or results should be able to explain the methods to others like them.
- **Replication:** Cross-validation is a good way to squeeze more value out of your limited and valuable data.

### **Assess outputs:**

- **Examine all important criteria:** Examine all criteria from all stakeholders. One criterion may not be enough. If multiple criteria, then what is the trade-off?
- **Prespecify criteria:** Prevents hind-sight bias, prevents reward hacking.
- **Assess face validity:** Ask human domain expert if forecast looks reasonable.
- **Choosing the right error measures:** As discussed in performance evaluation.

### **Other questions to consider at this stage:**

- How do the stakeholders view model performance?
- What other criterion did we fail or satisfy (that is, data security, privacy, telemetry around model performance, transparency, interpretability, and so on)

- When does the model break? (What if we 10x, 100x, 1000x a certain attribute, that is, stream rate?)
- When does it underperform?
- When is it more certain than it should be?
- Do the model assumptions still align with our assumptions about the problem?
- Whose buy-in do we need, who are the stakeholders, did we miss anyone initially
- Are there unrealistic understandings internally from product or leadership and externally from our stakeholders or customers that may derail the project?

At this stage, a diverse sample of five representative end-users will likely provide sufficient and directional data on the viability, areas of improvement and satisfaction with the prototype solution. It may not be a surprise where you should be at, and if that's impossible at least review the recorded feedback session.

Questions that are likely to be informative at this stage include:

Fill in the blank:

- I like that \_\_\_\_\_
- I wish that \_\_\_\_\_
- What if \_\_\_\_\_

It is also important to be aware of the ego and other people's desire to protect your ego.

It's important not to ask leading questions or questions that can be answered with a yes/no.

If the user provides a certain feedback, feel free to ask why repeatedly until you find the root cause. And when it gets too abstract—"because I want to be a good person" you can go from the abstract to concrete again by asking how. It is sometimes called the "why-how" ladder.

## Practical advice for ML prototyping

Now that we are aware of the process of design thinking let's look at some tips and tricks which can help create better prototypes.

Use a trained model and transfer learn wherever possible:

- Especially relevant for computer vision or language models
- Meta-learning sometimes has similar benefits but is less validated or proven as a technique

Always use a very small dataset and overfit first:

- Make sure your data is properly sampled and is representative of the overall dataset and hopefully population.
- It quickly reveals which models are likely to be the top performers, what data cleaning issues you might still need to deal with, how hard the learning task is, and if your models are under-capacitated or won't learn the relevant relations.
- It also ensures you have a well-established and vetted pipeline and functions to evaluate your model. You do not want to be spending hours on a DNN when the problem is your pipeline.

Look at the data!

- By look, I mean visualize. Visualize the loss, visualize the weights in the layers, visualize the examples your model consistently gets wrong; this will help you understand how to improve your model.
- You might consider sharing this finding with human experts to see if this is surprising and see if they have insights as to how to improve.

Track your hypothesis and test one variable at a time:

- At the end of an ML project, it's common to find that your experiments follow a tree structure:
  - Major hypothesis/approaches/architecture
    - Hyperparameters for each
    - Training procedure differences
- Keep notes that help you leverage failed experiments at every point in the tree, have a good model or experiment versioning system.

- If you change multiple parameters at once, it's much harder to tell or build intuition around what is likely the cause of change in performance, and how to improve further.
- Even for each set of parameters, you may want to parallelize multiple training runs before you base a new set of experiments on that finding. If the improvement is due to stochasticity, then you will waste a lot of time.

Tricks to improve your model:

- Always reduce bias first, then correct for overfitting
- What helps reduce bias AND overfitting?
  - More data
  - Better architecture and hyper-parameter search
- What helps reduce bias?
  - A bigger model with more capacity or parameters
  - Train for longer or use a better optimizer
- What helps with generalization?
  - Regularization
  - Data augmentation and training on synthetic data

## Conclusion

In this chapter, we have studied about the design thinking process for ML-based projects. It is also important to keep in mind that for sustainable AI, the empathetic design will always have the upper hand over technical design.

We studied about the different stages of design thinking, such as to define, ideate, prototype and test. Each of these stages adds value to the process of building an end-to-end machine learning solution.

In the next chapter, we will take a look at an end-to-end machine learning case study to conclude our book.

## Quiz

1. Why is prototyping an important stage in the design thinking process?
2. Is it necessary to have a time constraint for building a prototype?
3. Consider an ML use case and describe some of the testing assumptions present in that use case.
4. Apart from the usual ML criteria, what are the other criteria that we should keep in mind while designing our product, elaborate on one of them with an example?
5. Who/what is the center of a design thinking model?

## References

- *Figure 15.1* reference: <https://www.thesprintbook.com/how>
- *Figure 15.2* source reference: [https://static1.squarespace.com/static/57c6b79629687fde090a0fdd/t/5b19b2f2aa4a99e99b26b6bb/1528410876119/dschool\\_bootleg\\_deck\\_2018\\_final\\_sm+282%29.pdf](https://static1.squarespace.com/static/57c6b79629687fde090a0fdd/t/5b19b2f2aa4a99e99b26b6bb/1528410876119/dschool_bootleg_deck_2018_final_sm+282%29.pdf)
- *Figure 15.3* image reference: <https://designthinking.ideo.com/#design-thinking-today>

## Resources

Here is a list of recommended and free resources to learn more about design thinking:

- <https://www.thesprintbook.com/how>
- <https://dschool.stanford.edu/resources/design-thinking-bootleg>
- <https://www.gv.com/sprint/>

# CHAPTER 16

## Case Study for Machine Learning

### Introduction

Solving case studies is a great way to understand a concept end-to-end. In this chapter, we will pick up once the case study and look at how we can design the solution for a particular problem using the principles of Machine Learning. We will explore the different steps of creating a simple prototype solution—starting from data collection to ML modelling.

### Structure

- Machine learning case study overview
- End to end python code walkthrough
- Case study summary
- Conclusion

### Objective

- Walkthrough a machine learning case study end-to-end and understand how to work with data
- Answer some important questions when it comes to a machine learning-based problem statement

### Machine Learning Case Study

There are several case studies in the machine learning field based on the different industries and the use cases that are present in these industries. Here are some generic questions that we need to think about before working on an ML problem:

- What problem does this solve?

- Who is the target audience/client?
- What are the existing solutions present today?
- How and where is the input data being collected?
- What is the required output, and how should the output be structured? (a report, a document, pie charts, and so on)

In our case study, we will be answering some of these questions and look at how we can use our data to solve a real-time issue. Our solution could be an innovative approach to an existing model, or it could be a solution to contain/eliminate a problem in the business.

## Overview

In today's day and era, there are a lot of applications which help transfer money from one entity to another. Mobile banking applications, UPI applications and several other finance-related applications are all currently residing on our smartphone. There are not too many open datasets available for financial payments, especially in the mobile money transactions domain.

Financially transactions are confidential, and that's why we do not have too many datasets publicly available. Our case study is based on a synthetic dataset generated using a simulator called PaySim. It uses data from a private dataset to generate a synthetic dataset that replicates the normal operation of transactions and injects some improper/incorrect data to evaluate the performance of fraud detection methods.

In our case study, we will go through this dataset and explore the fraud prediction.

## About the data

Paysim synthetic dataset of mobile money transactions. Each step represents an hour of simulation. This dataset is scaled down 1/4 of the original dataset which is presented in the paper "PaySim: A financial mobile money simulator for fraud detection".

Here are the attributes of the dataset:

- **step**: Maps a unit of time in the real world. In this case, 1 step is 1 hour.

- `type`: CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER
- `amount`: The amount of the transaction in local currency
- `nameOrig`: A customer who started the transaction
- `oldbalanceOrg`: Initial balance before the transaction
- `newbalanceOrig`: Customer's balance after the transaction.
- `nameDest`: recipient ID of the transaction.
- `oldbalanceDest`: Initial recipient balance before the transaction.
- `newbalanceDest`: Recipient's balance after the transaction.
- `isFraud`: Identifies a fraudulent transaction (1) and non-fraudulent (0).
- `isFlaggedFraud`: Flags illegal attempts to transfer more than 200.000 in a single transaction.

In the next section, we will look at the step-by-step Python code that will perform the regression analysis on our chosen dataset.

## Python code and step-by-step regression analysis

This case study is similar to our case study - I but uses a different dataset with different types of attributes. The step-by-step regression analysis will help us understand the sequence of different actions that need to be performed when a raw dataset is received for the first time.

**Section I:** In this section, we will look at the **EDA (Exploratory Data Analysis)** and Visualisation aspects of a dataset.

1. The first step is to import all the required machine learning libraries for this particular regression analysis:

```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
```

```

from sklearn.model_selection import train_test_split,
learning_curve
from sklearn.metrics import average_precision_score
from xgboost.sklearn import XGBClassifier
from xgboost import plot_importance, to_graphviz

```

2. Read the data (which is stored in the CSV format) using python-pandas and store it as a DataFrame. While reading the data, you must mention the location where the dataset is stored.

The dataset to be used is `finpayments.csv`. I have stored the datasets under the `/UseCase` folder under the `/Datasets` folder on my local system:

```
df = pd.read_csv("financial_payments.csv")
```

3. You can now examine the rows and columns of the dataset by printing the head of the DataFrame:

```
df.head()
```

The following image captures the output of `df.head()`:

step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0

*Figure 16.1: Output of df.head()*

In the next step, we will look at the `describe` function.

4. Analyze the data by looking at the data types, and if the column observations are numerical, you can use the `describe` function to understand more about the data. You could also store the column names in an array to avoid typing them in the future if the column names are lengthy.

The following code snippet captures the output of `df.describe()`, which is used to describe the statistical parameters related to a dataset:

```
df.describe()
```

The following image captures the output of `df.describe()`:

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
count	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06
mean	2.433972e+02	1.798619e+05	8.338831e+05	8.551137e+05	1.100702e+06	1.224996e+06	1.290820e-03	2.514687e-06
std	1.423320e+02	6.038582e+05	2.888243e+06	2.924049e+06	3.399180e+06	3.674129e+06	3.590480e-02	1.585775e-03
min	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	1.560000e+02	1.338957e+04	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
50%	2.390000e+02	7.487194e+04	1.420800e+04	0.000000e+00	1.327057e+05	2.146614e+05	0.000000e+00	0.000000e+00
75%	3.350000e+02	2.087215e+05	1.073152e+05	1.442584e+05	9.430367e+05	1.111909e+06	0.000000e+00	0.000000e+00
max	7.430000e+02	9.244552e+07	5.968504e+07	4.958504e+07	3.560159e+08	3.561793e+08	1.000000e+00	1.000000e+00

**Figure 16.2:** Output of `df.describe()`

In the next step, we will look at the null values in the dataset.

- As the next step, we need to see if there are any null values in the dataset. From the output, we can see that there are no null values in the dataset.

```
df.isnull().values.any()
```

### Output:

False

- Let's look at the different types of payments now. We use the `value_counts()` function to perform the operation:

```
df.type.value_counts()
```

### Output:

```
CASH_OUT      2237500
PAYMENT       2151495
CASH_IN        1399284
TRANSFER       532909
DEBIT          41432
Name: type, dtype: int64
```

We can observe that amongst the five types of transactions, `TRANSFER` and `CASH_OUT` are the two primary modes where fraud is likely to occur.

- Let's look at the above two types of payments now. Using the `len` function, let's examine the number of transfers and `cash_outs` that are equal to 1.

```

dfFraudTransfer = df.loc[(df.isFraud == 1) & (df.type ==
'TRANSFER')]

dfFraudCashout = df.loc[(df.isFraud == 1) & (df.type ==
'CASH_OUT')]

print(len(dfFraudTransfer)): 4097
print(len(dfFraudCashout)): 4116

```

8. There are two flags in our dataset that stand out—`isFraud` and `isFlaggedFraud`. Let's look at some interesting data that shows us as to which transactions are fraud and which of them have been flagged fraud due to the system's understanding.

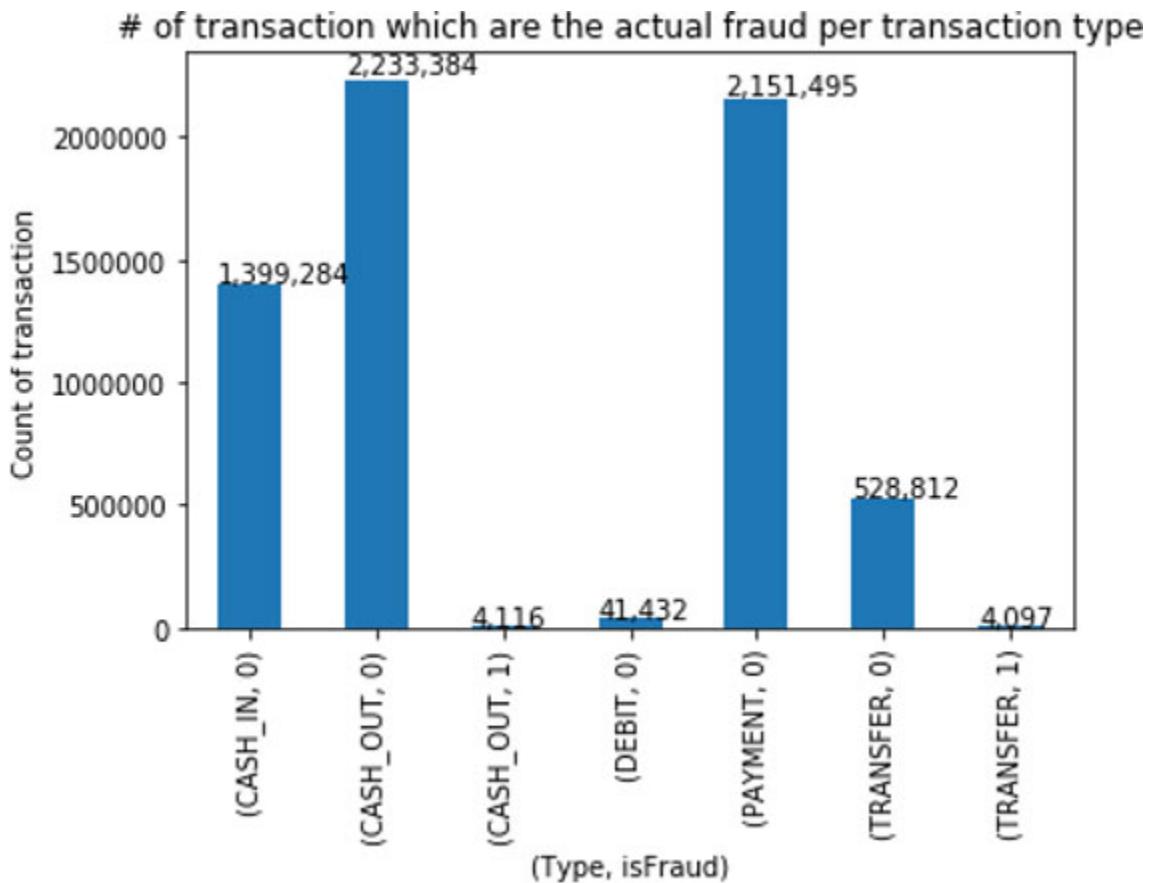
```

ax = df.groupby(['type',
'isFraud']).size().plot(kind='bar')

ax.set_title("# of transaction which are the actual fraud
per transaction type")
ax.set_xlabel("(Type, isFraud)")
ax.set_ylabel("Count of transaction")
for p in ax.patches:
    ax.annotate(str(format(int(p.get_height()), ',d')),
(p.get_x(), p.get_height()*1.01))

```

The output of the above plot showing `isFraud` transactions is the following image:



**Figure 16.3:** *isFraud* transactions

In the next step, we will look at the transactions that were flagged fraud.

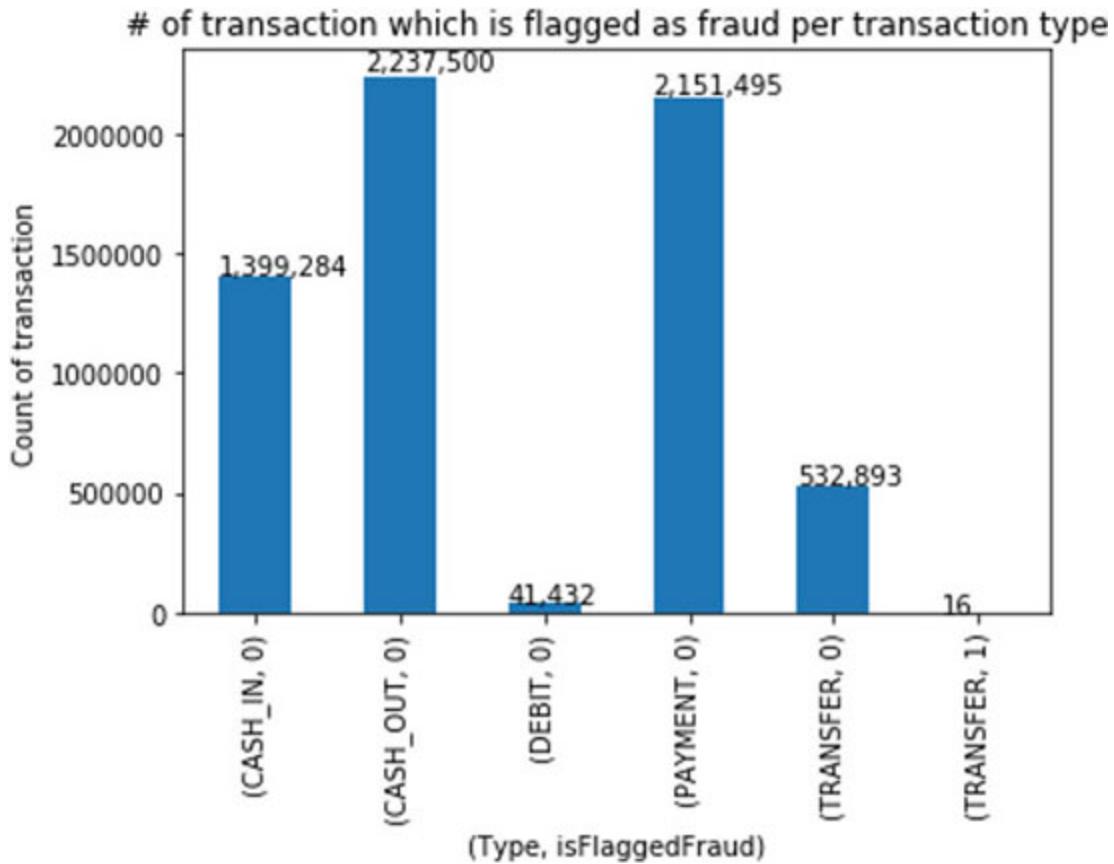
- Let's look at some interesting data that shows us as to which transactions are fraud and which of them have been flagged fraud due to the system's understanding:

```

ax = df.groupby(['type',
 'isFlaggedFraud']).size().plot(kind='bar')
ax.set_title("# of transaction which is flagged as fraud
per transaction type")
ax.set_xlabel("(Type, isFlaggedFraud)")
ax.set_ylabel("Count of transaction")
for p in ax.patches:
    ax.annotate(str(format(int(p.get_height()), ',d')), 
    (p.get_x(), p.get_height()*1.01))

```

The output of the above plot showing **isFlaggedFraud** transactions is the following image:



**Figure 16.4:** *isFlaggedFraud transactions*

In the next section, we will be exploring the modelling algorithm for this dataset.

**Section II:** In this section, we will look at the modelling aspects of this dataset. We will use the understanding from EDA and apply it in this section of our case study.

10. In this step, we will look at which are the important attributes to keep and which are the attributes to drop for now. We remove parameters like `step`, `nameOrig`, `nameDest` and `isFlaggedFraud`, which are not highly contributing to the final result(detecting fraudulent patterns).

```
from statsmodels.tools import categorical  
  
df_model = df.loc[(df['type'].isin(['TRANSFER',  
'CASH_OUT'])),:]
```

```

df_model.drop(['step', 'nameOrig', 'nameDest',
               'isFlaggedFraud'], axis=1, inplace=True)
df_model = df_model.reset_index(drop=True)
a = np.array(df_model['type'])
b = categorical(a, drop=True)
df_model['type_num'] = b.argmax(1)

print(df_model.head(5))

```

The output for the latest dataframe which contains the columns used for modelling is:

	type	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	\
0	TRANSFER	181.00	181.0	0.0	0.0	
1	CASH_OUT	181.00	181.0	0.0	21182.0	
2	CASH_OUT	229133.94	15325.0	0.0	5083.0	
3	TRANSFER	215310.30	705.0	0.0	22425.0	
4	TRANSFER	311685.89	10835.0	0.0	6267.0	

	newbalanceDest	isFraud	type_num
0	0.00	1	1
1	0.00	1	0
2	51513.44	0	0
3	0.00	0	1
4	2719172.89	0	1

*Figure 16.5: Output of the latest dataframe used for modelling*

In the next step, we will be splitting the data into the training and the test dataset.

11. In this step, we will use the `train_test_split` function to split the datasets into training and test datasets.

```

from sklearn.model_selection import train_test_split

# Whole dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.3, random_state = 0)

print("Number transactions train dataset: ",
      format(len(X_train), ',d'))
print("Number transactions test dataset: ",
      format(len(X_test), ',d'))
print("Total number of transactions: ",
      format(len(X_train)+len(X_test), ',d'))

```

## **Output:**

```
Number transactions train dataset: 1,939,286  
Number transactions test dataset: 831,123  
Total number of transactions: 2,770,409
```

12. To model this dataset, we will use the **SVM (Support Vector Machine)** algorithm with the linear kernel:

```
from sklearn.svm import SVC # "Support Vector Classifier"  
clf = SVC(kernel='linear')  
  
# fitting x samples and y classes  
clf.fit(X_train, y_train)
```

## **Output:**

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,  
     decision_function_shape='ovr', degree=3,  
     gamma='auto_deprecated',  
     kernel='linear', max_iter=-1, probability=False,  
     random_state=None,  
     shrinking=True, tol=0.001, verbose=False)
```

13. In this step, we will use the above-created model to predict the fraud variable in this dataset:

```
y_predicted = clf.predict(X_test)
```

14. We can now use the `accuracy_score` function from `sklearn.metrics` to find out the accuracy score of this prediction. In this case study, we will not be going deep-dive into improvising the model for accuracy based on the various accuracy parameters discussed in the previous chapters. Still, we will focus on just the case study flow.

Here is the command to find the accuracy of the predictions:

```
from sklearn.metrics import accuracy_score  
print(accuracy_score(y_test, y_predicted, normalize=True,  
                      sample_weight=None))
```

## **Output:**

```
0.9970341333352585
```

## Case study summary

Now that we have seen the steps that need to be taken when dealing with a particular dataset let's go back and try to answer the different questions that need to be answered when it comes to an ML Case Study.

- **What problem does this solve?**

Our case study addresses the issue of digital financial fraud. Through this case study, we are trying to understand as to how we can come up with a better threshold for fraud detection data.

- **Who is the target audience/client?**

In this case, the target audience would be organizations who are trying to prevent anti-money laundering and financial fraud. It is a constant issue where large fintech corporations are trying to figure out a way to reduce financial risk/fraud using all the data that they collect from the customers on a day to day basis.

- **What are the existing solutions present today?**

Several solutions are constantly monitoring data and trying to figure out unusual patterns or patterns that are out of the ordinary to detect fraud in a financial ecosystem. But finance-related data is sensitive and rarely available for analysis, which is one of the challenges that the data industry is trying to solve.

- **How and where is the input data being collected?**

In our case, we are producing the data using PaySim, which is a simulator.

- **What is the required output, and how should the output be structured? (a report, a document, pie charts, and so on.)**

It is a question which we still need to explore in our case study. It depends on the organization as to how the organization is planning to use the data. In case of fraud detection, organizations may want to monitor the suspected accounts for a long period, or they may want to create a report and understand how certain departments are performing to catch the fraud, and so on.

In our case, we are focusing on going through the process of producing results using a python code which can be used by

organizations.

## **Conclusion**

This case study is just an example of how we can use industry data to analyze and produce significant results that could be used by the industries for making a profit or for saving their losses. It is important to understand that the data we receive may not be in the best shape for consuming directly. In most of the situations, dirty data is collected and polished by the data scientists or analysts to make it usable for modelling.

Exploratory Data Analysis is one of the major steps in understanding the data before thinking about modelling. It is also important to understand how this data is going to be useful to the industry/organization. In several scenarios, conclusions based on data analysis can vary from one company to another based on the multiple factors and questions that we have looked at above.

Machine Learning can be a powerful tool for organizations to use the data that they have and create useful insights to serve their customers better. It can also be used to improve the existing systems of an organization based on the insights provided by machine learning algorithms on your data.

## **Citation used for a dataset**

PaySim first paper of the simulator:

E. A. Lopez-Rojas, A. Elmir, and S. Axelsson. "PaySim: A financial mobile money simulator for fraud detection". In: The 28th European Modeling and Simulation Symposium-EMSS, Larnaca, Cyprus. 2016