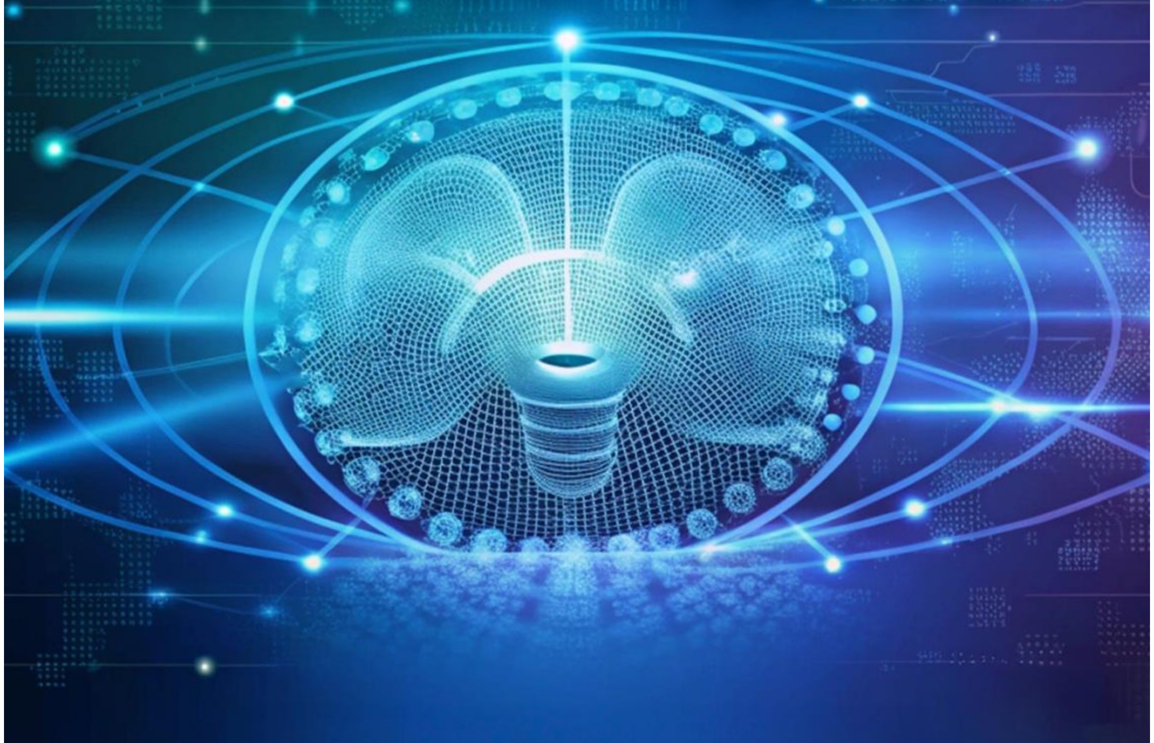
LEARNING TO PROGRAM WITH AI, BIG DATA AND THE CLOUD

PYTHON ARTIFICIAL INTELLIGENCE



DANIEL BEGUM

Learn AI with Python

Table of Contents

Reinforcement learning

CHAPTER 1
Introduction to AI and Python

Introduction

What's the very first thing that comes into your mind when you think of **Artificial Intelligence** (**AI**)? It may be an automated machine, robots, or an image of the brain with some processing. If yes, then your understanding of AI is appropriate but vague. So, you may be wondering, what exactly the concept of AI is? This chapter provides a brief overview of AI. It covers various fields of study in AI, real-life applications of AI, and agents and environments. This chapter also addresses the Python programming language, one of the most popular programming languages used by developers today for building AI applications. It also highlights features of Python, its installation, and steps to run the Python script.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Artificial Intelligence (AI)
- Learning AI
- Understanding intelligence
- Various fields of study in AI
- Application of AI in various industries
- How does artificial intelligence learn?
- AI – agents and environments
- AI and Python – how do they relate?
- Python3 – installation and setup

Objectives

After studying this unit, you will understand the basics of AI. You will also learn various fields of study in AI and its applications in various industries. You will be able to install Python 3 on Windows, Linux, and Mac OS X. You will also understand the reason for choosing Python for AI projects.

Introduction to Artificial Intelligence (AI)

John McCarthy, an American computer scientist, who was a pioneer and an inventor, coined the term **Artificial Intelligence** (**AI**) in his 1955 proposal for the 1956 Dartmouth Conference, the first artificial intelligence conference. According to him, AI is *The science and engineering of making intelligent machines, especially intelligent computer programs*.

As we can see, Artificial Intelligence is composed of two words, first is Artificial, which means man-made, and second is Intelligence, which means thinking power. Hence, we can say that AI means a man-made thinking power. We can define AI as:

"A branch of information technology by which we can create intelligent machines that can think like a human, behave like a human, and also able to make the decisions at its own."

AI is accomplished by studying how humans think, learn, and decide while trying to solve a problem, and then using this outcome as a base for developing intelligent machines. The best part of AI is that we do not have to preprogram a machine, instead we can create a machine with programmed algorithms that can work with its own intelligence.

Why to learn AI?

Are machines capable of thinking? This is a simple question that is very difficult to answer. Different researchers defined terms such as thought or intelligence in different ways. When we look more closely at AI, this is just one of the problems that are encountered.

But, one thing is clear that the current progress in the development of algorithms, combined with greater processing power and exponential growth in the amount of available data, means that AI is now capable of developing systems that can perform tasks that were previously viewed as the exclusive domain of human beings. Some of the capabilities of AI, due to which we should learn it, are as follows:

- **AI is capable of learning through data**: In our day-to-day life, we deal with huge amounts of data and our mind can't keep track of such huge data. AI's capability of learning through data helps us to automate things.
- **AI is capable of teaching itself**: In this digital era, data itself keep changing at a rapid pace, so the knowledge that is derived from such data must also be updated constantly. To fulfill this purpose, a system should be intelligent, and AI can help us to create such intelligent systems.
- **AI can respond in real time**: If you use the internet regularly, you're probably using some real-time applications in the fields of e-commerce, healthcare, retail, manufacturing, self-driving cars, and so on. AI along with the help of neural networks can analyze the data more deeply and hence can respond to the situations that are based on the conditions in real time.
- **AI can achieve a greater degree of accuracy**: Deep learning, a subset of machine learning, extends the potential of AI to more complex tasks that can only be computed through multiple steps. These tasks are often performed with a greater degree of accuracy.

Understanding intelligence

To build AI applications (smart systems that can think and act like a human), it's necessary to understand the concept of intelligence. As discussed before, different researchers defined terms such as thought or intelligence in different ways. Let's define intelligence keeping in mind the scope of AI:

- **Ability to take decisions**: From a set of many deciding factors, it’s important to take the optimal, correct, and accurate decisions. This measures intelligence in a generic way as well as in terms of AI.
- **Ability to prove results**: Another important factor that measures intelligence is the ability to prove that why this decision has been chosen.
- **Ability to think logically**: Do you think, in this world, everything can be proved by mathematical formulae or proof? No, as humans, for many things, we need to apply our common sense, think logically, and conclude. This ability also measures intelligence.
- **Ability to learn and improve**: How do we develop our experiences? Whenever we learn something new, we develop our experiences. These experiences help all of us to make better decisions and better opportunities in the future. This also measures intelligence in a generic way as well as in terms of AI because the more we learn from the external environment, the more we have the ability to improve ourselves.

Types of intelligence

According to Howard Gardner[1], an American development psychologist, there are eight multiple intelligences.

- **Linguistic–verbal intelligence**: It is the ability to speak, recognize, and use the mechanism of phonology, syntax, and semantics. Some of the characteristics of people with linguistic–verbal intelligence are:
    - They enjoy reading and writing.
    - They can explain things very well.
    - They are good at debating or giving persuasive speeches.
    - They are also good at remembering written and spoken information.

  For example, writers, narrators, teachers, and journalists.
- **Musical intelligence**: It is the ability to create, communicate, and understand pitch, rhythm, and meaning of sounds. Some of the characteristics of people with musical intelligence are:
    - They enjoy singing as well as playing musical instruments.
    - They can recognize musical patterns and tones easily.
    - They are good at remembering songs and melodies.
    - They have a great understanding of musical structure, rhythm, and notes.

  For example, musicians, singers, music teachers, and composers.
- **Logical–mathematical intelligence**: It is the ability to use, understand relationships in the absence of action or objects, and understand complex as well as abstract ideas. Some of the characteristics of people with logical–-mathematical intelligence are:

  - They have excellent problem-solving skills.
  - They enjoy thinking about abstract ideas.
  - They are good at solving scientific experiments.
  - They also like conducting scientific experiments.

  For example, mathematicians, engineers, computer programmers, and scientists.

- **Visual–spatial intelligence**: It is the ability to perceive visual information, change it, re-create images without reference to the objects, construct 3-dimensional images, move, and rotate them. Some of the characteristics of people with visual–spatial intelligence are:
  - They enjoy drawing and painting.
  - They recognize patterns very easily.
  - They are good at interpreting pictures, graphs, and charts.
  - They are also good at putting puzzles together.

  For example, architects, artists, astronauts, and physicists.

- **Bodily kinesthetic intelligence**: It is the ability to use part of or complete body to solve problems. It is the control of fine and coarse motor skills. Some of the characteristics of people with bodily kinesthetic intelligence are:
  - They have excellent physical coordination.
  - They enjoy creating things by themselves.
  - They are good at dancing and sports.

  For example, players, builders, actors, and dancers.

- **Intra-personal intelligence**: It is the ability to distinguish among one’s feelings, intentions, and motivations. Some of the characteristics of people with intra-personal intelligence are:
  - They are good at analyzing their strengths and weaknesses.
  - They enjoy analyzing and learning through theories and ideas.
  - They have excellent self-awareness.
  - They understand the basis for their motivations as well as feelings.

  For example, writers, theorists, and scientists.

- **Inter-personal intelligence**: Unlike intra-personal intelligence, it is the ability to recognize and make distinctions among other feelings, beliefs, and intentions. Some of the characteristics of people with interpersonal intelligence are:
  - They are good at communicating verbally.
  - They are also skilled at nonverbal communication.
  - They always tend to create a positive relationship with others.
  - They are also good at resolving conflicts in groups.

For example, psychologists, philosophers, and politicians.

- **Naturalistic intelligence**: It is the ability to explore the environment and learning about other species. The individuals who have this type of intelligence are said to be highly aware of even the smallest changes. Some of the characteristics of people with naturalistic intelligence are:
    - They would be interested in studying subjects such as botany, biology, and zoology.
    - They enjoy camping, gardening, hiking, and outdoor activities.
    - They don't enjoy learning topics that have no relation to nature.

  For example, farmer, gardener, biologist, and conservationist.

A system is artificially intelligent if it is equipped with at least one or at most all intelligence in it.

Various fields of study in AI

As soon as we start thinking about AI, various terms like **Machine Learning** (**ML**), **Deep Learning** (**DL**), **Natural Language Processing** (**NLP**), **Data Science**, **Statistical Analysis**, **Artificial Neural Network** (**ANN**), **Genetic Algorithms**, and so on come into our mind. But if we see broadly, AI is not an isolated domain, it's an umbrella of every technology that helps transcend human capabilities. Let's have a look at some of the fields of study within AI:

- **Machine Learning (ML)**: Machine Learning, one of the most popular fields of study, is a subset of AI that allows machines to learn on their own as humans can learn from their experiences. It learns from the dataset and makes predictions.
- **Deep Learning (DL)**: Deep Learning is a subset of ML concerned with algorithms inspired by the function of the brain called ANN. It makes the computation of a multi-layer neural network possible.
- **Logic**: According to the Oxford dictionary, *Logic is the reasoning conducted or assessed according to strict principles and validity*. To an extent, it carries the same meaning in AI as well. We can define logic as proof of validation behind any reason provided. But why it's important to include logic in AI? It's because we want our system (agent) to think like humans, and for doing so, it should be capable of making the decision based on the current situation.
- **Knowledge Representation**: We humans are best at understanding, reasoning, and interpreting knowledge because as per our knowledge, we can perform various actions in the real world. But how machines can do all these things comes under **Knowledge Representation** (**KR**). KR is concerned with AI agents thinking and how thinking contributes to the intelligent behavior of agents. Intelligence is dependent on knowledge because an AI agent will only be able to accurately act on the input when it has some knowledge or experience about that input.

Applications of AI in various industries

Artificial intelligence, machine learning, and deep learning are here, growing, and with each passing day they are making machines smarter and smarter. In fact, they are becoming a disruptive force that is redefining today's world. They have come roaring out of high-tech labs to become something that we use every day without even realizing it. Also, the acceleration we have seen in recent years shows no signs of slowing down. With applications ranging from heavy industry to healthcare, the presence and importance of AI and ML technology are being felt across a broad spectrum of industries. Let's have a look at the top five fast-growing industries that are tremendously reaping the benefits of this technology:

- **Education**: Education is the backbone of any nation. AI is improving the education system by replacing traditional techniques with personalized, and immersive learning techniques. This way, it helps teachers to tailor students' weaknesses. Two of the realities of immersive learning are **Augmented Reality** (**AR**) and **Virtual Reality** (**VR**). Augmented reality is a type of software that uses the device's camera to overlay digital aspects into the real world. It facilitates the teachers and the trainers in performing those tasks, in a safe environment, which they previously could not. On the other hand, virtual reality creates a 360-degree view digital environment, which allows students to interact directly with the study material by using e-learning resources on mobile devices.
- **Healthcare**: One of the latest advancements in healthcare is Google's Medical Brain, which is enabled with a new type of AI algorithm. Google's Medical Brain is used to make predictions about the likelihood of death among patients. AI is also helping the laboratory segment of healthcare with ML-enabled laboratory robots that can study new molecules and reactions. In recent years, cancer has been one of the leading causes of death. Companies like Infervision have developed an AI-based system, which is trained with suitable algorithms, to review CT scans and detect early signs of cancer. In this coronavirus pandemic phase, AI is also used to accurately forecast infections, deaths, and recovery timelines of the COVID-19.
- **Automobiles**: Driverless or self-driving vehicles are not a sci-fi thing anymore. With huge advancements in AI, it became a reality now. Tech giants such as Google, Apple, Amazon, Cisco, Intel, and Bosch are leading the R&D in autonomous driving. Whereas automobile companies such as **General Motors** (**GM**), Tesla, BMW, and Mercedes are some serious players in the self-driving vehicle game. Autonomai, enabled with Deep Learning and AI capabilities, is an autonomous middleware platform developed by an Indian company named Tata ELEXSI. It's not far when we will see and use self-driving vehicles on Indian roads as well.
- **E-Commerce**: In this e-commerce and digital era, we all have the experience of online shopping and we sometimes also buy the stuff that is not required at all or we seldom use. The new strategy of e-commerce companies is to sell the stuff to their customers even before they realize the need for it. Companies achieve this by realizing their customers' preferences and various other factors like attractive deals, special coupons, and discounts. This strategy is called 'purchase recommendations' or 'intuitive selling', which is purely based on AI algorithms. For example, by using AI algorithms and computer vision, Amazon go is redefining the way of shopping in supermarkets. It adds the items automatically in customers' virtual cart and once the customer leaves the store, adds the charges on the Amazon account. So, no more lines at the time of checkout.
- **Digital marketing**: Imagine how effective marketing becomes if most of the time-consuming tasks such as identifying the right perspective, segmenting as well as targeting

audiences, building a winning content strategy, and scheduling the release could be driven without human intervention. AI is bringing this power into marketing automation by using tools like Boomtrain, Phrases, Persado, Adext, RankBrain, Chatbots, and so on.

You may be wondering if AI, ML, and DL have any application(s) for day-to-day life or they are meant for industrial use only. Look around and you could see and feel AI-powered things and devices.

Following are some cool AI applications enhancing our lifestyle:

- **Virtual Personal Assistants (they are intelligent)**: Most of us are interacting with virtual personal assistants like Siri, Alexa, Cortana, and Google Now on a regular basis for getting the desired information. It's AI technology with the help of which these VPAs continually learn information about us to provide better services. In fact, now we can use Google Assistant to talk to the 'Tulips' flower. Google and Wageningen University made it possible by mapping tulip signals to human signals on Google Assistant's existing Neural Machine Translation. Google Assistant now added *Tulipish* as a language and offers translation between dozens of other human languages. So, now we can say, *Okay Google, talk to my Tulip*. Astonishing, right?
- **Video games**: I am sure everyone has memories about the classic video games like *Road Rash*, *Pac-Man*, *Super Mario*, *Virtua Fighter*, and *Nokia Snake* game. But if we see today's video games like *Fortnite*, *Call of Duty*, *Grand Theft Auto*, and *Far Cry*, they are very advanced because they are empowered by AI algorithms. These algorithms make today's games look highly realistic because the characters in the game understand a gamer's behavior, learn from stimuli, and change their traits accordingly. Such features attract a player to come back to play again and again.
- **Humanoid (human-like robots for humans)**: Around 2 years ago, a humanoid robot named *Sophia* became the first-ever robot to have a nationality. Yes, in October 2017, Sophia, created by a Hong Kong firm named Hanson Robotics, got Saudi Arabia's citizenship. This AI-powered robot can imitate human gestures, facial expressions, and initiate discussion on predefined topics as well. That's why we can call Sophia 'a social humanoid robot'.

  In fact, India is also not far behind. We have Rashmi, the world's first Hindi-speaking humanoid robot, who is hosting a show on Red FM since December 2018. It is created by Ranchi's Ranjit Srivastava. We can call Rashmi, 'The Indian Sister of Sophia'.
- **Maps and directions**: Everyone has Google Maps or any other app similar to it on their smartphones for finding directions and routes. With such apps, there is no more fear of getting lost. Here also AI is helping us out. Google uses **Graph Neural Networks** (**GNN**), which is an ML architecture to reduce the percentage of inaccurate **Expected Time of Arrivals** (**ETAs**).
- **Cab-service ride-sharing feature**: In today's scenario, one of the best ways to commute is cab services like Ola, Uber, and so on. To save expenses, many of us used to share our rides with other passengers. But have you ever thought:
  - How does the cab service app get a booking from the person going on the same route as yours?

    - In a shared ride, how an individual’s fare is determined?

All such queries have only one answer – AI algorithms.

How does artificial intelligence learn?

Today, AI helps various industries as discussed in the preceding section. These AIs are often self-taught, they work off a simple set of instructions to create a unique set of rules and strategies. So how exactly does a machine learn? There are various ways to build self-teaching programs, but they all rely on the following three basic types of machine learning:

- **Supervised learning**: It takes the data sample (usually called training data) and associated output (usually called labels or responses) with each data sample during the training process of the model. The main objective of supervised learning is to learn an association between input training data and corresponding labels.
- **Unsupervised learning**: Unsupervised learning methods (as opposed to supervised learning methods) do not require any pre-labeled training data. In such methods, the machine learning model or algorithm tries to learn patterns and relationships from the given raw data without any supervision. Although there are a lot of uncertainties in the result of these models, we can also obtain a lot of useful information like all kinds of unknown patterns in the data, the features that can be useful for categorization, and so on.
- **Reinforcement learning**: In reinforcement learning algorithms, a trained agent interacts with a specific environment. The job of the agent is to interact with the environment and once observed, it takes actions regarding the current state of that environment.

AI agents and environments

AI is all about practical reasoning, reasoning in order to do something, and an AI system is composed of an agent and its environment. The agents act in their environment and the environment may contain other agents.

What is an agent?

An agent may be defined as anything that can perceive its environment through sensors and acts upon that environment through effectors. An agent, having mental properties such as knowledge, belief, intention, and so on, runs in the cycle of perceiving, thinking, and acting. Examples of agents are:

- A **human agent** has sensory organs like eyes, ears, tongue, skin, and nose, which work as sensors. On the other hand, it has hands, legs, and vocal tract, which work as effectors.
- A **robotic agent** has cameras and infrared range finders, which act as sensors. On the other hand, it has various motors acting as effectors.
- A **software agent** has keystrokes, files, received network packages, and encoded bit strings, which work as sensors. On the other hand, it has sent network packages, content displays on the screen, which work as effectors.

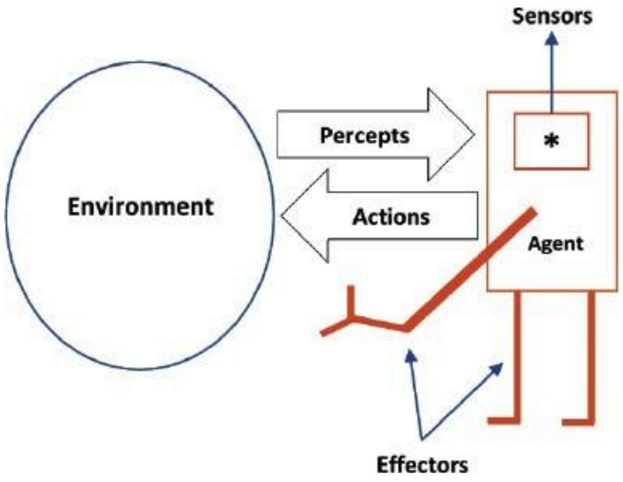


Figure 1.1: *Agent and its environment*

For an AI agent, the following are the four important rules:

- **Rule 1**: It must have the ability to perceive the environment.
- **Rule 2**: It must use observation to make decisions.
- **Rule 3**: The decisions it makes should result in an action.
- **Rule 4**: Every action it takes must be a rational action.

Agent terminology

- **The performance measure of an agent**: It may be defined as the criteria determining how successful an agent is.
- **The behavior of an agent**: It may be defined as the action that an agent performs after any given sequence of percepts.
- **Percept**: An agent's perceptual inputs at a given instance is called a percept.
- **Percept sequence**: It may be defined as the history of all that an agent has perceived till now.
- **Sensor**: Sensor, through which an agent observes its environment, is a device detecting the change in the environment and sending the information to other devices.
- **Effectors**: They are the devices affecting the environment. They can be hands, legs, arms, fingers, display screen, sent network packet, wings, fins, and so on.
- **Actuators**: Actuators, only responsible for moving and controlling a system, are the components of machines that convert energy into motion. Examples of actuators can be electric motors, gears, rails, and so on.

Rationality and rational agent

The rationality of an agent is concerned with the performance measure of that agent. As we know, the agent should perform actions to obtain useful information. So, in simple words, we can define rationality as the status of being sensible, reasonable, and having a good sense of judgment. There are following four factors on which the rationality of any agent depends upon:

- The **Performance Measures** (**PM**)of an agent.
- Agent’s **Percept Sequence** (**PS**).
- Agent’s **Prior Knowledge** (**PK**) about the environment.
- The **Actions** (**A**) an agent can carry out.

(PM, PS, PK, A)

An ideal rational agent is the one that has clear preferences, models uncertainty, and is capable of doing expected actions to maximize its performance measure, based on its percept sequence and built-in knowledge base. A rational agent is said to perform the right actions always. Here the right actions mean the actions that cause the agent to be most successful in the given percept sequence.

Structure of an AI agent

The main task of artificial intelligence is to create and design an agent program that implements the agent function. In this way, the following structure of an AI agent can be viewed as the combination of architecture and agent program:

Agent = Architecture + Agent program

Architecture, agent function, and agent program are the three main terms involved in the structure of an AI agent.

- **Architecture**: It is the machinery an AI agent executes on.
- **Agent function**: It may be defined as the map from the percept sequence to an action.

$$f : p^* \rightarrow A$$

- **Agent program**: It is an implementation of agent function that executes on the physical architecture to produce function.

P.E.A.S representation

P.E.A.S representation is a type of model in which the properties of an AI agent or rational agent can be grouped. It consists of four words:

- **P**: Performance measure
- **E**: Environment
- **A**: Actuators
- **S**: Sensors

As discussed, the objective for the success of an agent’s behavior is the performance measure.

Let’s see two examples of agents with their P.E.A.S representation:

- **Self-driving vehicles**: The P.E.A.S representation for self-driving vehicles will be:
    - **P (Performance)**: Safety, time, legal driving, and comfort.

    - **E (Environment)**: Road, road signs, other vehicles, and pedestrian.
    - **A (Actuators)**: Accelerator, steering, brake, clutch, signal, and horn.
    - **S (Sensors)**: Camera, speedometer, GPS, sonar, and accelerometer.

- **Vacuum cleaner**: The P.E.A.S representation for vacuum cleaners will be:
    - **P (Performance)**: Cleanness, battery life, efficiency, and security.
    - **E (Environment)**: Room, wooden floor, carpet, other obstacles like shoes, bed, table, and so on.
    - **A (Actuators)**: Brushes, wheels, and vacuum extractors.
    - **S (Sensors)**: Camera, cliff sensor, dirt detection sensor, bump sensor, and infrared wall sensor.

Types of agents

Based on the degree of perceived intelligence and capability, agents can be grouped into the following four classes:

- Simple reflex agent
- Model-based reflex agent
- Goal-based agent
- Utility-based agent

Simple reflex agent

- They choose actions based only on the current percept and ignore the rest of the percept history.
- They work based on the condition–action rule, which is a rule that maps a state, that is, condition to an action. If the condition is true, the action is taken, otherwise not. For example, a room cleaner agent works only if there is dirt in the room.
- Their environment is fully observable.

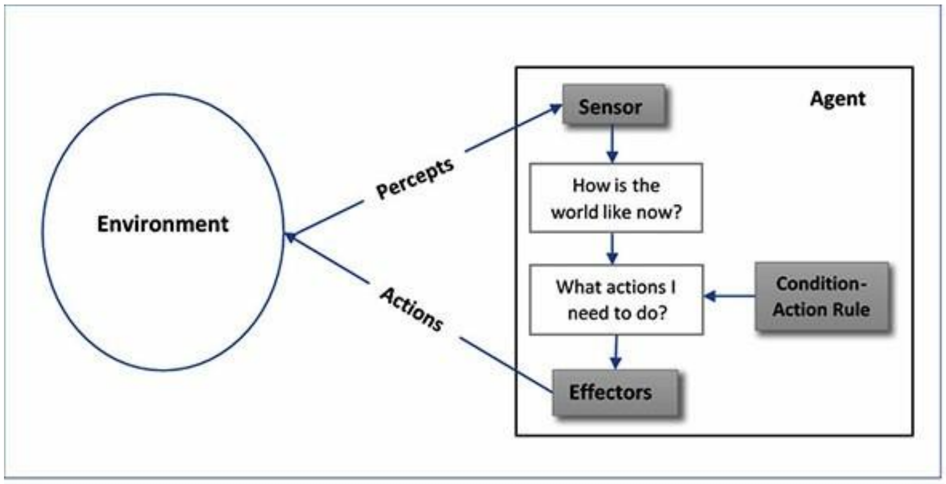


Figure 1.2: *Simple reflex agent*

Model-based reflex agent:

- In order to choose their actions, they use a model of the world.
- It must keep track of the internal state, adjusted by each percept, that depends on the percept history.
- They can handle partially observable environments.
- Model is the knowledge about how things happen in the world.
- Internal state is a representation of unobserved aspects of the current state, which depends upon percept history.
- In order to update the agent's state, it requires the following information:
    - How the world evolves?
    - How do the actions of agents affect the world?

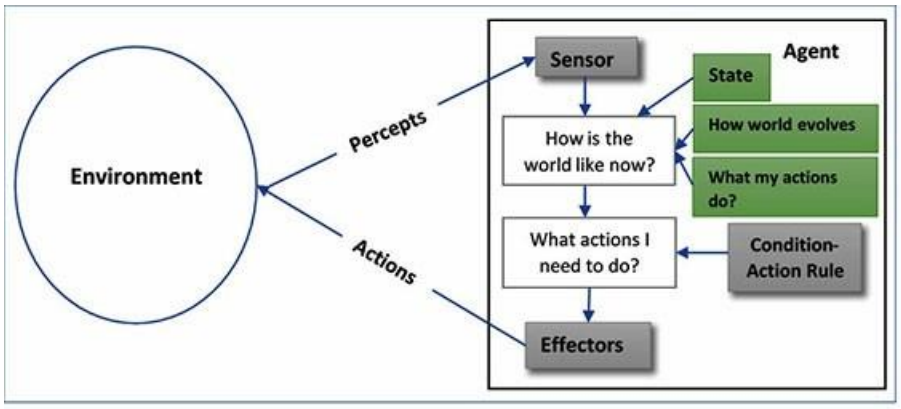


Figure 1.3: *Model-based reflex agent*

Goal-based agent

- They choose their actions and take decisions based on how far they are currently from their goals – a description of a desirable situation.
- Every action of such agents is intended to reduce the distance from the goal.
- This approach, that is goal-based, is more flexible than reflex agents because the knowledge supporting a decision is explicitly modeled, which allows for modifications.

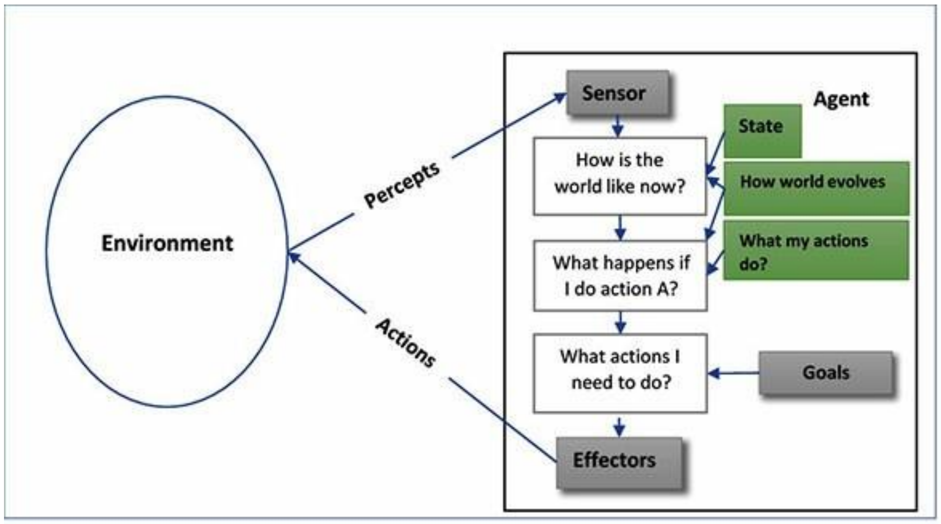


Figure 1.4: *Goal-based agents*

Utility-based agent

- They are developed having their end uses as building blocks.
- In order to decide which is the best among multiple possible alternatives, utility-based agents are used.
- They choose their actions and take decisions based on a preference (utility) for every state.
- Sometimes, achieving the desired goal is not enough because goals are inadequate when:
    - We have conflicting goals and only a few among them can be achieved.
    - Goals have some uncertainty of being achieved.

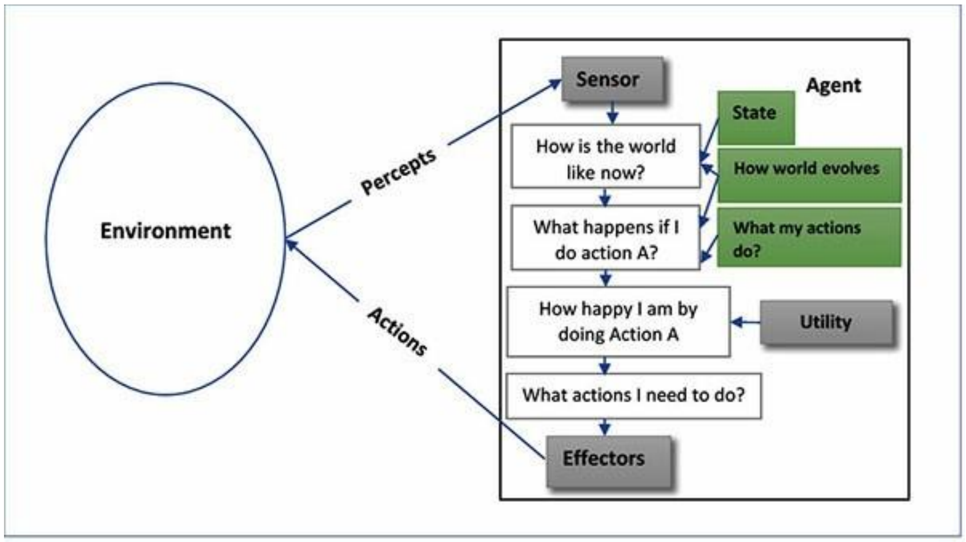

Environment
Percepts
Actions
Agent
Sensor
How is the world like now?
State
How world evolves
What happens if I do action A?
What my actions do?
How happy I am by doing Action A
Utility
What actions I need to do?
Effectors

Figure 1.5: *Utility-based agents*

What is an agent’s environment?

Everything in the world that surrounds the agent is called an agent’s environment. It can’t be a part of an agent itself, but it’s a situation in which the agent is present. In simple words, we can say that the environment is where an agent lives and operates. It’s an environment that provides an agent something to sense and act upon it.

Nature of environments

There are several aspects such as the shape and frequency of the data, the nature of the problem, and the volume of knowledge available at any given time, that distinguish one type of AI environment from another. If anyone wants to tackle a specific AI problem, they should first understand the characteristics of AI environments. From that perspective, based on the nature of the environment, we use several categories to group AI problems.

- **Fully observable versus partially observable**: Fully observable AI environments are those on which, at any given time, an agent sensor can sense or access the complete state of an environment. It’s very simple as there is no need to maintain the internal state to keep track of history. Image recognition operates in a fully observable AI environment.

  Partially observable AI environments are those on which, at any given time, an agent sensor cannot sense or access the complete state of an environment. Self-driving vehicle scenarios operate in a partial observable AI environment.

- **Complete versus incomplete**: Complete AI environments are those on which, at any given time, an agent has enough information to complete a branch of the problem. Chess is a classic example of such AI environments.

  On the other hand, incomplete AI environments are those in which agents can’t anticipate many moves in advance. The agents, at any given time, focus on finding a good equilibrium state. Poker is a classic example of incomplete AI environments.

- **Static versus dynamic**: Static AI environment is an environment that cannot change itself while an agent is deliberating. That's the reason they are easy to deal as the agent doesn't need to continually look at the world while deciding for an action. Speech analysis and crossword puzzles are the problems operating on a static AI environment.

  In contrast, dynamic AI environment is an environment that can change itself while an agent is deliberating. In dynamic environments, at each action, agents need to continually look at the world. Taxi driving and vision AI systems in drones are some problems operating in dynamic AI environments.

- **Discrete versus continuous**: Discrete AI environment is an environment in which there are a finite (although arbitrarily large) number of percepts and actions that can be performed within. The games such as Chess and GO also come under a discrete environment.

  In contrast to a discrete environment, a continuous environment relies on unknown and rapidly changing data sources. Vision AI systems in drones and self-driving vehicles are examples of a continuous AI environment.

- **Deterministic versus stochastic**: As the name implies, in a deterministic AI environment, an agent's current state and selected action can determine the next state of the environment. As in a fully observable environment, the agent does not need to worry about uncertainty in a deterministic environment as well. Most of the real-world AI environments are not deterministic in nature.

  On the other hand, a stochastic AI environment is an environment that cannot be determined completely by an agent. It's random in nature. Self-driving vehicles are examples of stochastic processes.

- **Single-agent versus multi-agent**: As the name implies, in a single-agent AI environment, there is only one agent that is involved and operating by itself.

  In contrast, in a multi-agent AI environment, multiple agents are involved and operating.

AI and Python – how do they relate?

There is a lot of confusion among researchers and developers about which programing language to choose for building AI applications. The list may include LISP, Prolog, Python, Java, C#, and a few more as well. The choice of a programming language depends upon many factors like ease of code, personal preference, and available resources. Although the skills of the developer always matter more than any programming language, here, we are going to justify just one, that is, Python programming language for AI.

What is Python?

Python, created by Guido Van Rossum in 1991, is an OOPs based high-level interpreted programming language and focuses on **Rapid Application Development** (**RAD**) and **Don't Repeat Yourself** (**DRY**). Due to ease of learning and adaptation, Python has become one of the fastest-growing programming languages. Python's ever-evolving libraries make it a good choice for any project whether IoT, Data Science, AI or Mobile App.

Why choose Python for building AI applications?

Python programming language is favored by developers for a whole set of applications, but what makes it a particularly good fit for applications/projects involving AI? Let’s have a look:

- **A great library and framework ecosystem**: One of the aspects that makes Python the most popular language used for AI is its abundance of libraries and frameworks. A library is a module or group of modules, published by various sources like PyPi, that includes a pre-written piece of code that saves development time and allows users to perform different actions or reach some functionality. As we know, ML and DL require continuous data processing, and Python’s libraries let us access, handle, and transform data. The following are some of the widespread libraries we can use for building AI applications:
    - **Scikit-learn**: It is very useful in handling basic ML algorithms like clustering, regression, classification, and so on.
    - **Pandas**: It is used for high-level data structures and analysis. It allows the filtering and merging of data. It can also be used for gathering data from external resources.
    - **Keras**: It’s a very useful Python library for deep learning. It uses the GPU in addition to the CPU, hence allows fast calculations.
    - **TensorFlow**: Another useful library for deep learning. It allows efficient training and utilizing an ANN with massive datasets.
    - **NLTK**: One of the most useful Python libraries for working with natural language recognition and computational linguistics.
    - **Matplotlib**: It is used for visualization. We can easily create 2D plots, histograms, charts with Matplotlib.
    - **Caffe**: Another very useful Python library for deep learning. It allows switching between the CPU and the GPU.
- **Ease of use and simplicity**: Python is almost unrivaled when it comes to ease of use and simplicity, particularly for novice AI developers. Python’s ease of use and simplicity has several advantages for ML and DL:
    - ML and DL both rely on extremely complex algorithms and multi-stage workflows. That’s why the less the developers worry about the intricacies of coding, the more they can focus on finding the solutions to the problems.
    - The simple syntax of Python makes it faster in development than many other programming languages. That’s the reason a developer can quickly test algorithms without having to implement them.
    - In addition to the preceding benefits, Python’s easily readable code is invaluable for collaborative coding.
- **Low-entry barrier**: The process of learning Python programming language is very easy because it resembles to everyday English language. That’s the reason data scientists can quickly pick up Python and start using it for developing AI applications without wasting too much time into learning the language.
- **Flexibility**: Due to its flexibility, Python for AI is a great choice:

    - Developers have an option to choose either OOPs or scripting.
    - No need to recompile the source code and developers can implement any changes and check the result quickly.
    - It can be combined with other programming languages.

  Moreover, Python's flexibility also allows a developer to choose from various programming styles like the imperative, functional, object-oriented, or procedural style.

- **Platform agnostic**: Python is also a very versatile language. What we mean is that Python is platform-independent and can run on any platform including Windows, macOS, Linux, Unix, and 21 others. With some small-scale changes in code, you can get your code running in the new OS. Again, this saves development time and money in testing on various platforms.
- **The abundance of community support**: It's always very helpful if there is strong community support built around the language. Python is an open-source programming language, which means that it is supported by a lot of resources. Moreover, a lot of Python documentation is also available online as well as in Python forums where developers can discuss errors, solve problems, and help each other out.

Python3 – installation and setup

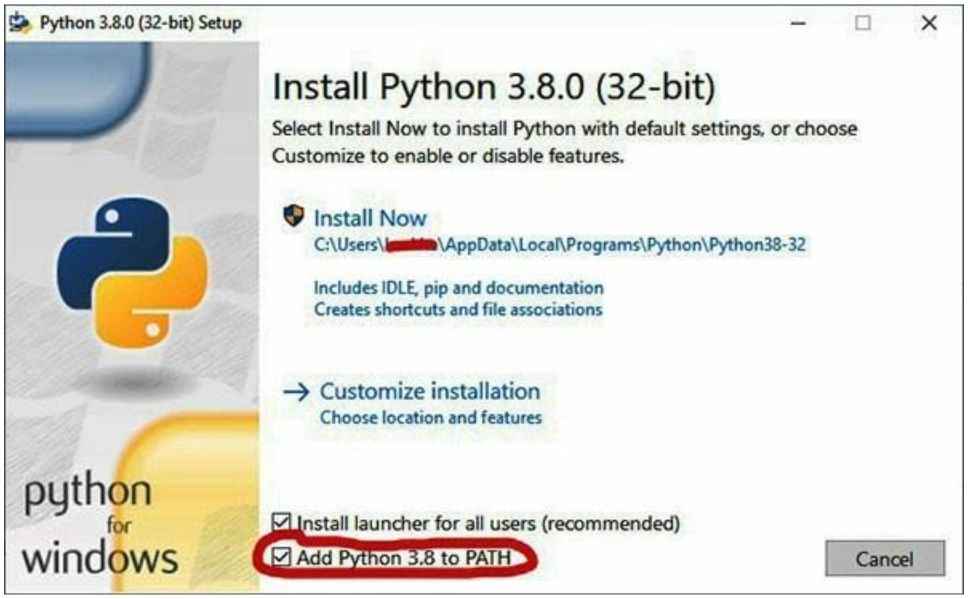
Let's see how to set up a working Python 3 distribution on Windows, macOs, and Linux.

Windows

Installing Python on Windows OS does not involve much more than downloading the Python installer from the Python.org website and running it. Follow the steps to install Python 3 on Windows:

1. **Downloading Python installer**
    1. First, open a browser window and go to **`Python.org`** website. Now, navigate to the download page for windows.
    2. Underneath the heading at the top that says **`Download the latest version for Windows`**, click on the link for the Latest Python 3 Release – Python 3.x.x (as of this writing, the latest is Python 3.8.0).
    3. Scroll to the bottom and select either of the following:
        - Windows x86-64 executable installer for 64-bit.
        - Windows x86 executable installer for 32-bit.
2. **Running the installer**

   Now after downloading an installer, we need to simply run it by double-clicking on the downloaded file. A dialog box should appear that looks something like shown in the following screenshot:



Install Python 3.8.0 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.8 to PATH

Figure 1.6: *Python setup*

In order to make sure that the interpreter will be placed in your execution path, check the box that says **`Add Python 3.x.x to PATH`**.

Linux

There are very high chances your Linux distribution has Python installed already, but it probably won't be the latest version. Instead of Python 3, it may have Python 2.

In order to find out what version(s) you have, try the following commands on terminal windows:

- **python –version**
- **python2 –version**
- **python3 --version**

One or more of preceding commands should respond with a version, as shown in the following:

```
$ python3 --version
Python 3.8.0
```

Suppose if the version shown is of Python 2 or a version of Python 3 that is not the latest one (3.8.0 as of this writing), then you should install the latest version. The procedure of installing the latest Python version will depend on the Linux distribution you are running.

Ubuntu

Depending on the Ubuntu distribution version, the instructions for installing Python vary. First, we need to determine our local Ubuntu version by running the following command:

```
$ lsb_release -a
```

We will get something like the following output for the preceding command:

```
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 16.04.4 LTS
Release: 16.04
Codename: xenial
```

Check the version number you see under `Release` in the output. Depending on the version number for your Ubuntu distribution, follow the instructions:

- Ubuntu 16.10 and 17.04 have Python 3.x.x in the `Universe` repository. we can install it with the following commands:

  ```
  $ sudo apt-get update
  $ sudo apt-get install python3.8
  ```

  Once installed, we can invoke it with the command `python3.8`.
- Ubuntu 14.04 or 16.04 do not have Python 3.x.x in the Universe repository; hence, we need to get it from a **Personal Package Archive** (**PPA**). For instance, to install Python from the PPA named `deadsnakes`, use the following commands:

  ```
  $ sudo add-apt-repository ppa:deadsnakes/ppa

  $ sudo apt-get update
  $ sudo apt-get install python3.8
  ```

  Once installed, we can invoke it with the command `python3.8`.

Linux Mint

We can follow the preceding instructions for Ubuntu 14.04 as Mint and Ubuntu use the same package management system. The PPA named `deadsnakes` works with Mint as well.

CentOS

The IUS Community is providing newer versions of software for Enterprise Linux distros that is, Red Hat Enterprise and CentOS. We can use their work to install Python 3.

In order to install, we must first update our system with the `yum` package manager. Use the following commands to do so:

```
$ sudo yum update
$ sudo yum install yum-utils
```

After that, we can install the CentOS IUS package by using the following command:

```
$ sudo yum install https://centos7.iuscommunity.org/ius-release.rpm
```

Now, with the help of the following commands, we can install Python and Pip:

```
$ sudo yum install python36u
$ sudo yum install python36u-pip
```

Fedora

Fedora has a roadmap to switch to Python 3, which indicates that the current version and the next few versions will all ship with Python 2 as the default; however, Python 3 will be installed. If the **`python3`** installed on Fedora is not Python 3.8, you can use the following command to install it:

```
$ sudo dnf install python3.8
```

Installing and compiling Python from Source

It may be possible that our Linux distribution will not have the latest version of Python, or we may not be able to build the latest version ourselves. In that case, we can use the following steps to build and compile Python from the source:

1. **Downloading the source code**

    1. To start with, we need to get the Python source code. As we did for Windows, we can go to the **`Downloads`** page of **`Python.org`** and check for the latest source (3.8.0) for Python 3.
    2. Now once the version is selected, at the bottom of the page there is a **`Files`** section. We now need to select the **`Gzipped source tarball`** and must download it on our machine.
    3. For them who prefer a command-line method, they can use **`wget`** to download it to their current directory:

        ```
        $ wget https://www.python.org/ftp/python/3.8.0/Python-3.8.0.tgz
        ```

2. **Preparing the system**

    1. In order to build Python from scratch, we need to follow some steps that are specific to that Linux distribution. Although, on all the distributions, the goal of these steps is same, but in case, if it does not use **`apt-get`**, you might still need to translate them according to your Linux distribution.
    2. Before getting started, we need to update the system packages on our machine. For apt-based systems (Debian, Ubuntu, and so on) use the following commands:

        ```
        $ sudo apt-get update
        $ sudo apt-get upgrade
        ```

    3. Next, make sure that your system has the tools needed to build Python. Some of them are listed as follows in the command. Commands for apt-based systems like Debian, Ubuntu, and so on are as follows:

        ```
        $ sudo apt-get install -y make build-essential libssl-dev zlib1g-dev
        libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm libncurses5-
        dev libncursesw5-dev xz-utils tk-dev
        ```

    4. Commands for yum-based systems like CentOS are as follows:

        ```
        $ sudo yum -y groupinstall development
        $ sudo yum -y install zlib-devel
        ```

3. **Building and compiling Python**

1. As we have the prerequisites and the TAR file, we can unpack the source into a directory. The following command will create a new directory called `Python-3.8.0`:

   ```
   $ tar xvf Python-3.8.0.tgz
   ```

2. Now go to the directory:

   ```
   $ cd Python-3.8.0
   ```

3. Now, in order to prepare the build, we need to run the `./configure` tool as follows:

   ```
   $ ./configure --enable-optimizations --with-ensurepip=install
   ```

4. Next, build the Python programs using `make`, where the `-j` option will ask you to split the building into parallel steps to speed up the compilation.

   ```
   $ make -j 8
   ```

5. As we want to install a new version of Python, we will use the `altinstall` target here so that the system's version of Python should not be overwritten. As you're installing Python into `/usr/bin`, we should run as root:

   ```
   $ sudo make altinstall
   ```

4. **Verifying the installation**

   1. Finally, with the help of the following command, we can test whether our new Python version is installed or not:

      ```
      $ python3.8 -V
      Python 3.8.0
      ```

macOS/Mac OS X

If you are using Mac OS X, the best way to install Python is through the Homebrew package manager. If you don't have Homebrew, you can install it by navigating to **http://brew.sh/**.

Once you have installed Homebrew, use the following command to install Python3:

```
$ brew install python3
```

Conclusion

The intent of this chapter was to get you familiarized with the foundations of Artificial Intelligence and Python before deep diving into building AI applications. The capabilities of AI, with a focus on important fields of study under AI, are introduced in this chapter. Machine learning, deep learning, logic, artificial neural network (ANN), and knowledge representation are some of the most important areas covered under AI. Keeping in mind the scope of AI, this chapter also defines intelligence.

AI is no longer a science-fiction term; it becomes a reality now. Industries are deploying AI and its subsets, namely, machine learning and deep learning for a more productive and profitable solution. In this chapter, we tried to make you feel the presence and importance of AI technology in five fast-growing industries, and also how these industries are reaping the benefits of this technology. Next up, we also explored some cool AI applications enhancing our day-to-day life.

The subject of AI is all about practical reasoning and is composed of agents and its environment. Concepts relevant to agents and environments have also been covered in this chapter including the structure of agents, types of agents, rationality, and nature of the environment.

We briefly described the relation between AI and Python including the features of Python that make it one of the most suitable languages for building AI applications. Finally, you learned how to install and set up Python on various platforms like Windows, macOS/Mac OS X, and Linux.

We brought everything covered in this chapter together to make you understand the basic concepts of AI, its impact on our lifestyle, and also learn about the Python programming language. This gets you ready for the next chapters, where you will implement AI algorithms with Python.

Questions

1. What is Artificial Intelligence (AI)? Describe some of the AI applications and explain how they are enhancing our lifestyle?
2. What are the most important fields of study within AI?
3. What is an AI agent? Explain its structure and types.
4. What is a rational agent? What are the factors on which the rationality of any agent depends?
5. Write down the features of the Python programming language that make it a good fit for building AI applications.

1 Gardner, Howard. Frames of Mind: The Theory of Multiple Intelligences. New York: Basic Books, 1983.

CHAPTER 2
Machine Learning and Its Algorithms

Introduction

Do you find any similarity between steam engines, age of science, and digital technology? They are known as the first three industrial revolutions responsible for fundamentally transforming our society and the world around us.

In this digital era, we are experiencing this for the fourth time. But, this fourth industrial revolution is powered by **Artificial Intelligence** (**AI**), **Machine Learning** (**ML**), **Deep Learning**, **IoT** (**Internet of Things**), **edge computing** along with increasing computing power like quantum computing. The data or the information is the driver and fuel of this industrial revolution.

No doubt, with better computational power and more storage resources, this data is increasing day by day at a very rapid pace. For businesses and organizations, the real challenge is to make sense of this huge data. That's the reason they are trying to build intelligence systems by using methodologies from ML, one of the most exciting fields of computer science. We can see ML as the application and science of algorithms that provide meaning to the data.

This chapter provides a brief overview of ML and its model. It also addresses various ML methods. Using the Python programming language, we will also implement some of the most useful ML algorithms.

Structure

- Understanding machine learning
- The landscape of machine learning algorithms
- Components of a machine learning algorithm
- Different learning styles in machine learning algorithms
    - Supervised learning
    - Unsupervised learning
    - Semi-supervised learning
    - Reinforcement learning
- Popular machine learning algorithms
    - Linear regression
    - Logistic regression
    - Decision tree

    - Random forest
    - Naïve Bayes
    - Support Vector Machine (SVM)
    - k-Nearest Neighbor (KNN)
    - k-means clustering

Objectives

After studying this chapter, you should be able to implement various popular machine learning algorithms, namely, Linear Regression, Logistic Regression, Decision Tree, Random Forest, Support Vector Machine (SVM), Naïve Bayes, k-Nearest Neighbor, and k-means clustering in the Python programming language. You will also learn different learning styles such as supervised, unsupervised and semi-supervised, and reinforcement used in ML algorithms.

Understanding Machine Learning (ML)

Machine learning, a subset of AI, is the practice of computer systems to extract patterns out of raw data by using an algorithm or a method. ML algorithms allow computer systems to learn from experience without explicit programming or any human intervention. To give you an example, a spam filter, one of the first applications of ML, can easily determine if the email is important or a spam.

The Landscape of Machine Learning Algorithms

The field of machine learning consists of learning algorithms that help the machine to learn from data and improve its performance with time. Also, based on its interaction with the environment or input data, there are different ways an algorithm can model a problem. In this section, we'll go through the components of an ML algorithm, different learning styles, or learning models that an ML algorithm can have, and we'll take a tour of the most popular ML algorithms also.

Components of a Machine Learning algorithm

Before deep diving into the components of the ML algorithm, we must understand ML through the very interesting definition given by Professor Mitchell in 1997[1]:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

The preceding definition focuses on the following three parameters:

- **(T)**: Task
- **(P)**: Performance
- **(E)**: Experience

These three are the main components of any of the learning algorithms shown in the following figure:

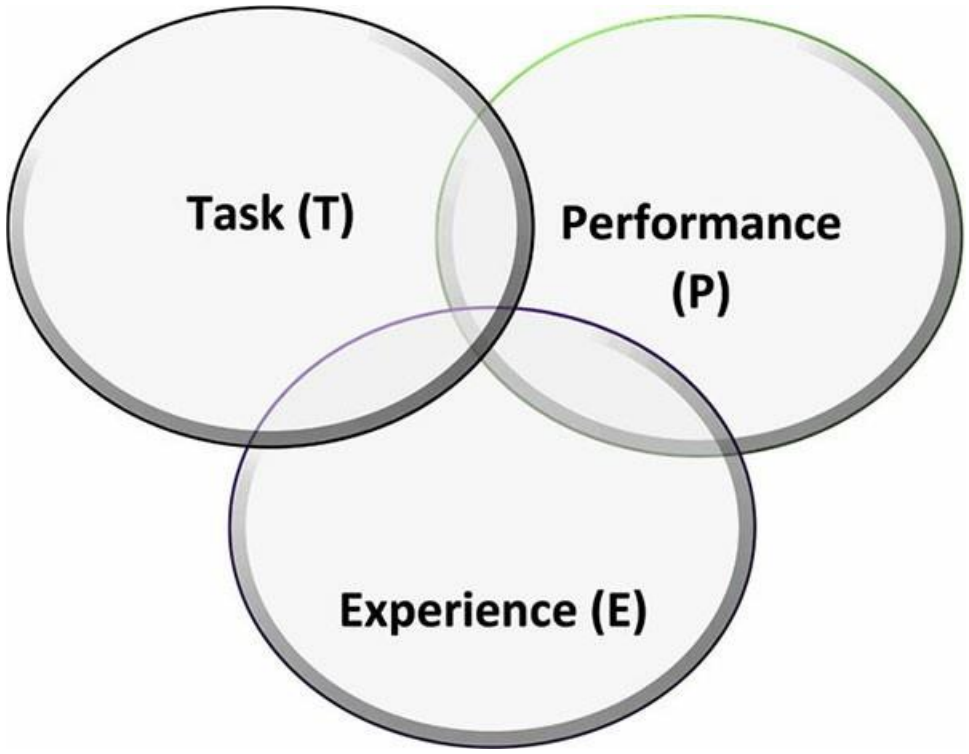


Figure 2.1: *Components of ML algorithm*

Let's simplify the definition of ML:

ML is that field of AI which consists of learning algorithms that improve their performance (P), at executing some task say T, over the time with experience E.

The three main components of the ML algorithm are described as follows:

- **Task (T)**: A task should be defined in a two-fold manner. A task, T, from a problem's perspective, can be defined as the real-world problem to be solved. The problem, finding the best marketing strategy or predicting the house price, can be anything. Whereas from the ML perspective, defining a task is quite different because it's difficult to solve ML-based tasks by using the traditional programming approach. A task, T, is called a machine learning-based task if it is based on the workflow that the system should follow to operate on sample data points. These sample data points typically consist of data attributes or features. Classification, Regression, Anomaly detection, Clustering, Translation, and so on, are some of the tasks that could be classified as the ML tasks.
- **Experience (E)**: In layman's terms, experience is the knowledge a person gets by doing something or observing someone else do it. In the case of machine learning-based tasks, it is the knowledge gained from sample data points provided to the ML algorithm. After getting the data points, the ML algorithm runs iteratively and learns from the inherent pattern. Such learning by the ML algorithm or model is called experience (E), which will be used to solve task T. There are various ways of learning and gaining experience, including supervised, unsupervised, semi-supervised, and reinforcement learning. We will discuss them in the next section.

- **Performance (P)**: How do we know if our ML model, which is supposed to perform a task T and learning or gaining experience E from sample data points over time, is performing well or not? This is where the third component of the ML algorithm comes into the picture. This component is called performance, P, which is a quantitative metric used to measure how well the ML model is performing the task, T, with experience, E. Accuracy score, F1 score, precision, confusion matrix, recall, and specificity are some of the performance metrics we can choose from to measure the performance of our ML model.

Different learning styles in machine learning algorithms

Let's look at the following four different learning styles in ML algorithms.

Supervised learning

Supervised learning methods are the ML methods that are most commonly used. It takes the data sample (usually called training data) and the associated output (usually called labels or responses) with each data sample during the training process of the model. The main objective of supervised learning is to understand the association between input training data and corresponding labels.

Let's understand it with an example. Suppose we have:

- Input variable:
- Output variable:

In order to learn the mapping function from the input to output, we need to apply an algorithm whose main objective is to approximate the mapping function so well that we can also easily predict the output variable (*Y*) for the new input data, as shown in the following example:

$$Y = f(x)$$

These methods are called supervised learning methods because the ML model learns from the training data where the desired output is already known. Logistic regression, k-Nearest neighbors (KNN), Decision tree, and Random Forest are some of the well-known supervised machine learning algorithms.

Based on the type of ML-based tasks, supervised learning methods can be divided into two major classes as follows:

- **Classification**: The main objective of the classification-based tasks is to predict categorical output responses based on the input data that is being provided. The output depends on the ML model's learning in the training phase. Categorical means unordered and discrete values; hence, the output responses will belong to a specific discrete category.

  For example, predicting high-risk patients and discriminating them from low-risk patients is also a classification task. Suppose for newly admitted patients, an emergency room in a hospital measures 12 variables (such as blood sugar, blood pressure, age, weight, and so on). After measuring these variables, a decision is to be taken whether to put the patient in

ICU or not. There is a simple condition that a high priority should be given to the patients who may survive more than a month.

- **Regression**: The main objective of regression-based tasks is to predict continuous numerical output responses based on the input data that is being provided. The output depends on the ML model's learning in the training phase. Similar to classification, with the help of regression, we can predict the output responses for unseen data instances, but that is with continuous numerical output values. Predicting the price of houses is one of the most common real-world examples of regression.

Unsupervised learning

Unsupervised learning methods (as opposed to supervised learning methods) do not require any pre-labeled training data. In such methods, the machine learning model or algorithm tries to learn patterns and relationships from the given raw data without any supervision. Although there are a lot of uncertainties in the result of these models, we can still obtain a lot of useful information like all kinds of unknown patterns in the data and the features that can be useful for categorization.

To make it clearer, suppose we have:

- **Input variable:** ***x***

  There would be no corresponding output variable. For learning, the algorithm needs to discover interesting patterns in data. K-means Clustering, Hierarchical Clustering, and Hebbian Learning are some of the well-known unsupervised machine learning algorithms.

  Based on the type of ML-based tasks, unsupervised learning methods can be categorized into the following broad areas:

- **Clustering**: Clustering, one of the most useful unsupervised machine learning algorithms/methods, is used to find the similarity and relationship patterns among data samples. Once the relationship patterns are found, it clusters the data samples into groups having similar features. The following figure illustrates the working of clustering methods:

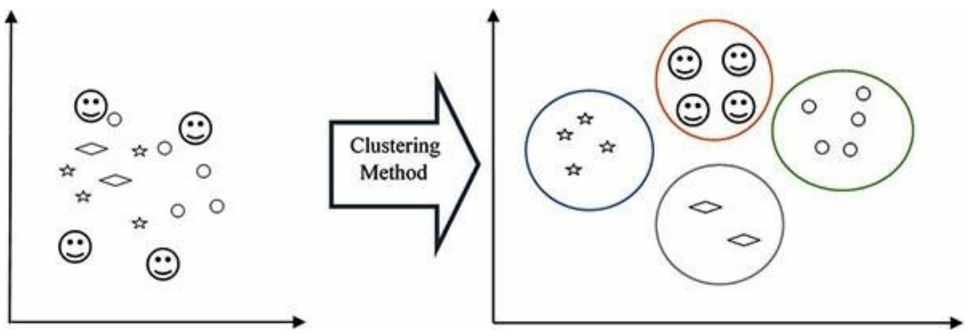


Figure 2.2: *Clustering Method*

- **Association**: One other useful unsupervised machine learning algorithm/method is Association. In order to find patterns representing the interesting relationships between a variety of items, association analyzes a large dataset. For example, analyzing customer shopping patterns comes under association. It is also known as Association Rule Mining or Market Basket Analysis.

- **Anomaly detection**: Sometimes, we need to find out and eliminate the observations that do not occur generally. In that case, the most useful unsupervised ML method is Anomaly detection. It uses learned knowledge to differentiate between anomalous and normal data points. K-means clustering, mean shift clustering, and K-nearest neighbors (KNN) are some of the unsupervised learning algorithms that can detect anomalous data based on its features.
- **Dimensionality reduction**: As the name implies, dimensionality reduction is used to reduce the number of feature variables for every data sample by selecting a principal feature. One of the main reasons behind using the dimensionality reduction method is the problem of feature space complexity (curse of dimensionality). This problem arises when we start analyzing and extracting features, probably millions of features from our data sample. For example, **Principal Component Analysis** (**PCA**), KNN are some of the popular dimensionality reduction methods.

Semi-supervised learning

Semi-supervised machine learning methods fall between supervised and unsupervised machine learning methods. In simple words, they are neither fully supervised nor fully unsupervised. For training, such methods use a small amount of pre-labeled annotated data and lots of unlabeled data. Following are the two approaches that one can follow to implement semi-supervised learning methods:

- **Approach-I**: In this approach, we can first use the small amount of annotated and labeled data to build the supervised model. Once done with the supervised model, we can then apply the same to large amounts of unlabeled data to get more labeled samples. Then, train the model on these labeled samples and repeat the process.
- **Approach-II**: In this approach, we can first use the unsupervised methods to cluster similar data samples and then annotate these groups. Once annotated, we can use them to train the model.

Reinforcement learning

Reinforcement machine learning methods are a bit different from supervised, unsupervised, and semi-supervised machine learning methods. In these kinds of learning algorithms, a trained agent interacts with a specific environment. The job of the agent is to interact with the environment and once observed, it takes actions regarding the current state of that environment. Let’s understand the working of reinforcement learning methods in the following steps:

1. Prepare an agent with some set of strategies.
2. Observe the environment’s current state.
3. Regarding the current state of the environment, select the optimal policy and perform suitable action accordingly.
4. An agent gets a reward or penalty based on the action it took according to the current state of the environment.

5. If needed, update the set of strategies.
6. Repeat the process until the agent learns and adopts the optimal policy.

Popular machine learning algorithms

There are so many ML algorithms that we can feel overwhelming when algorithm names are thrown around us. It is always expected from us to just know what these algorithms are and where they actually fit. So, let's take a tour of various popular ML algorithms and their implementation in the Python programming language.

Linear regression

A statistical model that attempts to model the linear relationship between a dependent variable with a given set of independent and explanatory variables by fitting a linear equation into the observed data is called linear regression. What does the linear relationship between variables mean? It means that with the change (increase or decrease) in the value of one or more independent variables, the value of the dependent variable will also change (increase or decrease) accordingly.

Mathematically, a linear regression line has an equation of the form:

$$Y = m X + b$$

Where Y is the dependent variable and X is the independent or explanatory variable.

The slope of the regression line is m. It represents the effect X has on Y.

b is the Y-intercept. When $X = 0, Y = b$

Types of linear regression

Simple Linear Regression (**SLR**) and **Multiple Linear Regression** (**MLR**) are the two types of linear regressions. Let's learn about them and their implementation using Python.

Simple Linear Regression (SLR)

SLR, the most basic version of linear regression, predicts a response using a single feature. It assumes that the two variables are linearly related.

Implementing simple linear regression in Python:

Let's see how we can implement SLR in the Python programming language. In the following example, we will use a small dataset. We can also use a dataset from the scikit-learn library, which will be used in our next example:

Example-1:

```
#Importing necessary packages

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

```
%matplotlib inline

#Defining a function for calculating values needed for Simple Linear Regression
(SLR)

def coef_estimation(x, y):
  n = np.size(x) #calculating number of observations ‘n’.
  mean_x, mean_y = np.mean(x), np.mean(y) #calculating mean of x and y vectors
  cross_xy = np.sum(y*x) - n*mean_y*mean_x #calculating cross-deviation and
  deviation about x.
  cross_xx = np.sum(x*x) - n*mean_x*mean_x
  reg_b_1 = cross_xy / cross_xx #calculating regression coefficients, i.e., b.
  reg_b_0 = mean_y - reg_b_1*mean_x
  return(reg_b_0, reg_b_1)

#Defining a function for plotting the regression line

def plot_regression_line(x, y, b):
  plt.scatter(x, y, color = "r", marker = "o", s = 20) #plotting actual points as
  scatter plot
  y_pred = b[0] + b[1]*x #predicting response vector
  plt.plot(x, y_pred, color = "g")#plotting the regression line and labels on it
  plt.xlabel('x')
  plt.ylabel('y')
  plt.show()

#Defining the main() function to provide dataset and calling preceding-defined
functions

def main():
  x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10, 11, 12, 13, 14])
  y = np.array([100, 300, 350, 500, 750, 850, 900,950, 1250, 1350, 1400, 1550,
  1600, 1650,1700])
  b = coef_estimation(x, y)
  print("Estimated coefficients:\nreg_b_0 = {} \nreg_b_1 = {}".format(b[0],b[1]))
  plot_regression_line(x, y, b)
if __name___ == "__main__":
     main()
```

We will get the following output for the preceding Python program:

Output:

```
Estimated coefficients:
reg_b_0 = 187.08333333333337
reg_b_1 = 118.03571428571429
```

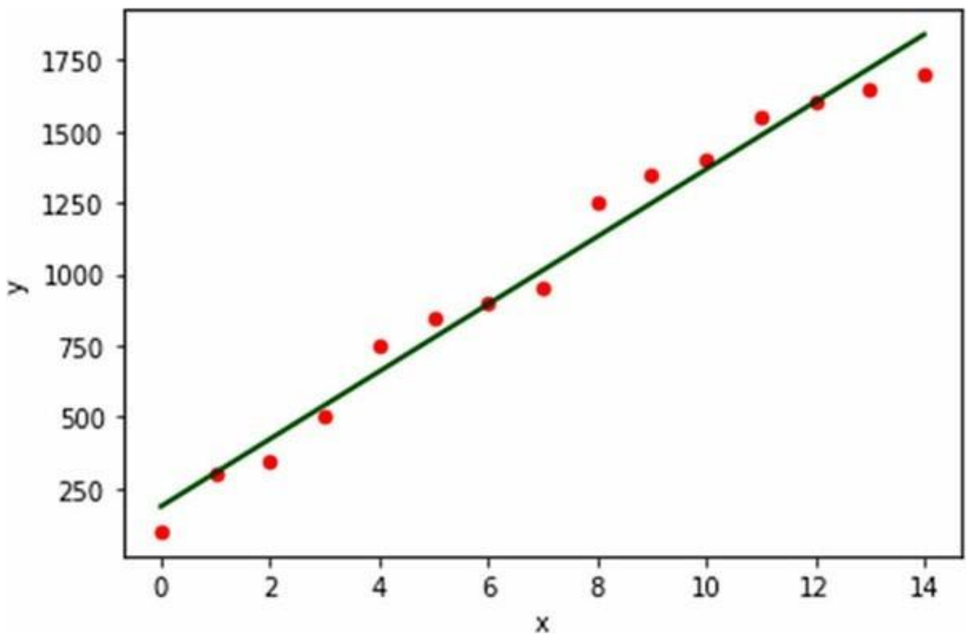


Figure 2.3: *Simple Linear Regression*

Example-2:

In this example, we will implement SLR by using a diabetes dataset from scikit-learn:

```
#Importing necessary packages

import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
%matplotlib inline

#Loading the dataset and creating its object

diabetes_data = datasets.load_diabetes()

#Using one feature

X = diabetes_data.data[:, np.newaxis, 2]

#Splitting the data into training and testing sets

X_train = X[:-35]
X_test = X[-35:]

#Splitting the target into training and testing sets

y_train = diabetes_data.target[:-35]
y_test = diabetes_data.target[-35:]

#Creating linear regression object

SLR_reg = linear_model.LinearRegression()

#Training the model using the training sets

SLR_reg.fit(X_train, y_train)
```

```
#Making predictions by using the testing set
y_pred = SLR_reg.predict(X_test)

# Printing Regression Coefficient, Mean Squared Error(MSE), Variance Score. Also
plotting the regression line and labels on it

print('Coefficients: \n', SLR_reg.coef_)
print("Mean squared error: %.2f"
    %  mean_squared_error(y_test,  y_pred))
print('Variance score: %.2f' % r2_score(y_test, y_pred))
plt.scatter(X_test, y_test, color='red')
plt.plot(X_test, y_pred, color='green', linewidth=3)
plt.xticks(())
plt.yticks(())
plt.show()
```

We will get the following output for the preceding Python program:

Output:

```
Coefficients:
  [963.82249207]
Mean squared error: 3487.66
Variance score: 0.26
```

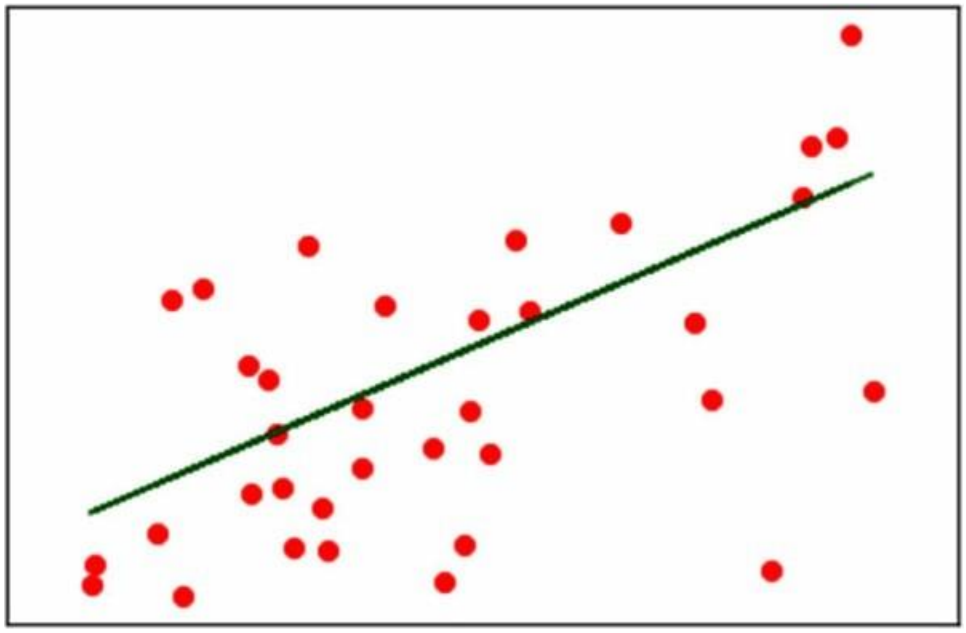


Figure 2.4: *Simple Linear Regression for diabetes dataset*

Multiple Linear Regression (MLR)

MLR, the extension of SLR, predicts a response that is a dependent variable using two or more than two features or independent variables.

Suppose a dataset is having observations and features, then the regression line for these features can be calculated with the help of the following equation:

$$y_i = b_0 + b_1 x_{i1} + b_2 x_{i2} + \ldots + b_p x_{ip}$$

Where Y_i is the predicted response value.

And $\boldsymbol{b_0, b_1, b_2, \ldots b_p}$are regression coefficients.

MLR model also includes the error known as residual error. This changes the preceding calculation as follows:

$$y_i = b_0 0 + b_1 x_{i1} + b_2 x_{i2} + \ldots + b_p x_{ip} + e_i$$

Implementing multiple linear regression in Python:

```
#Importing necessary packages
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model, metrics
%matplotlib inline

#Loading the dataset and creating its object
boston_data = datasets.load_boston(return_X_y=False)

#Defining feature matrix X and response vector Y

X  =  boston_data.data
y = boston_data.target

#Splitting the data into training and testing sets

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.6,
random_state=1)

#Creating regression object and do the training of the model

MLR_reg = linear_model.LinearRegression()
MLR_reg.fit(X_train, y_train)

# Printing Regression Coefficient, and Variance Score. Also plotting the
regression line and labels on it

print('Coefficients: \n', MLR_reg.coef_)
print('Variance score: {}'.format(MLR_reg.score(X_test, y_test)))
plt.style.use('bmh')
plt.scatter(MLR_reg.predict(X_train), MLR_reg.predict(X_train) - y_train,
       color = "green", s = 20, label = 'Train_data')
plt.scatter(MLR_reg.predict(X_test), MLR_reg.predict(X_test) - y_test,
       color = "blue", s = 10, label = 'Test_data')
plt.hlines(y = 0, xmin = 0, xmax = 50, color = 'red', linewidth = 1.25)
plt.legend(loc = 'upper right')
plt.title("Residual errors(eo)")
plt.show()
```

We will get the following output for the preceding Python program.

Output:

```
Coefficients:
  [-7.95572889e-02  7.11808367e-02  5.82382970e-02  1.48237233e+00
  -1.67360287e+01  2.95000985e+00  2.33290549e-02 -1.35721280e+00
  3.13822151e-01 -1.16929875e-02 -8.07436236e-01  6.67075368e-03
  -6.71019667e-01]
Variance score: 0.7325323805669589
```

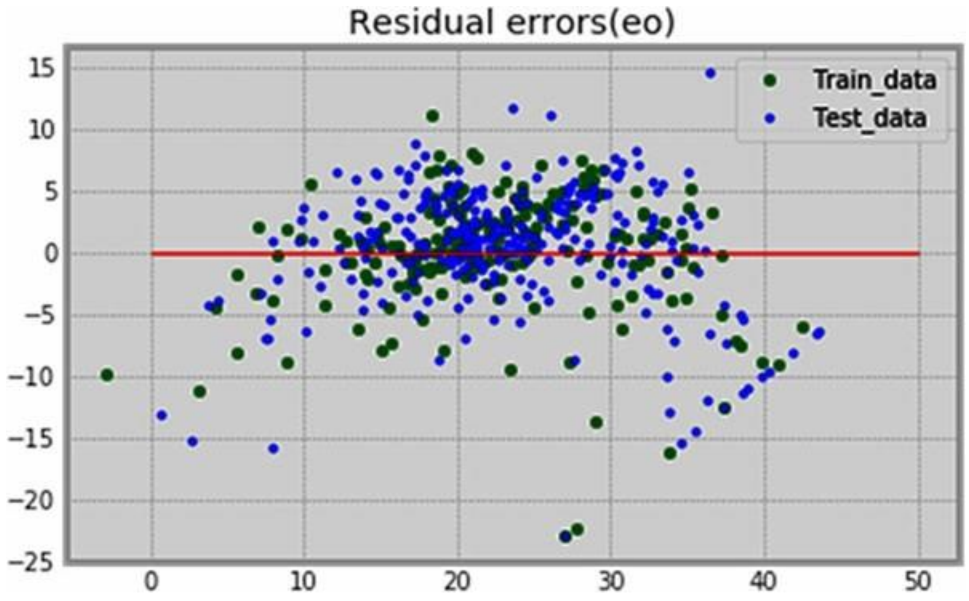


Figure 2.5: *Multiple Linear Regression*

Logistic regression

Logistic regression, a supervised learning classification algorithm, predicts the probability of a dependent variable. Being a classification algorithm, the dependent or target variable can have two possible classes (1 or 0). Here, 1 stands for success/yes and 0 stands for failure or no. In simple words, the target variable is binary in nature. LR is one of the simplest machine learning algorithms, which can be used for various classification problems like diabetes prediction, cancer detection, spam detection, and so on.

In the following example, we will implement logistic regression on digit datasets, which can be downloaded from **`sklearn.datasets`**:

Implementing logistic regression algorithm in Python:

```
#Importing necessary packages

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

#Downloading the digit dataset

from sklearn.datasets import load_digits
digits_dataset = load_digits()

# Printing total images and labels in the dataset

print(digits_dataset.data.shape)
print(digits_dataset.target.shape)
(1797, 64)
```

```
(1797,)
```

The preceding output shows that there are 1797 images (8 by 8 images for a dimensionality of 64) and 1797 labels (integers from 0 to 9).

```
#Let's have a look at the training data

plt.figure(figsize=(20,4))
for index, (image, label) in enumerate(zip(digits.data[0:10],
digits.target[0:10])):
   plt.subplot(1, 10, index + 1)
   plt.imshow(np.reshape(image, (8,8)), cmap=plt.cm.gray)
   plt.title('Training: %i\n' % label, fontsize = 20)
```

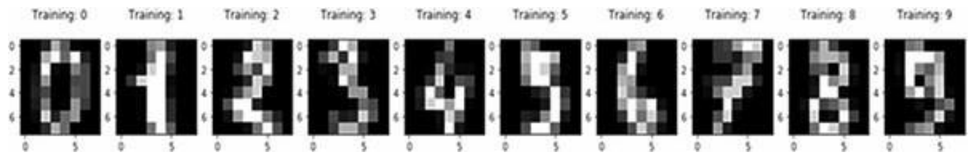


Figure 2.6: *Training data (0-9 digits) for Logistic Regression*

```
#Splitting the dataset into training and testing data set

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(digits.data, digits.target,
test_size=0.30, random_state=0) #70% data for Training and 30% Data for Testing

#Import the LogisticRegression class from sklearn and use the fit method to train
the model

from sklearn.linear_model import LogisticRegression
logRegression = LogisticRegression()
logRegression.fit(x_train, y_train)

#Predicting for images

logRegression.predict(x_test[0].reshape(1,-1))
logRegression.predict(x_test[0:10])
y_pred = logRegression.predict(x_test)

#Calculating performance metrics (Confusion matrix, Classification Report and
Accuracy).

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
print('Confusion Matrix:-\n', confusion_matrix(y_test, y_pred))
print('Classification Report:-\n', classification_report(y_test, y_pred))
print('Accuracy:-\n',accuracy_score(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:-
  [[45  0  0  0  0  0  0  0  0  0]
  [0 47  0  0  0  0  2  0  3  0]
  [0  0 51  2  0  0  0  0  0  0]
  [0  0  1 52  0  0  0  0  0  1]
  [0  0  0  0 48  0  0  0  0  0]
  [0  1  0  0  0 55  1  0  0  0]
  [0  1  0  0  0  0 59  0  0  0]
  [0  1  0  1  1  0  0 50  0  0]
  [0  3  1  0  0  0  0  0 55  2]
  [0  0  0  1  0  1  0  0  2 53]]
Classification Report:-
                precision    recall  f1-score   support
           0       1.00      1.00      1.00        45
           1       0.89      0.90      0.90        52
           2       0.96      0.96      0.96        53
           3       0.93      0.96      0.95        54
           4       0.98      1.00      0.99        48
           5       0.98      0.96      0.97        57
           6       0.95      0.98      0.97        60
           7       1.00      0.94      0.97        53
           8       0.92      0.90      0.91        61
           9       0.95      0.93      0.94        57
    accuracy                           0.95       540
   macro avg       0.96      0.96      0.96       540
weighted avg       0.95      0.95      0.95       540

Accuracy:-
  0.9537037037037037
```

The preceding output that shows our model gives 95.37 % accuracy.

Decision tree algorithm

Decision trees are the most powerful supervised learning classification algorithm. It works based on a tree that has the following two main entities:

- **Decision nodes**: Where the data is split.
- **Leaves**: Where we get the output.

Let's look at the following binary tree that predicts whether a person is fit or not. In order to predict this, we need to provide various information like eating habits, age of the person exercise habits, and so on.

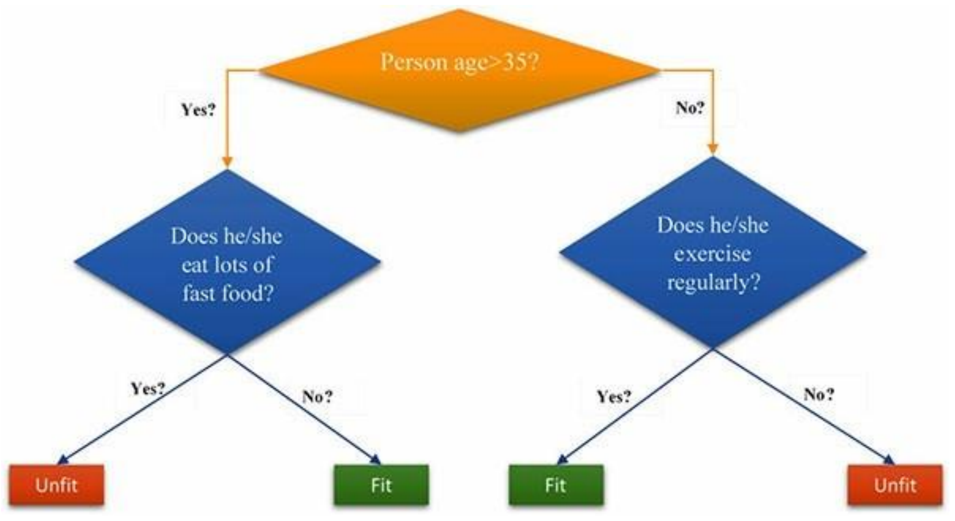

Person age>35?
Yes?
No?
Does he/she eat lots of fast food?
Does he/she exercise regularly?
Yes?
No?
Yes?
No?
Unfit
Fit
Fit
Unfit

Figure 2.7: Decision Tree

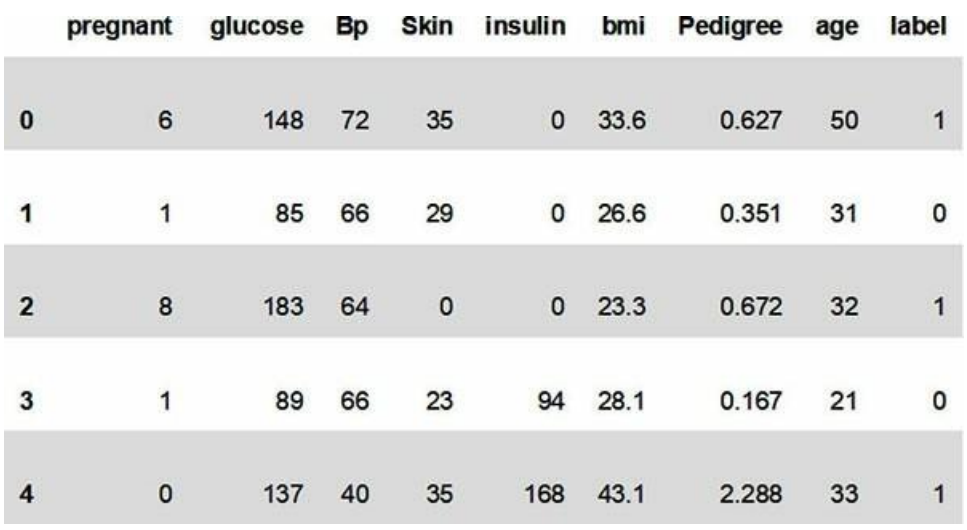
Implementing decision tree algorithm in Python:

Let's see how we can implement a Decision Tree Classifier in the Python programming language. For this, we are going to use the Pima Indians Diabetes dataset. You can download it from **https://archive.ics.uci.edu/ml/datasets/diabetes** and save it to your system.

```
#Importing necessary packages
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
```

```
#Download the Pima-Indians-Diabetes dataset and read it using Pandas as follows

Data_column_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi',
'pedigree', 'age', 'label']
Dataset_pima_diabetes = pd.read_csv(r"C:\Users\Desktop\pima-indians-
diabetes.csv", header=None, names= Data_column_names)
```

```
#With the help of following script, you can look at the dataset

Dataset_pima_diabetes.head()
```



	pregnant	glucose	Bp	Skin	insulin	bmi	Pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Figure 2.8: Pima-Indians Dataset

```
#Splitting the dataset in features and target variables

feature_columns = ['pregnant', 'insulin','bmi',
'age','glucose','bp','pedigree','skin']

#Features

X = Dataset_pima_diabetes[feature_columns]

#Target variable

y = Dataset_pima_diabetes.label

#Splitting the dataset for training and testing purpose. Here we are splitting
the dataset into 80% training data and 20% of testing data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)

#Train the model. We are using DecisionTreeClassifier() class of Scikit-learn
DT_classifier = DecisionTreeClassifier()
DT_classifier = DT_classifier.fit(X_train,y_train)

#Make predictions from trained model

y_pred = DT_classifier.predict(X_test)

#Calculating performance metrics (Confusion matrix, Classification Report and
Accuracy) of our decision tree classifier

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
print('Confusion Matrix:-\n', confusion_matrix(y_test, y_pred))
print('Classification Report:-\n', classification_report(y_test, y_pred))
print('Accuracy:-\n',accuracy_score(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:-
  [[78 21]
  [25 30]]
Classification Report:-
                precision    recall  f1-score   support
           0       0.76      0.79      0.77        99
           1       0.59      0.55      0.57        55
    accuracy                           0.70       154
   macro avg       0.67      0.67      0.67       154
weighted avg       0.70      0.70      0.70       154

Accuracy:-
  0.7012987012987013
```

You can see that the accuracy of our Decision tree classifier is around 70%.

With the help of the following code, we can also visualize the decision tree:

```
#Visualizing our decision tree
import graphviz
from sklearn import tree
dot_data =
tree.export_graphviz(DT_classifier,out_file=None,feature_names=feature_columns,cl
graph = graphviz.Source(dot_data)
graph.render("DTVisualize",view=True)
```

As output, the preceding code will generate a PDF file named DTVisualize.pdf having the decision tree of the Pima-Indians-diabetes dataset. We set the view parameter in a **`graph.render()`** to **`True`**, so that it will open the file as well. If you do not want it to open the file automatically, you can also set it to ‘False’.

Output:

```
'DTVisualize.pdf'
```

Random forest

Random forest, a supervised machine learning classification algorithm, creates decision trees on data samples, and after getting the prediction from each of them, it selects the best solution by means of voting. It reduces overfitting by averaging the result, that's why we get better results as compared with using a single decision tree. The following figure illustrates the working of the Random Forest algorithm:

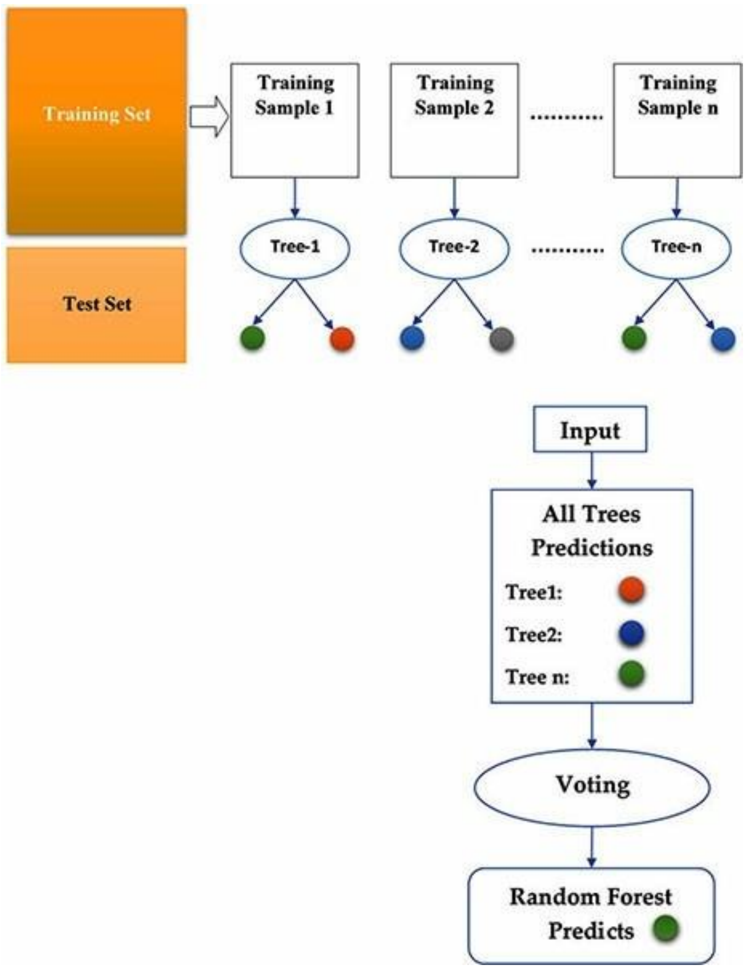


Figure 2.9: *Working of Random Forest Algorithm*

The Random forest starts with the selection of random samples from the dataset. It then constructs a decision tree for every sample and gets the prediction result from all of them. Once we get the predictions, it votes among them and selects the most voted prediction result as the final predicted result.

Implementing Random forest algorithm in Python:

In the following example, we will implement the Random Forest algorithm on the same dataset, that is, the Pima Indians Diabetes dataset, on which we implemented the Decision Tree Classifier. Let's implement it and see the variation in the accuracy result:

```
#Importing necessary packages

import pandas as pd
from sklearn.model_selection import train_test_split

#Download the Pima-IndianDiabetes dataset and read it using Pandas as follows

Data_column_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi',
'pedigree', 'age', 'label']
Dataset_pima_diabetes = pd.read_csv(r"C:\Users\Desktop\pima-indians-
diabetes.csv", header=None, names= Data_column_names)

#Splitting the dataset in features and target variables

feature_columns = ['pregnant', 'insulin','bmi',
'age','glucose','bp','pedigree','skin']
# Features
X = Dataset_pima_diabetes[feature_columns]
# Target variable
y = Dataset_pima_diabetes.label

#Splitting the dataset for training and testing purpose. Here we are splitting
the dataset into 80% training data and 20% of testing data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)

#Train the model. We are using RandomForestClassifier() class of Scikit-learn

RF_classifier = DecisionTreeClassifier()
RF_classifier = RF_classifier.fit(X_train,y_train)

#Make predictions from trained model

y_pred = RF_classifier.predict(X_test)

#Calculating performance metrics (Confusion matrix, Classification Report and
Accuracy) of our decision tree classifier

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
print('Confusion Matrix:-\n', confusion_matrix(y_test, y_pred))
print('Classification Report:-\n', classification_report(y_test, y_pred))
print('Accuracy:-\n',accuracy_score(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:-
  [[89 10]
  [24 31]]

Classification Report:-
               precision    recall  f1-score   support
           0       0.79      0.90      0.84        99
           1       0.76      0.56      0.65        55
    accuracy                           0.78       154
  macro avg       0.77      0.73      0.74       154
 weighted avg       0.78      0.78      0.77       154
```

```
Accuracy:-
0.7792207792207793
```

You can see the accuracy of our random forest classifier is around 78%, which is around 8% higher than the accuracy we achieved by the decision tree classifier.

Naïve Bayes algorithm

Naïve Bayes, one of the simplest supervised learning algorithms, is a statistical classification technique based on applying Bayes' theorem. It assumes that all the predictors are independent of each other, that is, the existence of a feature in a class is independent of the existence of any other feature in a similar class. Such kind of assumption, which is considered naïve, is called class conditional independence.

In Bayesian classification, our main interest is to find the probability of a label given some observed features, that is, the posterior probability. Mathematically, we can express it with the help of the Bayes theorem:

$$P(L \mid features) = \frac{P(L)P(features \mid L)}{P(features)}$$

Where:

- **P(L|features):** Posterior probability of class.
- **P(L)**: Prior probability of class.
- **P(features | L)**: The probability of predictor given class.
- **P(features)**: The prior probability of predictor.

Naïve Bayes models for building Naïve Bayes classifier in Python:

One of the most powerful Python libraries that helps in building Naïve Bayes classifier is Scikit-learn. It has the following three Naïve Bayes models:

- **Gaussian Naïve Bayes**: It assumes that the data from each label is drawn from a simple Gaussian distribution. Gaussian distribution (also known as the normal distribution) is a probability distribution, which is symmetric about the mean. In graph form, it will appear as a bell curve, which shows that the data near the mean are more frequently occurs than the data which is far from the mean.
- **Multinomial Naïve Bayes**: It assumes that the data from each label is drawn from a simple multinomial distribution. Multinomial Naïve Bayes is appropriate for the features representing discrete counts.
- **Bernoulli Naïve Bayes**: It assumes the features to be binary, that is, 0s and 1s. For example, one of the best applications of Bernoulli Naïve Bayes is test classification with the **Bag of Words** (**BoW**) model.

Implementing Naïve Bayes classifier with binary labels (single feature):

In the case of a single feature, the Naïve Bayes classifier calculates the probability of an event with the help of the following steps:

- First, the Naïve Bayes classifier calculates the prior probability for the given class labels.
- Second, it finds likelihood probability with each attribute for each class.
- Once it calculates prior and likelihood probabilities, it puts the values in the Bayes formula and calculates the posterior probability.
- Finally, it will check which class has a higher probability.

For this example, we are using a dummy dataset having the following three columns:

- **Weather**
- **Temperature**
- **Play**

Here **`Weather`** and **`Temperature`** are the features and **`Play`** is the label.

```
#Importing necessary packages

from sklearn import preprocessing #LabelEncoder
from sklearn.naive_bayes import GaussianNB #Gaussian Naive Bayes model

#Assigning features and label variables to the columns of our dataset

weather=['Sunny','Sunny','Overcast','Rainy','Rainy','Overcast','Sunny',
'Sunny','Rainy','Overcast','Rainy']
temp=['Hot','Hot','Hot','Mild','Cool','Cool','Mild','Cool','Mild','Mild','Mild']
play=['N','N','Y','Y','N','N','Y','Y','Y','Y','Y']

#Creating Label Encoder
Lbl_encoder = preprocessing.LabelEncoder()

# Convert string labels into numbers
encode_weatherdata= Lbl_encoder.fit_transform(weather)
encode_temperaturedata= Lbl_encoder.fit_transform(temp)
encode_labeldata= Lbl_encoder.fit_transform(play)

# Printing encoded data
print (“Encoded Weather Data is:”, encode_weatherdata)
print (“Encoded Temperature Data is:”, encode_temperaturedata)
print (“Encoded Label Data is:”, encode_labeldata)

# Combining both the features in a single list of tuples
combn_features=tuple(zip(encode_weatherdata, encode_temperaturedata))
print (“Combined features in a single list of tuples:”, combn_features)

#Generating a Gaussian Classifier
NB_model = GaussianNB()

# Training our model
NB_model.fit(combn_features, encode_labeldata)

#Predicting the Output
predicted= NB_model.predict([[1,0]]) # 1:-Rainy, 0:-Cool
print ("Predicted Value:", predicted)
```

We will get the following output for the preceding Python program.

Output:

```
Encoded Weather Data is: [2 2 0 1 1 0 2 2 1 0 1]
Encoded Temperature Data is: [1 1 1 2 0 0 2 0 2 2 2]
Encoded Label Data is: [0 0 1 1 0 0 1 1 1 1 1]

Combined features in a single list of tuples is: ((2, 1), (2, 1), (0, 1), (1, 2),
(1, 0), (0, 0), (2, 2), (2, 0), (1, 2), (0, 2), (1, 2))

Predicted Value: [0]
```

Implementing Naïve Bayes classifier with multiple labels:

In this example, we will do a multi-class classification in Naïve Bayes. Such kind of classification is used in cases where one needs to classify, for example, a news article about technology, cricket, politics, or the economy.

In the model-building part, we can use the iris-flower dataset, perhaps the best-known database to be found in pattern recognition. You can download it from https://archive.ics.uci.edu/ml/datasets/Iris. Iris-Flower Dataset comprises four features, namely, `sepal length (cm)`, `sepal width (cm)`, `petal length (cm)`, and `petal width (cm)`. This data has three types of iris-flower: `setosa`, `versicolor`, and `virginica`. Here, we can build a model to classify the types of iris flower. This dataset is also available in the Scikit-learn library.

```
#Importing necessary packages
from sklearn import datasets # scikit-learn dataset library
from sklearn.model_selection import train_test_split # train_test_split function
from sklearn.naive_bayes import GaussianNB #Gaussian Naive Bayes model

#Loading iris-flower dataset
iris_flower = datasets.load_iris()

# Printing the names of the features
print ("Features: ", iris_flower.feature_names)

# Printing the label type of flowers
print ("Labels: ", iris_flower.target_names)

# Printing data shape
iris_flower.data.shape

# Printing the iris-flower data features (top 10 records)
print (iris_flower.data[0:10])

# Printing the iris-flower labels (0:setosa, 1:versicolor, 2:virginica)
print (iris_flower.target)

# Splitting iris-dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(iris_flower.data,
iris_flower.target, test_size=0.3,random_state=115) # 70% data for training and
30% data for testing purpose

#Generating a Gaussian Classifier
NB_iris = GaussianNB()

#Training the classifier model using the training sets
NB_iris.fit(X_train, y_train)

#Predicting the response for testing dataset
y_pred = NB_iris.predict(X_test)
```

```
#Importing the scikit-learn metrics module for calculating the accuracy
from sklearn import metrics
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

We will get the following output for the preceding Python program.

Output:

```
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
width (cm)']
Labels: ['setosa' 'versicolor' 'virginica']

(150, 4)

[[5.1 3.5 1.4 0.2]
  [4.9 3.  1.4 0.2]
  [4.7 3.2 1.3 0.2]
  [4.6 3.1 1.5 0.2]
  [5.  3.6 1.4 0.2]
  [5.4 3.9 1.7 0.4]
  [4.6 3.4 1.4 0.3]
  [5.  3.4 1.5 0.2]
  [4.4 2.9 1.4 0.2]
  [4.9 3.1 1.5 0.1]]
Accuracy: 0.9333333333333333
```

The preceding output shows that the accuracy of our classifier is around 93%.

Support Vector Machine (SVM)

A **Support Vector Machine** (**SVM**), a supervised ML classification algorithm, is a powerful yet flexible algorithm. They were first introduced in the 1960s, but in the 1990s, they also got refined. SVM can handle multiple continuous and categorical variables. That's the reason, SVMs are becoming extremely popular nowadays.

Working of SVM algorithm

An SVM model is basically a representation of numerous classes in a hyperplane, generated iteratively by SVM, in a multidimensional space. One of the most important goals of SVM is to find a **Maximum Marginal Hyperplane** (**MMH**) and for this, it divides the datasets into classes. This is illustrated in the following figure:

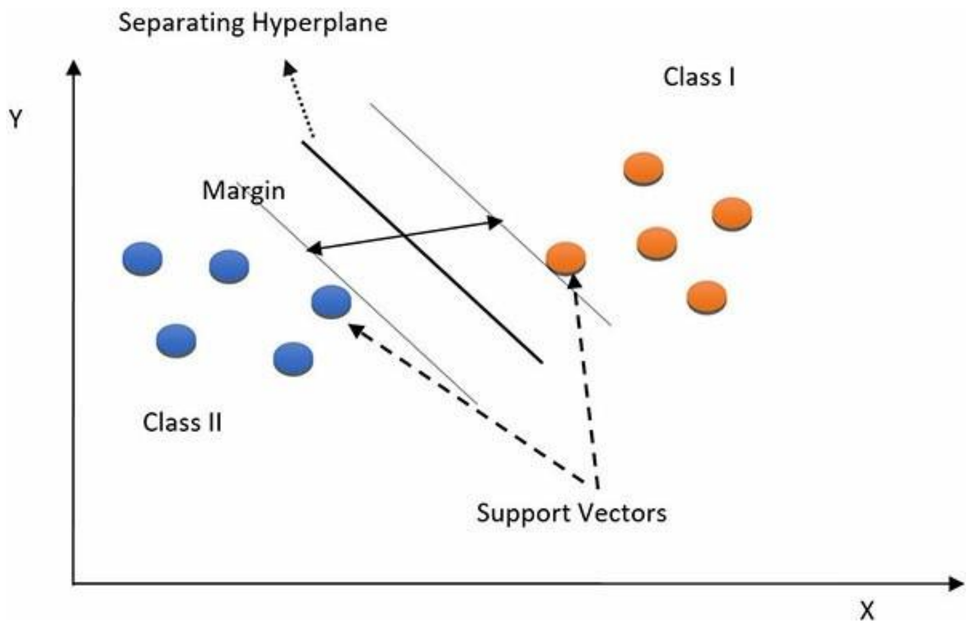


Figure 2.10: *Support Vector Machine*

Let’s have a look at some of the important points in SVM:

- **Support Vectors**: Support vectors are the data points that are closest to the hyperplane.
- **Margin**: As shown in the preceding figure, it is the gap between two lines on the closest data points of different classes. We can calculate margins as the perpendicular distance from the line to the SVs (support vectors).
- **Hyperplane**: It is the space or decision plane divided between a set of objects having different classes.

Implementing SVM algorithm in Python:

Let’s see how we can implement SVM in Python programming language. For this, we are going to generate a sample dataset, from **`sklearn.dataset.sample_generator`**, having linearly separable data.

```
#Importing necessary packages
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
import seaborn as sns; sns.set()

#Generating sample dataset having linearly separable data
from sklearn.datasets.samples_generator import make_blobs
X_data, y_data = make_blobs(n_samples=500, centers=2, random_state=0,
cluster_std=0.30)
plt.scatter(X_data[:, 0], X_data[:, 1], c=y_data, s=30, cmap='winter');
```

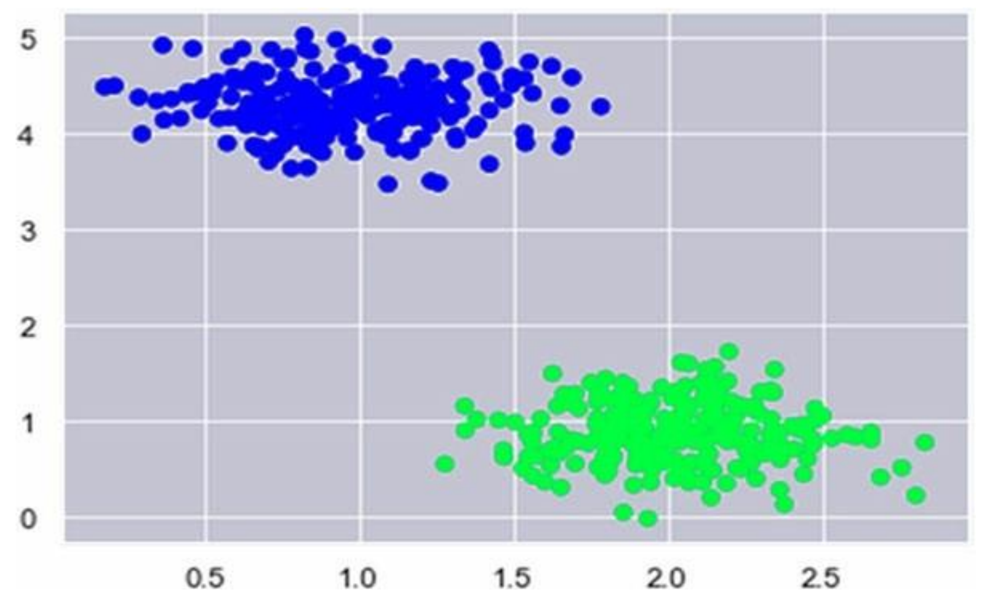


Figure 2.11: *Sample Dataset having linearly separable dataset*

The preceding output is having sample dataset with 500 samples and 2 clusters.

```
# Implementing discriminative classification i.e. Dividing the classes from each
other
xfit = np.linspace(-1, 3.5)
plt.scatter(X_data[:, 0], X_data[:, 1], c=y_data, s=30, cmap='winter')
for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
  plt.plot(xfit, m * xfit + b, '-k')
plt.xlim(-1, 3.5);
```

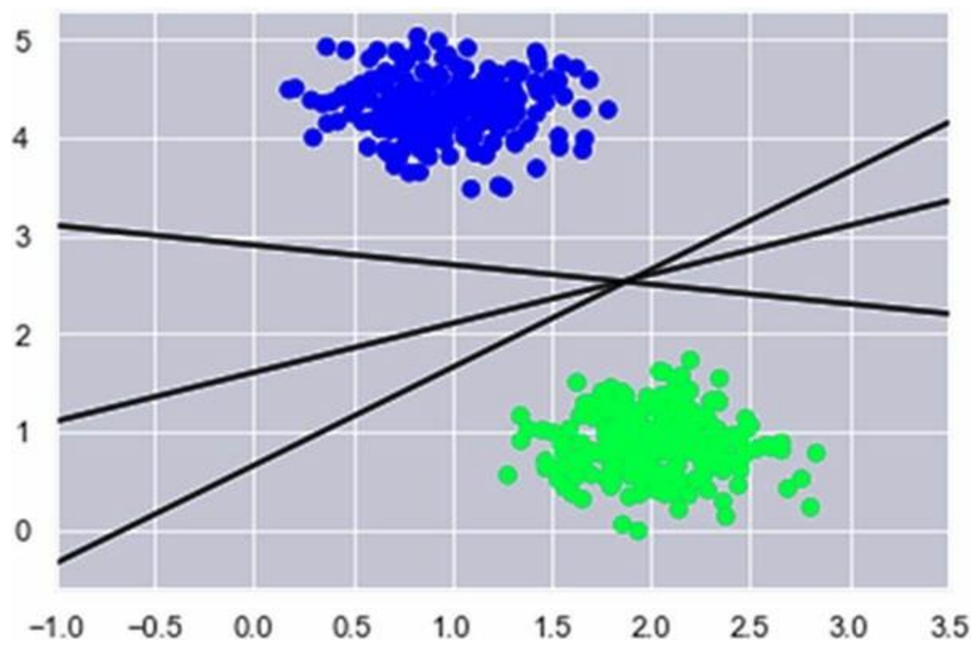


Figure 2.12: *Discriminative Classification*

```
# In order to find MMH, drawing a margin of some width, up to the nearest point,
around each line
```

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X_data[:, 0], X_data[:, 1], c=y_data, s=30, cmap='winter')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
  yfit = m * xfit + b
  plt.plot(xfit, yfit, '-k')
  plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none', color='#AAAAAA',
  alpha=0.4)
plt.xlim(-1, 3.5);
```

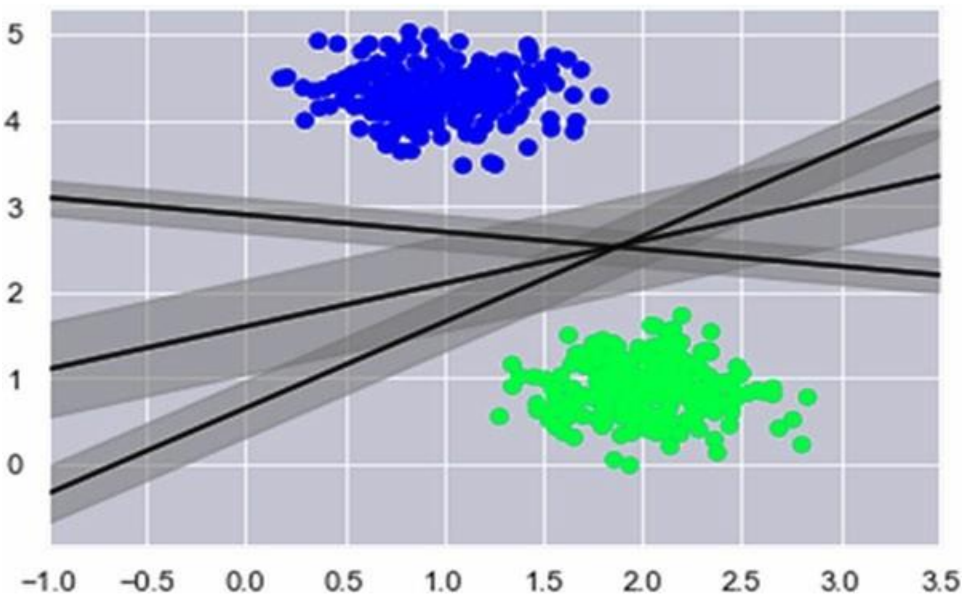


Figure 2.13: *Discriminative Classification with margin of width*

```
#Training the algorithm. Calling fit method of SVC class
from sklearn.svm import SVC # "Support vector classifier"
model = SVC(kernel='linear')
model.fit(X_data, y_data)

#Plotting the decision function for 2-dimensional SVC

def decision_function(model, ax=None, plot_support=True):

  if ax is None:
      ax = plt.gca()
  xlim = ax.get_xlim()
  ylim = ax.get_ylim()

  x = np.linspace(xlim[0], xlim[1], 30)
  y = np.linspace(ylim[0], ylim[1], 30)
  Y, X = np.meshgrid(y, x)
  xy = np.vstack([X.ravel(), Y.ravel()]).T
  P = model.decision_function(xy).reshape(X.shape)
  ax.contour(X, Y, P, colors='k',
         levels=[-1, 0, 1], alpha=0.5,
         linestyles=['--', '-', '--'])
  if                            plot_support:
      ax.scatter(model.support_vectors_[:, 0],
           model.support_vectors_[:, 1],
           s=400, linewidth=2, facecolors='none');
  ax.set_xlim(xlim)
  ax.set_ylim(ylim)

plt.scatter(X_data[:, 0], X_data[:, 1], c=y_data, s=30, cmap='winter')
decision_function(model);
```

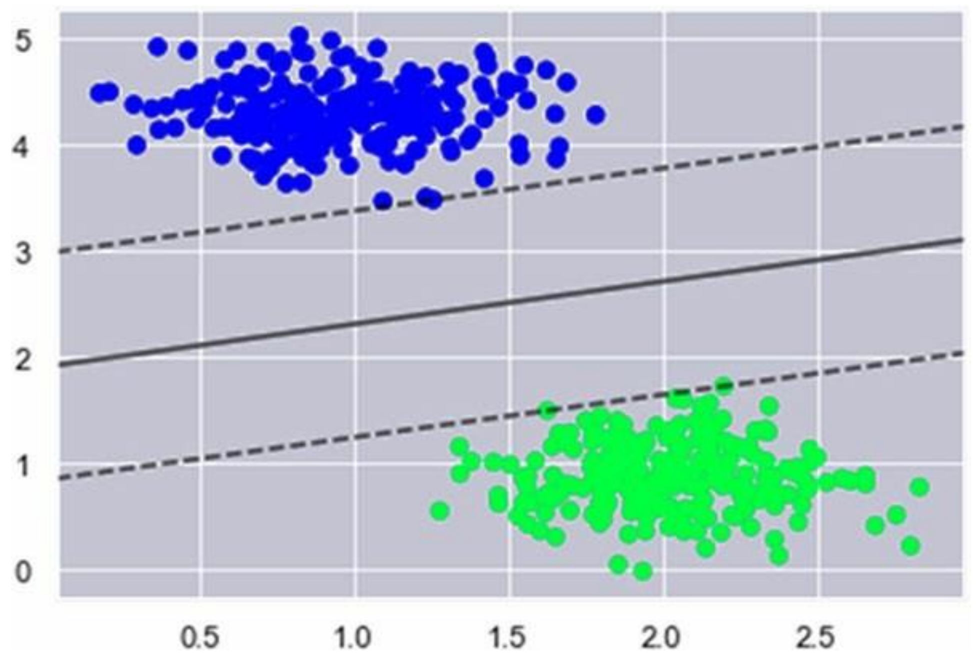


Figure 2.14: *Decision function for 2-D SVC*

```
# Getting the support vectors points
model.support_vectors_

array([[1.2591839, 3.48188418],
    [1.62869156, 1.49705048],
    [2.19960206, 1.72547019]])
```

Kernel SVM

In the preceding example, we have implemented SVM that finds decision boundaries for linearly separable data. But, in the case of non-linearly separable data, the preceding SVM algorithm cannot be used. For that, we need to use a modified version of SVM called Kernel SVM. In the following implementation example, we will be using different kernels, polynomial, **Radial Basis Function** (**RBF**), and sigmoid on the Iris-flower dataset.

Implementing Kernel SVM in Python:

```
#Importing necessary packages
from sklearn import datasets # scikit-learn dataset library
from sklearn.model_selection import train_test_split # train_test_split function

#Loading iris-flower dataset
iris_flower = datasets.load_iris()

# Printing the names of the features
print ("Features: ", iris_flower.feature_names)

# Printing the label type of flowers
```

```
print ("Labels: ", iris_flower.target_names)

# Printing data shape
iris_flower.data.shape

# Printing the iris-flower data features (top 10 records)
print (iris_flower.data[0:10])

# Printing the iris-flower labels (0:setosa, 1:versicolor, 2:virginica)
print (iris_flower.target)

# Splitting iris-dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(iris_flower.data,
iris_flower.target, test_size=0.3,random_state=115) # 70% data for training and
30% data for testing purpose

# Training the algorithm by using SVC class fit method. Here we are using
Polynomial kernel
from sklearn.svm import SVC
svm_K_classifier = SVC(kernel='poly', degree=8)
svm_K_classifier.fit(X_train, y_train)

#Predicting the response for testing dataset
y_pred = svm_K_classifier.predict(X_test)

# Importing the scikit-learn classification_report and confusion_matrix module
for evaluating the algorithm
from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix:\n', confusion_matrix(y_test,y_pred))
print('Classification Report:\n',classification_report(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:
  [[18 0 0]
  [0 10 0]
  [0 3 14]]

Classification Report:
                precision    recall  f1-score   support
            0       1.00      1.00      1.00        18
            1       0.77      1.00      0.87        10
            2       1.00      0.82      0.90        17
     accuracy                           0.93        45
    macro avg       0.92      0.94      0.92        45
weighted avg       0.95      0.93      0.93        45
```

Similarly, we can train the algorithm by using the **Radial Basis Function** (**RBF**) kernel:

```
# Training the algorithm by using SVC class fit method. Here we are using rbf
kernel
from sklearn.svm import SVC
svm_K_classifier = SVC(kernel='rbf')
svm_K_classifier.fit(X_train, y_train)

#Predicting the response for testing dataset
y_pred = svm_K_classifier.predict(X_test)
```

```
# Importing the scikit-learn classification_report and confusion_matrix module
for evaluating the algorithm
from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix:\n', confusion_matrix(y_test,y_pred))
print('Classification Report:\n',classification_report(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:
  [[18 0 0]
  [0 10 0]
  [0 2 15]]
Classification Report:
                precision    recall  f1-score   support
            0       1.00      1.00      1.00        18
            1       0.83      1.00      0.91        10
            2       1.00      0.88      0.94        17
     accuracy                           0.96        45
    macro avg       0.94      0.96      0.95        45
weighted avg       0.96      0.96      0.96        45
```

Now, train the algorithm by using Sigmoid kernel:

```
# Training the algorithm by using SVC class fit method. Here we are using rbf
kernel
from sklearn.svm import SVC
svm_K_classifier = SVC(kernel='sigmoid')
svm_K_classifier.fit(X_train, y_train)

#Predicting the response for testing dataset
y_pred = svm_K_classifier.predict(X_test)

# Importing the scikit-learn classification_report and confusion_matrix module
for evaluating the algorithm
from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix:\n', confusion_matrix(y_test,y_pred))
print('Classification Report:\n',classification_report(y_test,y_pred))
```

We will get the following output for the preceding Python program:

Output:

```
Confusion Matrix:
  [[0 18 0]
  [0 10 0]
  [0 17 0]]

Classification Report:
                precision    recall  f1-score   support
            0       0.00      0.00      0.00        18
            1       0.22      1.00      0.36        10
            2       0.00      0.00      0.00        17
     accuracy                           0.22        45
    macro avg       0.07      0.33      0.12        45
weighted avg       0.05      0.22      0.08        45
```

k-Nearest Neighbor (kNN)

This is a supervised machine learning algorithm for classification and regression but mainly used for classification. To predict the values of new data points, KNN uses feature similarity. The new data points will be assigned a value, which is based on how closely it matches the points in the training set. Let's understand its working in the following steps:

1. First, we need to provide KNN the dataset, that is, the training and test data.
2. Second, we must provide the value of k (it can be any integer), that is, the nearest data points to the KNN algorithm.
3. Finally, for every point in the test data, the KNN algorithm will do the calculation as follows:

    1. It calculates the distance between test data and each row of training data. It can use any of the methods, Euclidean, Manhattan, or Hamming distance. But it generally uses the Euclidean function. Let's understand these distances:

        - **Euclidean distance**: It calculates the distance between two real-valued vectors. Mostly we use it to calculate the distance between two rows of data having numerical values (floating or integer values). The following is the formula to calculate Euclidean distance:

          $$d(r,s) = \sqrt{\sum_{i=1}^{n}(s_i - r_i)^2}$$

          Here:

          r and *s* are the two points in Euclidean *n*-space.

          s_i and r_i are Euclidean vectors.

          n denotes the *n*-space.

        - **Hamming distance**: It calculates the distance between two binary vectors. Mostly we find the binary strings when we use one-hot encoding on categorical columns of data. In one-hot encoding, the integer variable is removed and a new binary variable will be added for each unique integer value. For example, if a column had the categories say `Length`, `Width`, and `Breadth`. We might one-hot encode each example as a bitstring with one bit for each column as follows:

            - Length = [1, 0, 0]
            - Width = [0, 1, 0]
            - Breadth = [0, 0, 1]

          The Hamming distance between any of the preceding two categories mentioned can be calculated as the sum or the average number of bit differences between the two binary strings. We can see that the Hamming difference between length and width categories is about 2/3 and 0.666 because 2 out of 3 positions are different.

- **Manhattan distance**: The Manhattan distance, also known as the City Block distance, is calculated as the sum of absolute differences between the two vectors. It is mostly used for the vectors that describe objects on a uniform grid such as a city block or chessboard. The following is the generalized formula to calculate Manhattan distance in n-dimensional space:

  $$D = \sum_{i=1}^{n} |r_i - s_i|$$

  Here,

  s_i and r_i are data points.

  n denotes the *n*-space.

4. Once calculated the distance, KNN now sorts the distance values in ascending order and chooses the top K rows from this sorted array.
5. Now, based on the most frequent class of these rows, a class to the test point will be assigned.

Implementing KNN Classifier in Python

In this example, we will be implementing the KNN classifier in Python. We will be using the Iris-Flower dataset:

```
#Importing necessary packages
import numpy as np
import pandas as pd

#Downloading the Iris-flower dataset
path = https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
# Assigning column names to the dataset
column_names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width',
'Class']

# Reading dataset to Pandas dataframes
iris_data = pd.read_csv(path, names= column_names)

# Data preprocessing
X = iris_data.iloc[:, :-1].values
y = iris_data.iloc[:, 4].values

# Splitting the dataset into train and test data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25) # 75%
data for training and 25% data for testing purpose

# Data scaling for sending scaled data to the train the model
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Training the algorithm by using KNeighborsClassifier class fit method
from sklearn.neighbors import KNeighborsClassifier
Knn_classifier = KNeighborsClassifier(n_neighbors=8)
```

```
Knn_classifier.fit(X_train, y_train)

#Predicting the response for testing dataset
y_pred = Knn_classifier.predict(X_test)

# Importing the scikit-learn classification_report and confusion_matrix module
for evaluating the algorithm
from sklearn.metrics import classification_report, confusion_matrix
print('Confusion Matrix:\n', confusion_matrix(y_test,y_pred))
print('Classification Report:\n',classification_report(y_test,y_pred))
```

We will get the following output for the preceding Python program.

Output:

```
Confusion Matrix:
  [[12 0 0]
  [0 16 0]
  [0 2 8]]

Classification Report:
                precision    recall  f1-score   support
     Iris-setosa       1.00      1.00      1.00        12
Iris-versicolor       0.89      1.00      0.94        16
  Iris-virginica       1.00      0.80      0.89        10
        accuracy                           0.95        38
       macro avg       0.96      0.93      0.94        38
    weighted avg       0.95      0.95      0.95        38
```

Implementing KNN regressor in Python

In this example, we will implement the KNN regressor in Python. The steps are almost same as we used while implementing the KNN classifier in Python.

```
#Importing necessary packages
import numpy as np
from sklearn.datasets import load_iris
iris_data = load_iris()
X = iris_data.data[:, :4]
y = iris_data.target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

#This is the only change i.e. we need to import KNeighborRegressor class and
train the model
from sklearn.neighbors import KNeighborsRegressor
knn_regressor = KNeighborsRegressor(n_neighbors=5)
knn_regressor.fit(X_train, y_train)

# Printing the Mean Square Error
print ("The MSE is:",format(np.power(y-knn_regressor.predict(X),4).mean()))
```

We will get the following output for the preceding Python program.

Output:

```
The MSE is: 5.666666666666667
```

K-Means clustering

K-means clustering, one of the simplest unsupervised machine learning algorithms, groups similar data points together and discovers underlying patterns. *k* in K-means refers to the fixed number of clusters in a dataset. While computing the centroids, the k-means algorithm assigns data points in such a manner that the sum of the squared distance between the data points and centroid would be minimum.

The way the k-means clustering algorithm works is shown in the following steps:

1. Specify the number of clusters, that is, *k*.
2. It shuffles the dataset to initialize centroids and then randomly selects *k* data points for the centroids.
3. Next, it keeps iterating the following until it finds the optimal centroid, that is, there is no change to the centroids:
    1. First, it calculates the sum of squared distance between data points and centroids.
    2. Second, it assigns each data point to the nearest cluster.
    3. Finally, it takes the average of all data points of clusters to compute the centroids for those clusters.

Implementing k-means clustering in Python

In this example, we will implement k-means clustering on the 2-dimensional dataset sample, which we generated on 700 sample values, having 5 clusters.

```
#Importing necessary packages
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
from sklearn.cluster import KMeans

#Generating 2-dimensional dataset having 5 blobs
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=700, centers=5, cluster_std=0.8)

#Let’s visualize our dataset
plt.scatter(X[:, 0], X[:, 1], s=10);
plt.show()
```

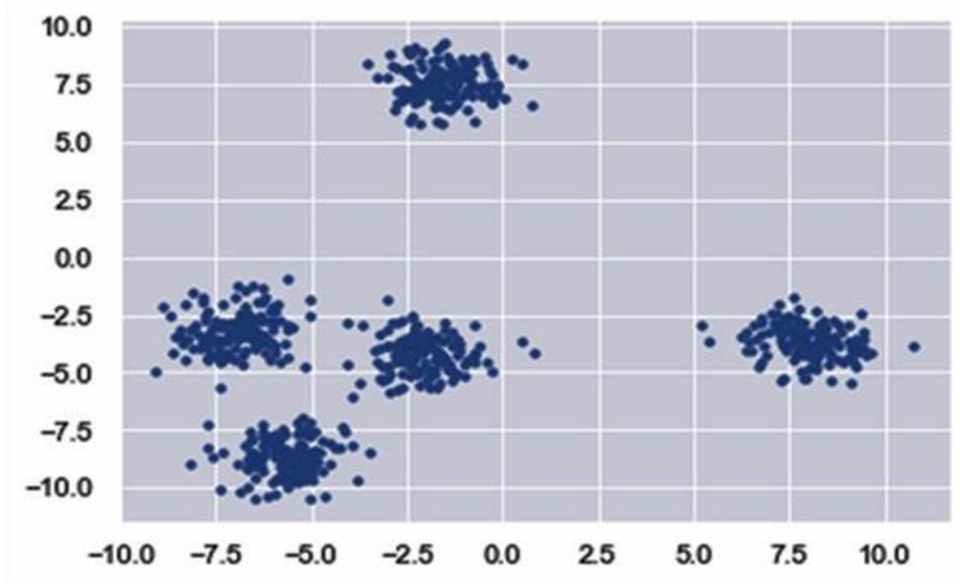


Figure 2.15: *2-D Data having 5-blobs*

```
#Creating an object of k-means and providing number of clusters
kmeans = KMeans(n_clusters=5)

#Train the model by using fit method of k-means() class
kmeans.fit(X)

# do the predictions
y_kmeans = kmeans.predict(X)

# Plotting and visualizing cluster’s centers picked by k-means estimators
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=10, cmap='rainbow')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black');
plt.show()
```

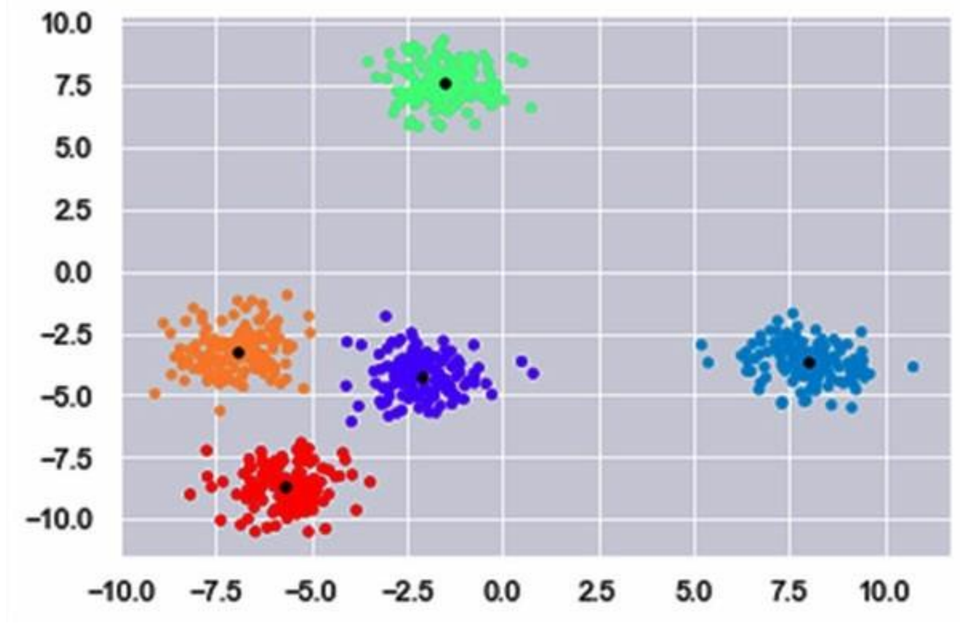


Figure 2.16: *Cluster centers picked by k-means estimators*

Conclusion

In this chapter, we learned the basics of Machine Learning and explored various components associated with it. We also understood different learning styles, namely, supervised, unsupervised and semi-supervised, and reinforcement used in machine learning algorithms. We learned that the main objective of supervised learning, composed of subfields such as classification and regression, is to learn an association between input training data and corresponding labels. While classification models allow us to predict categorical output responses for the given input data, we can use the regression model to predict continuous numerical output responses for the given input data.

We understood that the unsupervised learning methods do not require any pre-labeled training data and it learns patterns and relationships from given raw data without any supervision. Classification, Dimensionality Reduction, Anomaly Detection, and Association are the subfields of unsupervised learning. We also explored that semi-supervised machine learning methods neither fully supervised nor fully unsupervised use a small amount of pre-labeled annotated data and lots of unlabeled data for training purposes. In reinforcement learning, the agent interacts with the environment and acts according to the current state of the environment.

We described some popular machine learning algorithms such as Linear Regression, Logistic Regression, Decision Tree, Random Forest, **k-Nearest Neighbor** (**kNN**), k-means clustering, Naïve Bayes, and **Support Vector Machine** (**SVM**). Along with their descriptions, these algorithms have also been implemented using the Scikit-learn Python library for machine learning. We implemented these algorithms in Jupyter Notebook.

We went over the roadmap for applying machine learning to real-world problems, which we will use as a foundation of deeper discussions and hands-on examples in the upcoming chapters.

Questions

1. What is Machine Learning (ML)? What are the various components a machine learning algorithm has?
2. Explain different learning styles in the machine learning algorithms.
3. What is Logistic Regression? Implement it in the Python programming language.
4. What is Multiple Linear Regression? How can we implement it in the Python programming language?
5. What is the difference between the Decision Tree and the Random Forest algorithm? Find their accuracy after implementing both on the same dataset in the Python programming language.
6. Briefly explain the working of k-nearest neighbor (KNN) and K-means clustering algorithm.

---

[1] Tom M Mitchell et al. "Machine learning. 1997". In: Burr Ridge, IL: McGraw Hill 45.37 (1997), pp. 870–877.

CHAPTER 3

Classification and Regression Using Supervised Learning

Introduction

Artificial intelligence, especially **Machine Learning** (**ML**), is continuously growing and set to be the most transformative technology existing over the next decade. Self-driving vehicles, fraud detection systems, tumor detection, and instant machine translation are a few of the advancements and applications of ML that have already started having an impact on society. Machine learning algorithms started stepping into every aspect of our lives too. They have already become a part of our daily routines. From voice-enabled personal assistants like Alexa, Siri, Google Assistant, and Cortana to optimized music, movies, and news recommendations to suggestive searches, everything we use is directly or indirectly affected by ML.

In the previous chapter, you got a brief overview of ML and various learning styles like supervised, unsupervised, semi-supervised, and reinforcement in ML algorithms. This chapter provides details about supervised ML tasks: Classification and Regression. It also addresses various steps to build a Classifier and Regressor using the Python programming language. You will also get a glimpse of the best evaluation metrics to check the performance of both classification and regression algorithms.

Structure

In this chapter, we will discuss the following topics:

- Classification
- Various steps to build a classifier using Python
- Performance metrics for classification
    - Confusion matrix
    - Accuracy
    - Precision
    - Recall
    - Specificity
    - F1 score
- Regression
- Various steps to build a regressor using Python
- Performance metrics for regression

    - **Mean Absolute Error** (**MAE**)
    - **Mean Squared Error** (**MSE**)
    - R2 – the coefficient of determination

Objectives

After studying this chapter, you should be able to build a classifier and regressor using the Python programming language. You will also learn about various performance metrics used to evaluate classification and regression models.

Classification

Classification, a sub-field of supervised ML, may be defined as the process of predicting classes or categorical output responses based on the input data that is being provided. The output depends upon the ML model's learning in the training phase. Categorical means unordered and discrete values; hence, the output responses will belong to a specific discrete category.

Let's understand it with an example of weather prediction. We will keep it simple and try to predict the weather among two categories, sunny or rainy. This prediction will be based on multiple data samples that consist of the following attributes:

- Humidity
- Temperature
- Atmospheric pressure
- Precipitation
- Wind

This classification can be termed as a binary classification problem because there are only two distinct classes, sunny and rainy. The following figure depicts this task:

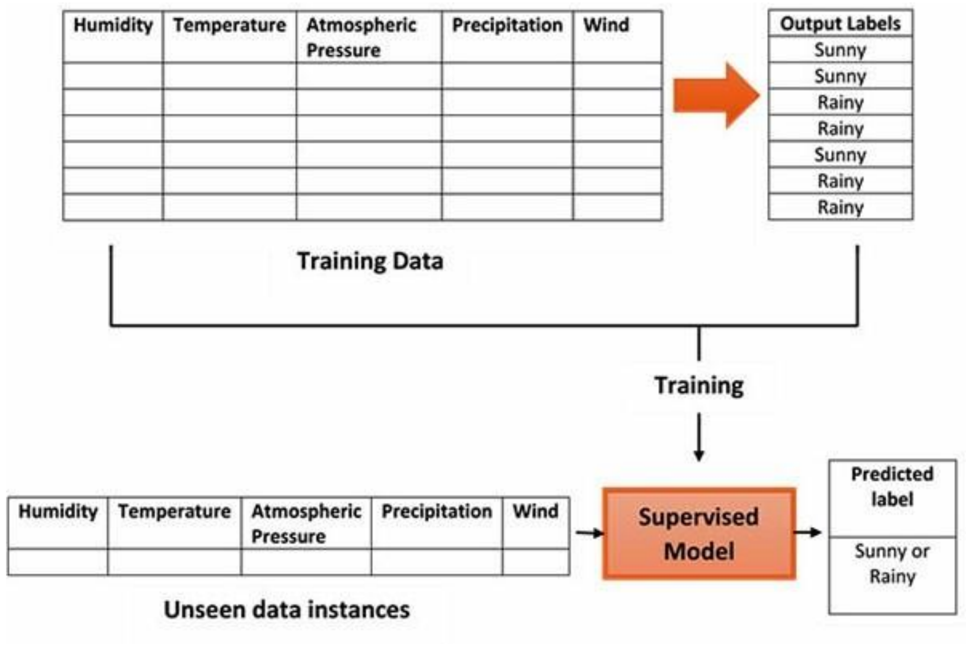


Figure 3.1: *Classification for weather prediction*

The preceding example represents a binary classification, but in case if there are more than two distinct classes, it becomes a multi-class classification task. In such classification problems, the prediction response can be one among the probable set of classes. For example, predicting numeric digits from scanned handwritten images is a multi-class classification problem because the output class label for any image can be any digit (0–9). The following figure will depict the difference between binary and multi-class classification.

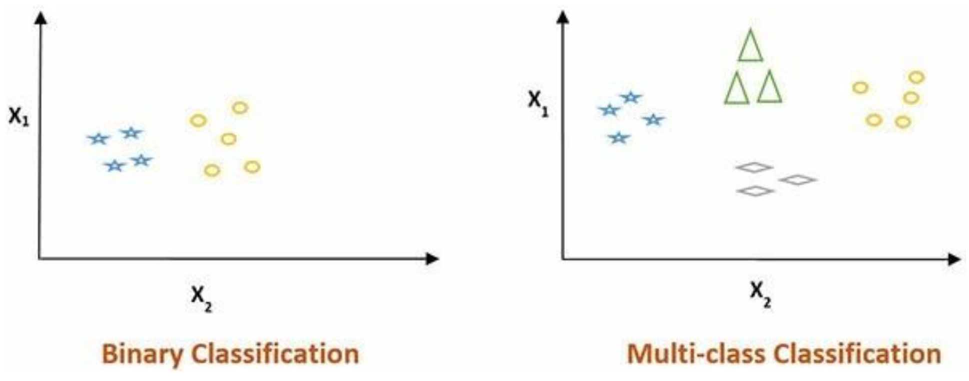


Figure 3.2: *Binary versus multi-class classification*

Various steps to build a classifier using Python

In the previous section, we understood classification with an example. Now let us build a classifier in the Python programming language. Following are the various steps to do so:

Step 1 – Import ML library

To start building a classifier in Python, we need to have an ML library. Here, by using the following Python command, we will import Scikit-learn[1] – an open-source machine learning library:

```
import sklearn
```

Step 2 – Import dataset

We know that to train an ML model, we always need a dataset. In this step, we will import the sklearn's Breast Cancer Wisconsin Diagnostic Dataset[2], which is widely used for classification purposes and containing 569 instances and information on 30 features. The classification labels used in this dataset are, namely, malignant or benign. We can use the following command to import it from Scikit-learn:

```
from sklearn.datasets import load_breast_cancer
```

Once imported, we can load the dataset in a variable name so that we can use it throughout our program. The following is the command to load this dataset in a variable named `data_cancer`:

```
data_cancer = load_breast_cancer()
```

Sklearn's *Breast Cancer Wisconsin Diagnostic Dataset* has an important set of information, namely, Classification label names, The actual labels, Feature names, and Attributes. We will give new variables, namely, `target_names`, `target`, `feature_names`, and `data`, respectively, for each of them as follows:

```
labelnames = data_cancer['target_names']
labels  = data_cancer['target']
featurenames = data_cancer['feature_names']
features = data_cancer['data']
```

With the help of the following command, you can print this set of information:

```
print(labelnames)
```

Output:

```
['malignant' 'benign']
```

The preceding output shows that the dataset has two classification labels, namely, `malignant` and `benign`.

```
print(labels)
```

Output:

```
  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 1 0 0
   1 0 1 0 0 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1 1 1 0 1 1 0 1 1
   1 1 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1
   1 1 1 1 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 1 0
   1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1
   1 0 1 1 1 1 1 0 0 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1
   1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 1
```

```
 1 1 0 1 0 1 0 1 1 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0 0
 0 1 0 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1
 1 0 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 0 1 1 1 1 1 0 1 1
 0 1 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1
 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 1 0 1 0 1 1 1 1 1 0 1 1 0 1 0 1 0 0
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 0 0 0 0 0 0 1]
```

The preceding output shows that the classification labels are mapped to `0` and `1`, where `0` represents `malignant` and `1` represents `benign`.

```
print(featurenames)
```

Output:

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

```
print(features)
```

Output:

```
[[1.799e+01 1.038e+01 1.228e+02 … 2.654e-01 4.601e-01 1.189e-01]
 [2.057e+01 1.777e+01 1.329e+02 … 1.860e-01 2.750e-01 8.902e-02]
 [1.969e+01 2.125e+01 1.300e+02 … 2.430e-01 3.613e-01 8.758e-02]
 …
 [1.660e+01 2.808e+01 1.083e+02 … 1.418e-01 2.218e-01 7.820e-02]
 [2.060e+01 2.933e+01 1.401e+02 … 2.650e-01 4.087e-01 1.240e-01]
 [7.760e+00 2.454e+01 4.792e+01 … 0.000e+00 2.871e-01 7.039e-02]]
```

Step 3 – Organizing data-training and testing set

The accuracy of an ML model can be measured by testing that model on new unseen data. For that purpose, it is important to split the dataset into two parts, namely, training and testing set. Let us see how we can divide our dataset into 70% training set and remaining 30% into testing set:

```
from sklearn.model_selection import train_test_split
train, test, train_labels, test_labels =
train_test_split(features,labels,test_size = 0.30, random_state = 0)
```

Step 4 – Creating ML model

In the previous step, we divided our dataset into 70% training and 30% testing set. Now let us create our ML model with the help of the Naïve Bayes classifier. The command is given as follows:

```
from sklearn.naive_bayes import GaussianNB #import the Gaussian Naïve Bayes model
NB_Gaussian = GaussianNB() #initialize the model
```

Step 5 – Train the model

To use the previously created ML model for testing purposes, we first need to train the model. It can be done by using the following `fit()` function:

```
NB_Classifier = NB_Gaussian.fit(train, train_labels)
```

Step 6 – Predicting test set result

In this step, let us evaluate our trained model on the test dataset and predict the result. We will use the following function called `predict()`:

```
predictions = NB_Classifier.predict(test)
```

For example, printing the predicted values for malignant and benign classes:

```
print(predictions)
```

Output:

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

Step 7 – Evaluating the accuracy

Finally, we can evaluate the accuracy of our trained ML model as follows:

```
from sklearn.metrics import confusion_matrix
cm_NBClassifier= confusion_matrix(test_labels, predictions)
print('Confusion Matrix:',cm_NBClassifier)
```

Output:

```
Confusion Matrix: [[57   6]
        [  7 101]]
```

```
from sklearn.metrics import accuracy_score
print('Accuracy:',accuracy_score(test_labels,predictions))
```

Output:

```
Accuracy: 0.9239766081871345
```

We can see that the accuracy of our ML model is 92.40%.

Lazy earning versus eager learning

In a nutshell, we can divide the previously stated classification process mainly into the following two steps:

1. Building an ML model with a given set of training tuples.
2. Applying that ML model to a given set of testing tuples.

And based on these steps, in the data classification process, there are two types of learners, namely, lazy and eager learners.

Lazy learners simply store training datasets or do only a little processing and wait for the testing tuple to start classifying the data. Such learners spend less time learning and more time on data classification. Case-based reasoning and **K-nearest neighbors** (**k-NN**) are lazy learner algorithms.

On the other hand, Eager learners, when given a training dataset, start building a classification model without waiting to get the testing tuple. Such learners spend more time learning and less time on data classification. **Artificial Neural Networks** (**ANN**), Naïve Bayes, and Decision Trees are some eager learner algorithms.

Performance metrics for classification

What do you think, our job is finished once we finish the implementation of our ML model? No, certainly not because we should find out how effective this model is. There are different evaluation metrics to check the performance of classification algorithms. We should choose the metrics for our ML model very sensibly as it influences how the performance of the ML model is measured. The following are some of the important performance metrics to evaluate predictions for classification problems:

Confusion matrix

The confusion matrix, mainly used for classification problems having the output of two or more types of classes, is one of the most intuitive metrics to find the accuracy of your ML model. The following figure depicts the structure of a confusion matrix:

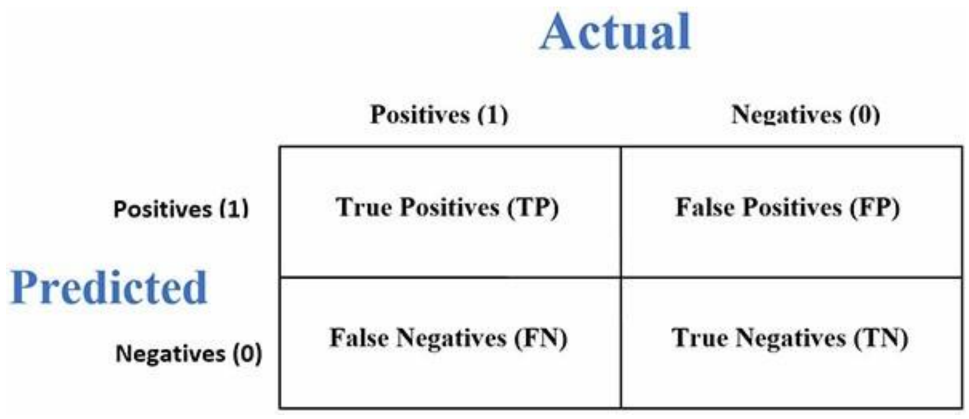


Figure 3.3: *Structure of the confusion matrix*

We can see that the confusion matrix is a table having two dimensions, namely, Actual and

Predicted. These dimensions have the values, namely, **True Positives** (**TPs**), **True Negatives** (**TNs**), **False Positives** (**FPs**), and **False Negatives** (**FNs**). Let's understand the terms associated with the confusion matrix:

- **True Positives (TPs)**: Both Actual class and Predicted class = 1. For example, the case will come under TPs if someone is Diabetic (1), and the ML model also classifies his/her case as Diabetic (1).
- **True Negatives (TNs)**: Both Actual class and Predicted class = 0. For example, the case will come under TNs if someone is Non-Diabetic (0), and the ML model also classifieshis/her case as NON-Diabetic (0).
- **False Positives (FPs)**: Actual class = 0 and Predicted class = 1. For example, the case will come under FPs if someone is Non-Diabetic (0), but the ML model classifies his/her case as Diabetic (1).
- **False Negatives (FNs)**: Actual class = 1 and Predicted class = 0. For example, the case will come under FNs if someone is Diabetic (1), and the ML model classifies his/her case as Non-Diabetic (0).

Scikit-learn ML library provides the **`confusion_matrix`** function with the help of which we can compute the confusion matrix for the created classification model.

Accuracy

Accuracy, one of the most common performance metrics for ML classification models, is the total correct predictions made divided by all predictions made. Following is the formula to calculate the accuracy of our classification model:

$$Accuracy = \frac{TPs + TNs}{TPs + FPs + FNs + TNs}$$

Scikit-learn ML library provides the accuracy_score function with the help of which we can compute the accuracy of the created classification model.

When to use?

Accuracy can be used as an evaluation measure for balanced classification problems, that is, where the target classes are not skewed.

For example, the fruits image dataset is having 60% images of apples and the remaining 40% images of oranges, which means the target classes are balanced. In such cases, accuracy as a metric for our classification model will be a good choice.

When NOT to use Accuracy:

On the other hand, accuracy as an evaluation measure is NOT useful where the target class is very sparse.

For example, what will be the result of prediction if we will try to predict whether an asteroid will hit the earth or not? The model will predict NO every time and it will be 99% accurate. In this way, the classification ML model will be accurate and not valuable at all.

Precision

Precision is a measure that answers the following question:

How much predicted proportions are truly positive?

By using a confusion matrix, we can calculate the precision by using the following formula:

$$Precision = \frac{TPs}{TPs + FPs} = \frac{True\ Potivies}{Total\ Predicted\ Positives}$$

We can see that precision tells us how precise our ML model is, that is, how many actual positives are there out of those predicted positives.

Precision can be used as an evaluation measure for systems where the cost of FPs is very high. For example, in email spam detection, FPs means that a non-spam email has been identified as spam. So, for models like spam detection, the precision should be high otherwise the user might lose important emails.

Recall

Recall is another useful measure that answers the following question:

How many actual positives are correctly classified?

By using a confusion matrix, we can calculate recall by using the following formula:

$$Recall = \frac{TPs}{TPs + FNs}$$

For example, Recall will be 0 in an asteroid prediction problem – if an asteroid will hit the earth or not because we never predicted a true positive. On the other hand, Recall will be 1 if we predict 1 for every example.

Recall can be used as an evaluation measure for systems in which our motive is to capture as many positives as possible. For instance, the ML system predicts whether a person has cancer or not. In such systems, we would like to detect the disease although we are not very sure.

Specificity

Specificity, the exact opposite to recall, is a measure that answers the following question:

How many actual negatives are correctly classified?

By using a confusion matrix, we can calculate recall by using the following formula:

$$Specificity = \frac{TNs}{TNs + FPs}$$

F1 score

F1 score, utilizing tradeoff of precision versus recall, is the harmonic mean of precision and recall. The score is a number between 0 and 1(the best is 1 and the worst is 0) and can be calculated with the help of the following formula:

*F1 Score= 2 * (precision * recall) / (precision + recall)*

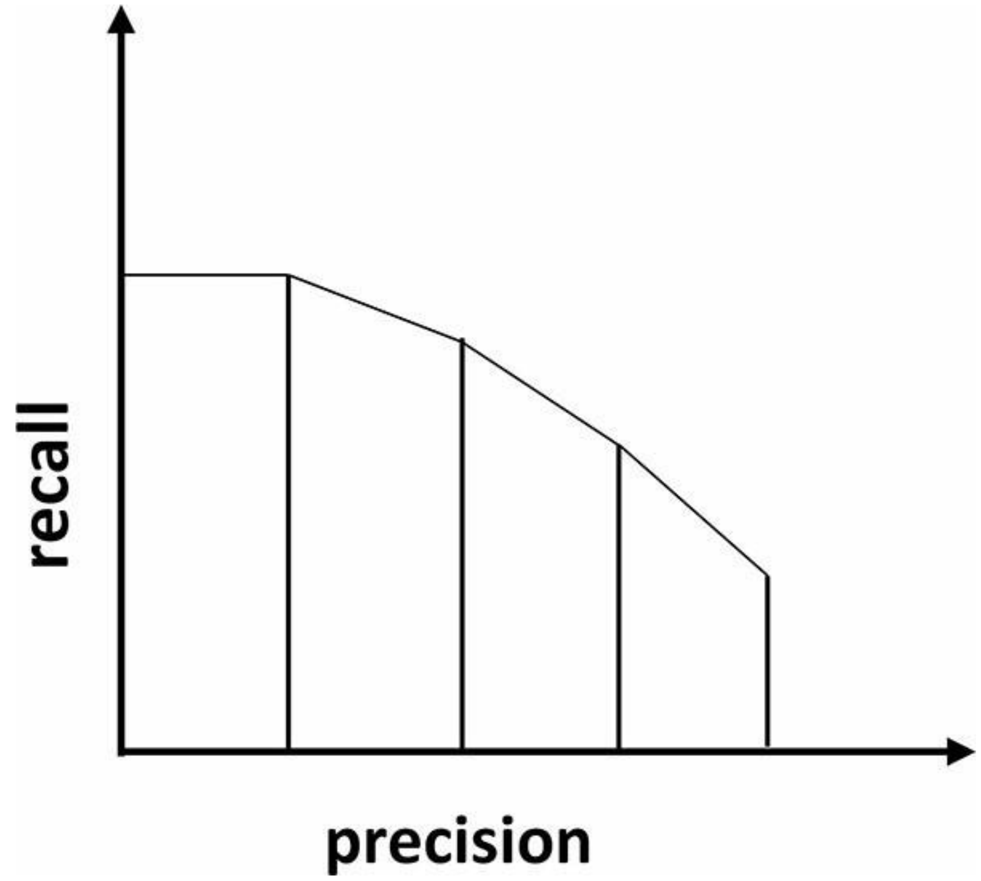


Figure 3.4: *Precision–Recall tradeoff*

Our ML classifier should have both good precision and a good recall. The F1 score is having an equal relative contribution of precision and recall hence maintains a balance between the two.

Scikit-learn ML library provides the `f1_score` function with the help of which we can compute the F1 score of our classifier. The function can be imported from `sklearn.metrics`.

Regression

Regression, a sub-field of supervised ML, may be defined as the process of predicting continuous numeric values based on the input data that is being provided. In simple words, the regression task is an ML task having a value estimation as to its main objective. Unlike classification, which uses classes to learn the specific relationship between independent variables, that is, inputs, and corresponding dependent variables, that is, outputs, regression ML models use input data features and their corresponding continuous numeric output variables.

As discussed in the previous chapter, **Simple Linear Regression** (**SLR**) and **Multiple Linear Regression** (**MLR**) are the two most commonly used regression models. SLR predicts a response using a single feature and assumes that the two variables are linearly related, whereas MLR predicts a response using two or more than two features or independent variables. The following figure shows SLR and MLR models to predict house prices based on their plot area:

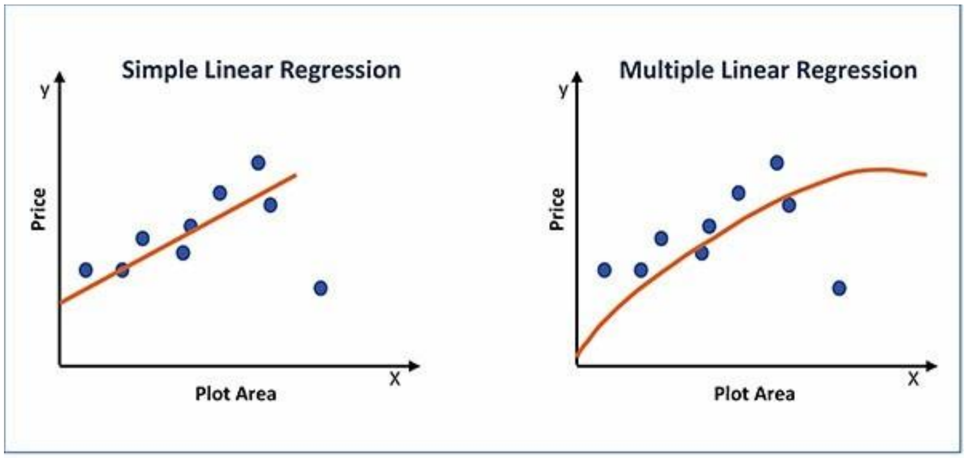


Figure 3.5: *Simple and Multiple Linear Regression models*

Various steps to build a regressor using Python

In this section, we are going to learn various steps to build a regressor using the Python programming language. For this purpose, we will use Scikit-learn – an open-source machine learning library.

Step 1 – Import ML library

The process of building a regressor using Python starts by importing the ML library. As discussed, we will use the Scikit-learn ML library, which can be imported with the help of the following Python command:

```
import sklearn
```

Step 2 – Import dataset

Once we import the Scikit-learn ML library, we need a dataset to train our machine learning model. To give you an example, we will create a linear regression model by using sklearn's *Boston house-price dataset*. We can use the following command to import this dataset:

```
from sklearn.datasets import load_boston
```

After the dataset is imported, we need to load the dataset. With the help of the following command, we will load this dataset in a variable named `boston_data`:

```
boston_data = datasets.load_boston(return_X_y=False)
```

Also after the dataset is imported, we need to define the feature matrix `X` and the response vector `Y` as follows:

```
X  =  boston_data.data
y = boston_data.target
```

Step 3 – Organizing data into training and testing set

To test the ML model on new unseen data, we need to split the dataset into two parts, training set and test set. For this purpose, Scikit-learn provides us a function named `train_test_split()`. Let's see how we can divide the dataset into 70% training set and the remaining 30% test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)
```

Step 4 – Creating ML model

Once we divide the dataset into the training and test set, our ML model has to be created. Here, for creating the ML model for the cancer dataset, we will use `LinearRegression()` from `sklearn.linear_model`. The command is as follows:

```
from sklearn import linear_model
MLR_reg = linear_model.LinearRegression() #creating regression object
```

Step 5 – Train the model

To use this ML model for testing, we first need to train this model. It can be done by using `fit()` function as follows:

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

Step 6 – Plotting the regression line

We can now plot the regression line as follows:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import metrics
%matplotlib inline

plt.style.use('bmh')
  plt.scatter(MLR_reg.predict(X_train), MLR_reg.predict(X_train) - y_train, color
  = "green", s = 20, label = 'Train_data')
plt.scatter(MLR_reg.predict(X_test), MLR_reg.predict(X_test) - y_test, color =
"blue", s = 10, label = 'Test_data')

plt.hlines(y = 0, xmin = 0, xmax = 50, color = 'red', linewidth = 1.25)
plt.legend(loc = 'upper right')
plt.title("Residual errors(eo)")
plt.show()
```

Output:

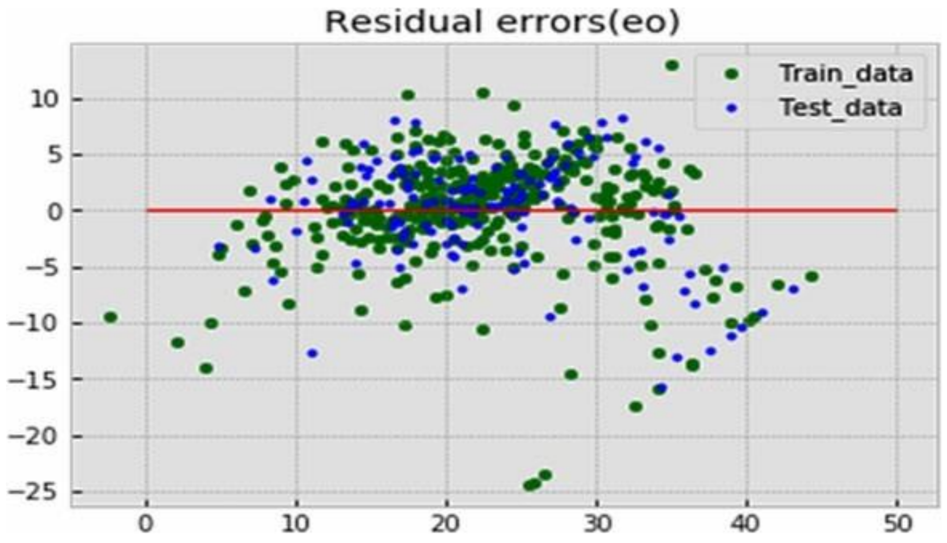


Figure 3.6: *Multiple Linear Regression model for Boston House Price prediction*

Step 7 – Calculating the variance

Last but not the least, we can calculate the variance, that is, how far observed values differ from the average of predicted values, for our linear regression model with the help of the following command:

```
print('Variance score: {}'.format(MLR_reg.score(X_test, y_test)))
```

Output:

```
Variance score: 0.7836295385076281
```

Performance metrics for regression

The following are some of the important performance metrics to evaluate predictions for regression problems:

Mean Absolute Error (MAE)

As the name implies, it represents the absolute difference between the target value and the predicted value. Rather than indicating underperformance or overperformance of our regression model, MAE gives us an idea of how wrong the model predictions were. The following are some of the characteristics of MAE:

- **Robust to outliers**: MAE does not penalize the errors as extremely as **Mean Squared Error** (**MSE**). That’s the reason it is not suitable for the applications that pay more attention to the outliers.
- **Linear score**: Another important characteristic of the MAE performance metric is that it always results in a linear score. By linear score, we mean that all the individual differences are weighted equally.

Let’s have a look at the formula to calculate MAE:

$$MAE = \frac{1}{n}\sum|Y - \hat{Y}|$$

Here, *y* is the Actual Output Value and Ŷ is the Predicted Output Value.

Scikit-learn ML library provides us the mean_absolute_error function to compute MAE.

Mean Squared Error (MSE)

MSE is the average of the squared difference between the target value and the predicted value. The only difference between MAE and MSE is that rather than taking the absolute difference between the target value and the predicted value in MAE, MSE squares that difference. The advantage of MSE over MAE is that MSE penalizes even a very small error, which leads to over-estimation of how bad our regression model is.

Let’s have a look at the formula to calculate MSE:

$$MSE = \frac{1}{n}\sum(Y - \hat{Y})^2$$

Here also, *Y* is the Actual Output Value and Ŷ is the Predicted Output Value.

Scikit-learn ML library provides us the `mean_squared_error` function to compute MAE.

R-Squared (R2)

What if the MSE for your regression model is 29? Is your model good enough or it needs improvement? On the other hand, what if the MSE for your regression model is 0.2?

By looking at MSE, it is very hard to predict if the regression model is good. Hence, we have to measure how good a regression model is than the constant baseline.

For such measurements, we have a performance metric called R2 or the coefficient of determination. Although R2 is like MSE but is scale free, the advantage of being scale free means, despite the output value (very small or very large), the value of R2 will always be between -∞ and 1.

Let’s have a look at the formula to calculate R2:

$$R^2 = 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y}_i)^2}$$

In the preceding equation, the numerator is MSE (Model) and the denominator is MSE (Baseline), that is, the variance in values.

In a nutshell, R2 represents the ratio between how good a regression model is and how good the naïve mean model is.

Scikit-learn ML library provides us the r2_score function to compute R2.

Adjusted R-squared (R2)

The major flow with R-squared is that its value never decreases no matter how many number of variables we add to the regression model. In fact, the value of R-squared remains the same or increases with the addition of new independent variables to the data. Clearly, this doesn't make sense because every independent variable might not be useful in determining the output.

Adjusted R2 deals with this issue because it considers the number of independent variables used for determining the target variable.

Let's have a look at the formula to calculate R2:

$$Adjusted\ R^2 = \{1 - \left[\frac{(1-R^2)(n-1)}{(n-m-1)}\right]\}$$

In the preceding equation:

- *n* is the number of data points in the dataset.
- *m* is the number of independent variables.
- *R* is the R-squared value determined by the model.

Example:

The following is a simple Python recipe that gives us an insight about how to use these performance metrics on our regression model:

```
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
Xactual = [2, -1, 8, 9]
Ypredict = [1.5, -0.7, 6, 7.9]
print ('Mean Absolute Error(MAE) =',mean_absolute_error(Xactual, Ypredict))
print ('Mean Squared Error(MSE) =',mean_squared_error(Xactual, Ypredict))
print ('Coefficient of Determination (R Squared) =',r2_score(Xactual, Ypredict))
```

Output:

```
Mean Absolute Error(MAE) = 0.9749999999999999
Mean Squared Error(MSE) = 1.3874999999999997
Coefficient of Determination (R Squared) = 0.9195652173913044
```

Conclusion

In this chapter, we learned about two main tasks, Classification and Regression, of Supervised Machine Learning. We learned that the classification model allows us to predict categorical output responses for the given input data, whereas the regression model allows us to predict continuous numerical output responses for the given input data.

Starting with importing the dataset to evaluating the performance, we also explored various steps for building a classifier and regressor in the Python programming language along with an example. In a nutshell, the basic pipeline of classification or regression tasks works as follows:

- The task, whether classification or regression, starts with some initial configuration of the model.
- Once configured, based on the input, the model predicts the output.
- The predicted output is now compared with the target value.

- Based on the comparison, the measure of model performance is taken.
- Finally, to achieve the optimal value of the performance metric we choose, we need to iteratively adjust the various parameters of the model.

We have also covered the various performance matrices for both classification and regression.

However, the constant standard for evaluating the performance of a classifier is different for different tasks, accuracy, the total correct predictions made divided by all predictions made, is a common standard in the case of every classification task. Other important matrices for evaluating the performance of a classification algorithm that we covered in this chapter include recall, precision, Specificity, F1 Score, and so on. On the other hand, this is quite true for regression as well. Some of the important metrics for evaluating the performance of a regression algorithm that we covered in this chapter are **Mean Absolute Error** (**MAE**), **Mean Squared Error** (**MSE**), and R-squared (R^2).

In the next chapter, you will learn about clustering, which is one of the most useful areas of unsupervised learning.

Questions

1. Explain classification with an example. State the differences between binary and multi-class classification.
2. What are the various steps to build a classifier using the Python programming language? Explain with an example.
3. What is a Confusion matrix? Explain various terms associated with it.
4. What is the significance of the F1 Score as an evaluation metric for classification algorithms?
5. What is Regression? State the differences between simple linear regression and multiple linear regression?
6. What are the various steps to build a regressor using the Python programming language? Explain with an example.
7. Explain Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R2) performance evaluation metrics for regression algorithms.

1 https://scikit-learn.org/stable

2 https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

CHAPTER 4

Clustering Using Unsupervised Learning

Introduction

Machine learning works well if we have a labeled data. But what about the cases when we do not have the labeled data? One option is to get some labeled data and the other is to use unsupervised machine learning. Unsupervised ML algorithms discover the interesting patterns in data for learning without any label data and teacher for guidance. That's the reason they are closely aligned with what we call true **Artificial Intelligence** (**AI**).

In the previous chapter, you got a brief overview of supervised ML tasks: Classification and Regression. This chapter provides details about Clustering, an unsupervised ML task. It addresses various methods to form clusters. You will get to know about the most commonly used Clustering algorithms and their implementation in the Python programming language. Also, you will get a glimpse of the best evaluation metrics for clustering.

Structure

In this chapter, we will discuss the following topics:

- Clustering
- Various methods to form clusters
- Important ML clustering algorithms
    - K-means clustering Algorithm
    - Mean-shift clustering Algorithm
    - Hierarchical clustering algorithm
- Performance metrics for clustering
    - Silhouette analysis
    - Davies–Bouldin index (DB index)
    - Dunn index

Objectives

After studying this unit, you should be able to implement various machine learning clustering algorithms, such as k-means, mean-shift, and hierarchical clustering, in the Python programming language. You will also learn about the performance metrics for clustering algorithms.

Clustering

What if you need to find the similarity and relationship patterns among your data samples using machine learning? To do so, ML provides us an unsupervised learning method called Clustering. As the name implies, this method, after finding the relationship patterns, clusters the data samples that have similar features into groups. We can depict the working of clustering methods in the following figure:

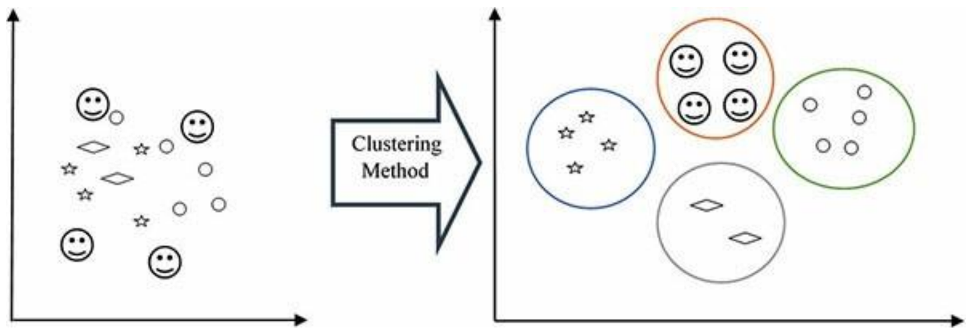


Figure 4.1: *Clustering*

It is quite clear from *Figure 4.1* that the system has four distinct clusters of images. The first cluster depicts the star images, the second cluster depicts smiley images, the third cluster depicts diamond images, and the fourth cluster depicts circle images. The main objective of the clustering method is always to create different clusters in such a way that elements that belong to the same clusters are near each other and the elements of other clusters are far apart.

Various methods to form clusters

The clusters, as you can see in the preceding figure, are formed in a spherical form. But do you think it is the only way/method to form clusters? No, there are various other methods also with the help of which we can form clusters. Some of the important methods are discussed as follows:

- **Density-based method**: As the name implies, density-based methods use dense region to form clusters of data samples. The advantages of using this method are that it is having a good accuracy rate and the ability to merge two clusters together. The following are the two popular examples of density-based methods:
    - **Density-Based Spatial Clustering of Applications with Noise** (**DBSCAN**): It separates core samples of high-density from low-density samples and expands clusters from them. This method is most suitable for the data having clusters of similar density.
    - **Ordering Points to Identify Clustering Structure** (**OPTICS**): It is closely related to DBSCAN, separates core samples of high-density from low-density samples, and expands clusters from them. But unlike DBSCAN, OPTICS keeps cluster hierarchy for a variable neighborhood radius. This method is most suitable for large datasets.
- **Hierarchical-based method**: As the name implies, hierarchical-based methods use tree-type structures to form clusters of data samples. In other words, the clusters are formed based on the hierarchy. The two categories of this method are bottom–up (agglomerative)

and top–down (divisive). The following are the two popular examples of hierarchical-based methods:

  - **Clustering using Representatives** (**CURE**): CURE, for efficiently handling the clusters and eliminating outliers, adopts a middle ground in between the centroid based and the all-point extremes. It is an efficient data clustering algorithm for large datasets and also identifies clusters having non-spherical shapes.
  - **Balanced Iterative Reducing Clustering using Hierarchies** (**BIRCH**): BIRCH first generates a compact summary of large datasets retaining as much information as possible and then clusters this smaller summary rather than clustering the large dataset itself. Its major drawback is that it can only process metric attributes that is, attributes in which no categorical attributes are present.

- **Partitioning**: As the name implies, partitioning methods use partitioning of an object into n number of clusters where the number of clusters formed will be equal to the number of partitions. The following are the two popular examples of hierarchical-based methods:

  - **Clustering Large Applications based upon Randomized Search** (**CLARANS**): CLARANS[1], introduced by Raymond T.Ng and Jiawei Han of IEEE computer society, is a partitioning method of clustering particularly designed to cluster spatial data.
  - **k-means clustering**: K-means clustering, one of the simplest unsupervised machine learning algorithms, groups similar data points together and discovers underlying patterns. 'k' in K-means refers to the fixed number of clusters in a dataset. While computing the centroids, the k-means algorithm assigns data points in such a manner that the sum of the squared distance between the data points and centroid would be minimum.

- **Grid**: As the name implies, grid methods use grid-like structures to form clusters. Once formed, various clustering operations performed on these grids will be independent of the number of data objects. The following are the two popular examples of hierarchical-based methods:

  - **Statistical Information Grid** (**STING**): STING[2], proposed by Wang, Yang, and Muntz at VLDB 97, is a clustering method in which the spatial area is divided into rectangular cells. For each rectangular cell, the higher-level cell is divided into various smaller cells in the next lower level. The statistical information of every cell is calculated and stored beforehand to calculate its parameters. To answer spatial data queries, it uses a top–down approach. This process will be repeated until the bottom layer is reached.
  - **Clustering in Quest** (**CLIQUE**): CLIQUE[3], proposed by Agrawal, Gehrke, Gunopulos, Raghavan, is based on automatically identifying the subspace of high-dimensional data space that allows better clustering when compared with the original space.

Important ML clustering algorithms

In this section, we will be discussing some important ML Clustering algorithms and their implementation in Python.

K-means clustering algorithm

K-means clustering, one of the simplest unsupervised machine learning algorithms, groups similar data points together and discovers underlying patterns. ‘k’ in K-means refers to the fixed number of clusters in a dataset. While computing the centroids, the k-means algorithm assigns data points in such a manner that the sum of the squared distance between the data points and centroid would be minimum.

The way the k-means clustering algorithm works is explained in the following steps:

1. Specify the number of clusters, that is, ‘k’.
2. It shuffles the dataset to initialize centroids and then randomly selects ‘k’ data points for the centroids.
3. Next, it keeps iterating the following until it finds the optimal centroid that is, there is no change to the centroids:
    1. First, it calculates the sum of squared distance between data points and centroids.
    2. Second, it assigns each data point to the nearest cluster.
    3. Finally, it takes the average of all data points of clusters to compute the centroids for those clusters.

Implementing k-means clustering in Python

Example1:

In this example, we will implement k-means clustering on the 2-dimensional dataset sample, which we generated on 200 sample values, having 3 clusters.

```
#Importing necessary packages
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
from sklearn.cluster import KMeans

#Generating 2-dimensional dataset having 3 blobs
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=200, centers=3, cluster_std=0.8)

#Let’s visualize our dataset
plt.scatter(X[:, 0], X[:, 1], s=10);
plt.show()
```

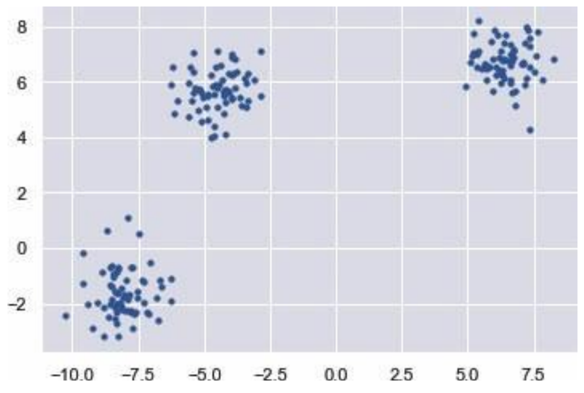


Figure 4.2: *2-D data having 3-blobs*

```
#Creating an object of k-means and providing number of clusters
kmeans = KMeans(n_clusters=3)

#Train the model by using fit method of k-means() class
kmeans.fit(X)

# do the predictions
y_kmeans = kmeans.predict(X)

# Plotting and visualizing cluster's centers picked by k-means estimators
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=10, cmap='rainbow')
centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='black');
plt.show()
```

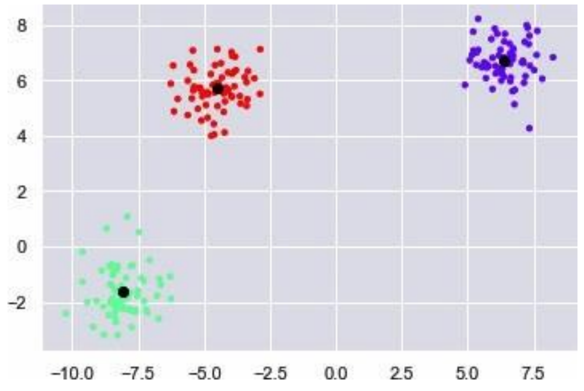


Figure 4.3: *Cluster centers picked by k-means estimators*

Example 2:

In this example, we will apply k-means clustering on the scikit-learn's digit dataset. Let's see how k-means will identify the digits without the original label information.

```
#Importing necessary packages
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
```

```
#Loading sklearn digit dataset. We will also make its object.
digits = load_digits()
digits.data.shape
```

Output:

```
(1797, 64)
```

We can see from the preceding output that digit dataset is having 1797 samples and 64 features.

```
#Creating an object of k-means.
kmeans = KMeans(n_clusters=10, random_state=0)

#Train the model by using fit method of k-means() class
clusters = kmeans.fit_predict(digits.data)
kmeans.cluster_centers_.shape
```

Output:

```
(10, 64)
```

We can see from the preceding output that 10 clusters with 64 features are created by k-means.

```
#Picking up cluster centers learned by K-means clustering algorithm.
fig, ax = plt.subplots(2, 5, figsize=(8, 3))
centers = kmeans.cluster_centers_.reshape(10, 8, 8)
for axi, center in zip(ax.flat, centers):
  axi.set(xticks=[], yticks=[])
  axi.imshow(center, interpolation='nearest', cmap=plt.cm.binary)
```

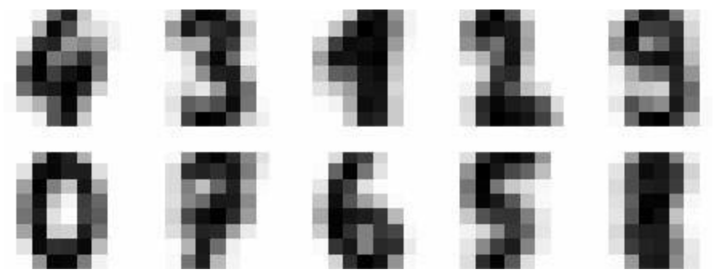


Figure 4.4: *Cluster centers picked by k-means estimators*

```
#Matching the learned cluster lables with true lables found in them.
from scipy.stats import mode
labels = np.zeros_like(clusters)
for i in range(10):
  mask = (clusters == i)
  labels[mask] = mode(digits.target[mask])[0]

#Checking the accuracy.
from sklearn.metrics import accuracy_score
accuracy_score(digits.target, labels)
```

Output:

```
0.7952142459654981
```

`

Elbow method for k evaluation in K-means Clustering algorithm

What is the fundamental step for any unsupervised algorithm? It is to evaluate the optimal number of clusters into which the data points may be clustered. One of the most popular methods to evaluate the optimal value of k is the elbow method. It first runs the k-means clustering on the dataset for a range of values for k. Then it computes an average score for each value of k in that range. By default, it computes:

- **Distortion**: Distortion score is the average of the squared distances from the cluster centers of the respective clusters.
- **Inertia**: It may be defined as the sum of squared distances from each point to its assigned center.

While implementing the preceding k-means clustering, we choose **`k = 3`**. It is because from the visualization of data, we can see that the optimal number of clusters should be around 3. But we may be wrong because visualization of data alone cannot provide the right answer always. So, let's evaluate the optimal value of k for our dataset by using the elbow method:

```
#Importing necessary packages
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn import metrics
from scipy.spatial.distance import cdist
import numpy as np
from sklearn.cluster import KMeans

#Generating 2-dimensional dataset having 3 blobs
from sklearn.datasets.samples_generator import make_blobs
X, y_true = make_blobs(n_samples=200, centers=3, cluster_std=0.8)

#Let's visualize our dataset
plt.scatter(X[:, 0], X[:, 1], s=10);
plt.show()
```

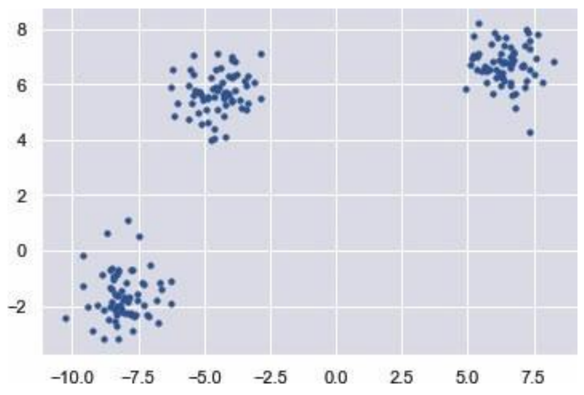


Figure 4.5: *Data having 3-blobs*

```
distortions = []
inertias = []
mapping_1 = {}
mapping_2  = {}
K = range(1, 9)
for k in K:
  # Building and fitting the model
  kmeans_model = KMeans(n_clusters=k).fit(X)
  kmeans_model.fit(X)

  distortions.append(sum(np.min(cdist(X,
  kmeans_model.cluster_centers_,'euclidean'), axis=1)) / X.shape[0])
  inertias.append(kmeans_model.inertia_)

  mapping_1[k] = sum(np.min(cdist(X, kmeans_model.cluster_centers_,'euclidean'),
  axis=1)) / X.shape[0]
  mapping_2[k] = kmeans_model.inertia_

# Visualizing the result using different values of distortion
plt.plot(K, distortions, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Distortion')
plt.title('The Elbow Method using Distortion')
plt.show()
```

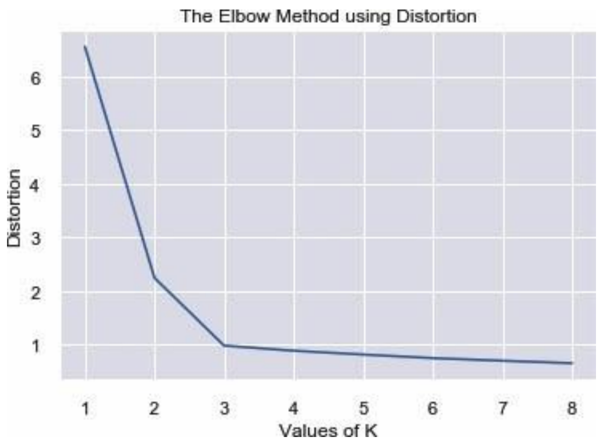


Figure 4.6: *The Elbow method using different values of distortion*

```
# Visualizing the result using different values of inertia
plt.plot(K, inertias, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Inertia')
plt.title('The Elbow Method using Inertia')
plt.show()
```

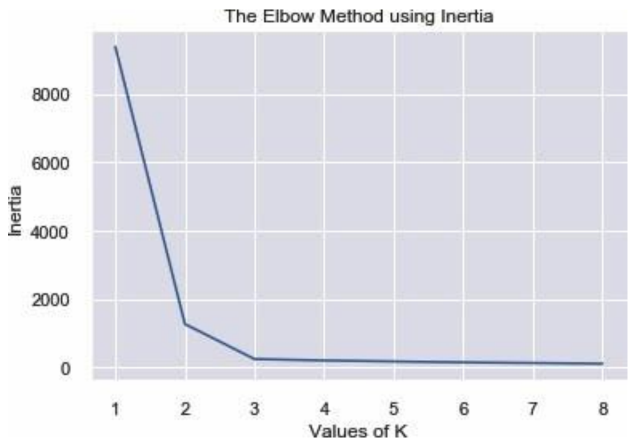


Figure 4.7: *The Elbow method using different values of inertia*

We can find the optimal value of k at the elbow, that is, the point after which the distortion and inertia start decreasing linearly (check the preceding outputs). That's the reason, for our dataset, we can conclude that the optimal number of clusters is 3.

Advantages of K-means Clustering algorithm

Some of the advantages of K-means clustering are given as follows:

- Easy to understand and implement.
- In the case of having a large number of variables, the K-means clustering algorithm is much faster than the Hierarchical clustering algorithm.
- In comparison with the Hierarchical clustering algorithm, we get tighter clusters with the K-means clustering algorithm.

Disadvantages of K-means Clustering algorithm

Some of the disadvantages of K-means clustering are given as follows:

- Difficult to predict the value of k.
- Initial input (like the number of clusters) and order of data can impact the final output.
- Sensitive to rescaling, that is, the output will be impacted strongly if we rescale our data by any means like normalization or standardization.

Mean-shift clustering algorithm

The mean-shift algorithm, used in unsupervised learning, is one of the most dominant ML clustering algorithms. It is also called a non-parametric algorithm because it does not make any kind of assumptions. As the name implies, this algorithm iteratively assigns the data points to the clusters by shifting points toward the cluster centroid (contains the highest density of data points).

One of the advantages of the mean-shift algorithm over other ML clustering algorithms, like k-means, is that it does not require the user to specify the number of clusters in advance.

Let us understand the working of the mean-shift clustering algorithm with the help of the following steps:

1. The first step is the centroid initialization in which all the data points are initialized to cluster centroids. In this way, we can start with as many clusters as data points. The aim of the mean-shift algorithm is to achieve the optimal number of clusters.
2. In this step, the algorithm will update the location of the new centroid.
3. Once updated, the algorithm will now iteratively repeat this process. It will then move to the highest density of data points, that is, cluster centroid.
4. Finally, it will stop once the centroids reach a position from where they cannot move further.

Implementing mean-shift clustering algorithm in Python:

Let’s see how we can implement mean-shift clustering in the Python programming language. For this, we are going to generate a sample dataset from **`sklearn.dataset.sample_generator`**.

```
#Importing necessary packages
import numpy as np
import pandas as pd
from sklearn.cluster import MeanShift
from sklearn.datasets.samples_generator import make_blobs
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#Generating sample dataset
from sklearn.datasets.samples_generator import make_blobs
centers = [[2,2,2],[5,5,5],[3,10,10]]
X, _ = make_blobs(n_samples = 300, centers = centers, cluster_std = 0.5)
plt.scatter(X[:,0],X[:,1])
plt.show()
```

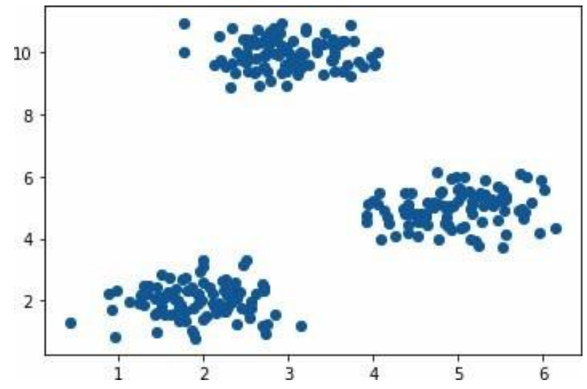


Figure 4.8: *Sample dataset*

The preceding output is having sample dataset with 300 samples and 3 clusters.

```
# Train the model using meanshift algorithm
ms = MeanShift(bandwidth = 2)
ms.fit(X)

# Storing the cordinates for the cluster centers
cluster_centers = ms.cluster_centers_

# Plotting the data points and centroids. We will be using Axes3D to plot 3D
graph.
fig = plt.figure()
ms_ax = fig.add_subplot(111, projection='3d')
ms_ax.scatter(X[:,0], X[:,1], X[:,2], marker='x')
ms_ax.scatter(cluster_centers[:,0], cluster_centers[:,1], cluster_centers[:,2],
marker='x', color='red', s=500, linewidth=10, zorder=10)
plt.show()
```

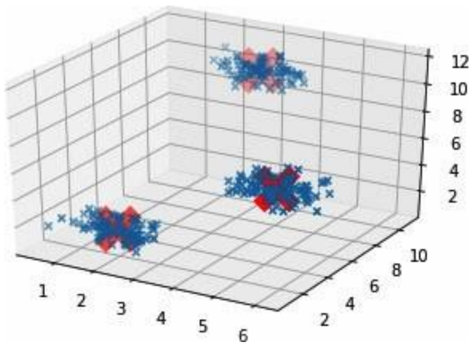


Figure 4.9: *3D graph of centroids*

Advantages of mean-shift clustering algorithm

Some of the advantages of mean-shift clustering are given as follows:

- We do not make any model assumptions as we do while implementing K-means or Gaussian mixture.
- Mean-shift clustering can model the complex clusters having a non-convex shape.
- It can automatically determine the number of clusters with the help of only one parameter named bandwidth.
- Mean-shift clustering does not have the problem of local minima.

Disadvantages of mean-shift clustering algorithm

Some of the disadvantages of mean-shift clustering are given as follows:

- It is not useful in the case of high dimensions, that is, when the number of clusters changes abruptly.
- In some specific applications, we need a specific number of clusters.
- The mean-shift algorithm does not differentiate between meaningful and meaningless modes.

Hierarchical clustering algorithm

Hierarchical clustering algorithm amalgamates the unlabeled data points having the same characteristics. The two categories of this clustering algorithm are given as follows:

- **Agglomerative hierarchical algorithms**: As the name implies, this category of hierarchical clustering algorithm first treats the data points as a single cluster and then agglomerate, that is, merge the pairs of clusters by using the bottom–up approach.
- **Divisive hierarchical algorithms**: As the name implies, this category of hierarchical clustering algorithm divides one big cluster into small clusters by using the top–down approach.

Agglomerative hierarchical clustering is one of the most commonly used clustering algorithms.

Let us understand its working with the help of the following steps:

1. Agglomerative hierarchical clustering always starts with having some number of clusters because it treats every data point as a single cluster. For this purpose, let us say we have an M number of clusters.
2. The nature of agglomerative clustering is to join two closet data points to form a big cluster. Hence, it will first give us M-1 clusters and similarly, to form bigger clusters, it will join two closest clusters resulting in a total of M-2 clusters.
3. Repeat the preceding two steps to form a big cluster. The step should be repeated until there remain no more data points left to combine. In other words, until M becomes 0.
4. At last use dendrograms to divide that one big cluster into multiple clusters.

Understanding the role of dendrograms in agglomerative hierarchical clustering

As discussed, the role of the dendrogram is to split one big cluster into multiple clusters of correlated data points. In this way, the role of the dendrogram started once we created the big cluster. Let's understand the role of dendrogram with the help of the following Python program:

```
#Importing necessary packages
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

#Plotting the data points for our example
X = np.array([[10,11],[13,21],[18,20],[24,18],[33,38],[86,76],[74,86], [63,85],
[78,65],[88,97],])
labels = range(1, 11)
plt.figure(figsize=(10, 7))
plt.subplots_adjust(bottom=0.1)
plt.scatter(X[:,0],X[:,1], label='True Position')

for label, x, y in zip(labels, X[:, 0], X[:, 1]):
  plt.annotate(label,xy=(x, y), xytext=(-3, 3),textcoords='offset points',
  ha='right', va='bottom')
plt.show()
```

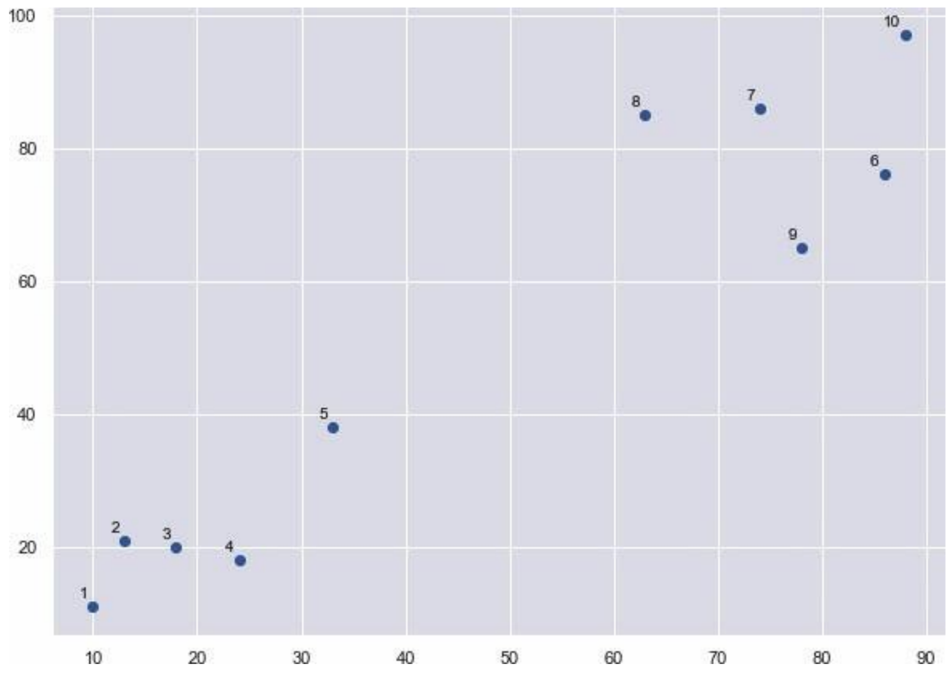


Figure 4.10: *Data points*

From the preceding figure, it is clear that we have only two clusters for our example but as we know in real life there can be thousands of clusters.

```
# Plotting the dendograms of our datapoints
from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt
linked = linkage(X, 'single')
labelList = range(1, 11)
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top',labels=labelList,
distance_sort='descending',show_leaf_counts=True)
plt.show()
```

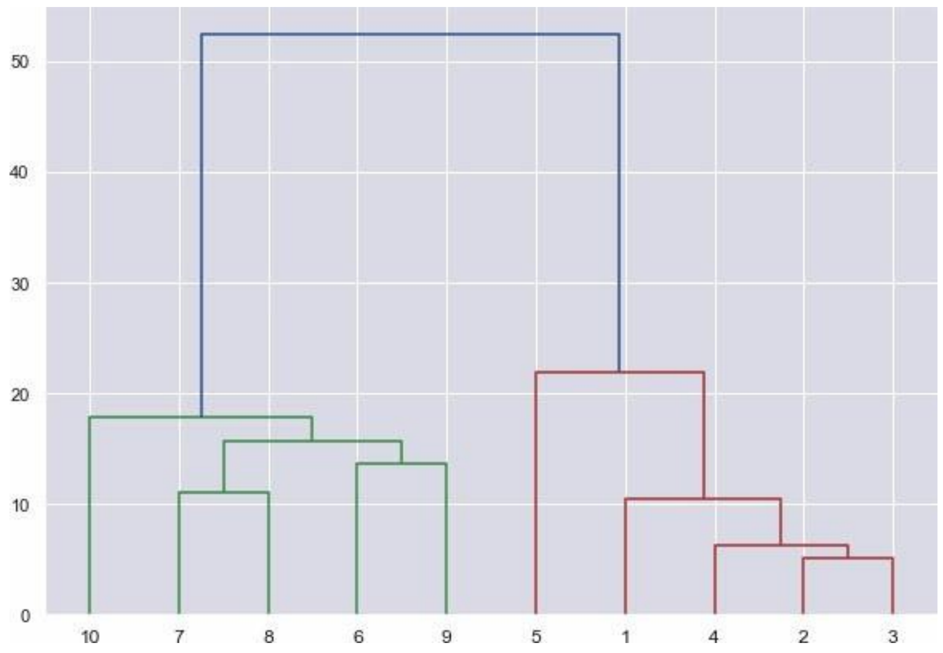


Figure 4.11: *Dendrograms created from our data points*

```
# Predicting the clusters by using AgglomerativeClustering from sklearn.cluster
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean',
linkage='ward')
cluster.fit_predict(X)

# Plotting the clusters
plt.scatter(X[:,0],X[:,1], c=cluster.labels_, cmap='rainbow')
```

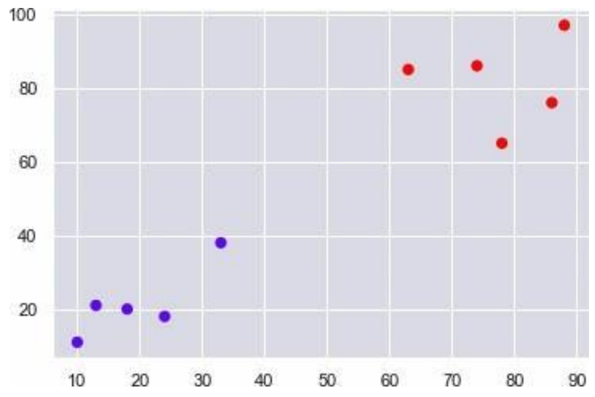


Figure 4.12: *Two clusters plotted from our data points*

Implementing hierarchical clustering algorithm in Python

Let’s see how we can implement hierarchical clustering in the Python programming language. For this, we are going to use Pima-Indians Diabetes dataset..

```
#Importing necessary packages
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
```

```
import numpy as np
from pandas import read_csv
```

```
#Getting the data points from the Pima Indians Diabetes dataset DATASET
path = r"C:\diabetes.csv"
headernames = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age',
'class']
data = read_csv(path, names=headernames)
array = data.values
X = array[:,0:8]
Y = array[:,8]
data.shape
```

Output:

```
(768, 9)
```

```
# Plotting the dendograms of our datapoints
diabetes_patient_data = data.iloc[:, 3:5].values
import scipy.cluster.hierarchy as shc
plt.figure(figsize=(10, 7))
plt.title("Patient Dendogram")
dend_patient = shc.dendrogram(shc.linkage(data, method='ward'))
```

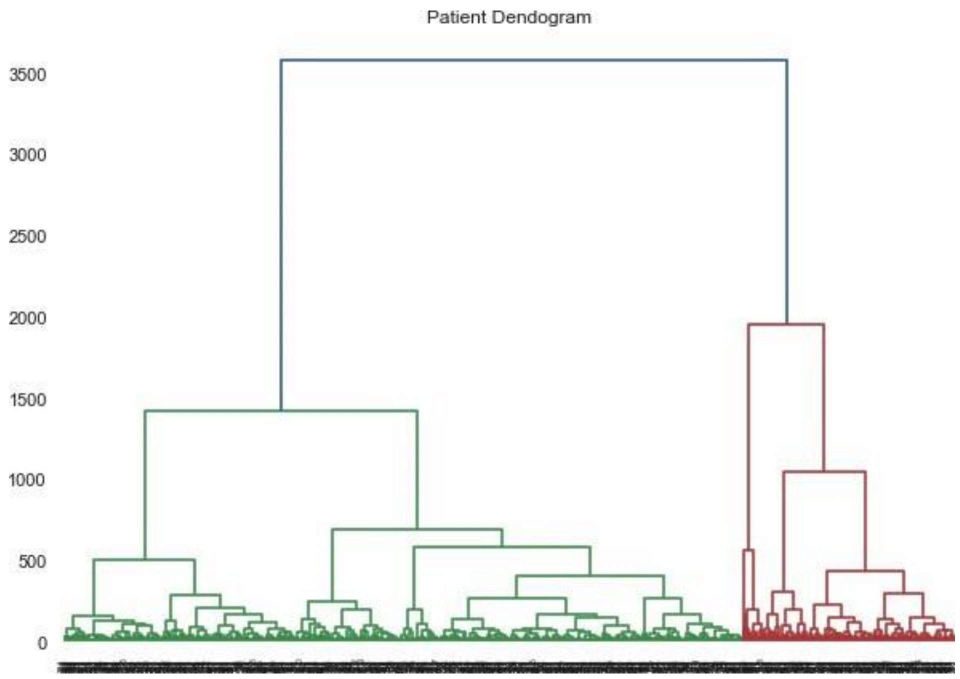


Figure 4.13: *Dendrograms created from Pima-Indian-Database*

```
# Predicting the clusters by using AgglomerativeClustering from sklearn.cluster
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=4, affinity='euclidean',
linkage='ward')
cluster.fit_predict(diabetes_patient_data)
```

```
# Plotting the clusters
plt.figure(figsize=(10, 7))
plt.scatter(diabetes_patient_data [:,0], diabetes_patient_data [:,1],
c=cluster.labels_, cmap='rainbow')
```

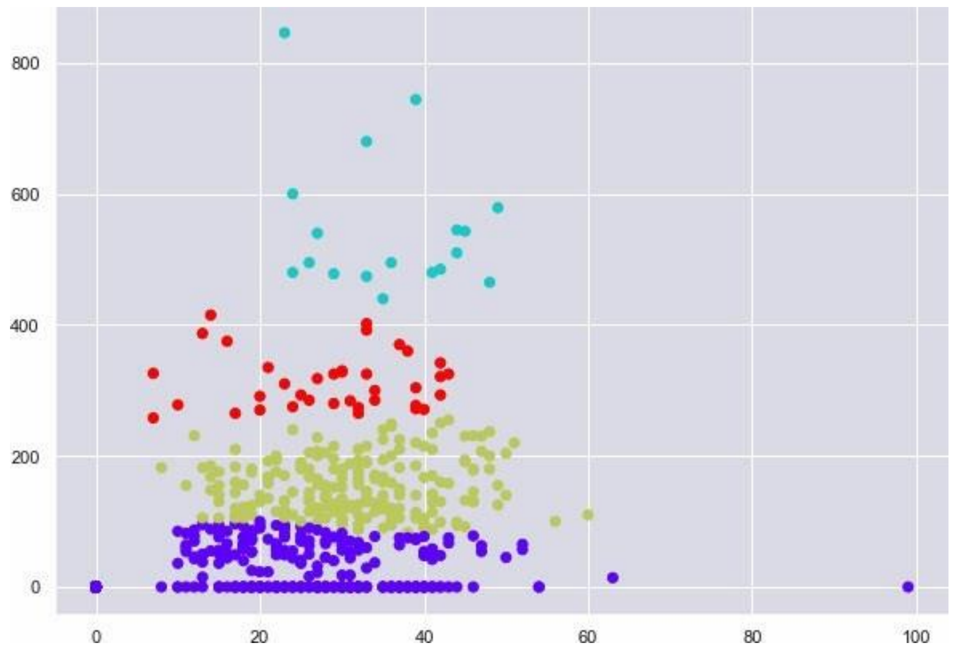


Figure 4.14: *Clusters created from Pima-IndiansDiabetes Database*

Advantages of hierarchical clustering algorithm

Some of the advantages of hierarchical clustering are given as follows:

- It is easy to understand and implement.
- No need to pre-specify the number of clusters. We can easily obtain the number of clusters by cutting the dendrogram at a proper level.

Disadvantages of hierarchical clustering algorithm

Some of the disadvantages of hierarchical clustering are given as follows:

- It does not work well on a large amount of data, missing data, and with mixed data types.
- In comparison with other efficient algorithms, such as k-means, the computation time for clustering is long.

Performance metrics for clustering

Which of these learning algorithms, supervised or unsupervised, do you think is easier for the quality assessment? In the case of supervised learning algorithms, we already have the output labels for each example; hence, it is quite easier to measure the performance of supervised learning algorithms. Conversely, in the case of unsupervised learning, we have the unlabeled data; hence, it is not that easy to measure their performance. However, there are some performance metrics that can give us an insight into the change in clusters. But these performance metrics, instead of measuring the validity of the ML model's prediction, will only evaluate the comparative performance between two ML models.

The following are some of the performance metrics with the help of which we can measure the quality of clustering algorithms:

Silhouette analysis

This metric measures the distance between clusters to check the quality of the clustering model. It uses the Silhouette score to assess the parameters such as the number of clusters. Silhouette score, ranges between [-1, 1], measures the closeness of each point in one cluster to the points in other nearby clusters. Let's perform the analysis of the Silhouette score:

- **+1 Silhouette Score**: This score indicates that the sample we are using is very far away from its nearby cluster.
- **0 Silhouette Score**: This score indicates that the sample used is on the decision boundary that separates two nearby clusters.
- **-1 Silhouette Score**: This score indicates that the sample used is assigned to the wrong cluster.

The following is the formula for calculating the Silhouette score:

$$\text{silhouette score} = (p - q)/\max(p,q)$$

Here, p is the mean distance to the points in the nearest cluster, and q is the mean intra-cluster distance to all the points.

Davies–Bouldin index

Davies–Bouldin index, another good metric for clustering analysis, helps us get the answer to the following two questions:

- Is there enough space between the two clusters?
- What is the density of clusters?

The following is the formula for calculating the DB index:

$$DB = \frac{1}{n}\sum_{i=1}^{n} max_{j\neq i}\left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)}\right)$$

Here, n is the number of clusters, and σi is the average distance of all points in cluster i from the cluster centroid ci.

The lower the Davies–Bouldin index value, the better the cluster density.

Dunn index

Dunn index works the same as Davies–Bouldin index but they both differ in the following points:

- Dunn index considers only those clusters in a clustering model that are close with each other, whereas Davies–Bouldin index considers the separation of each one of the clusters.
- Dunn index boosts when the performance of the clustering model boosts, whereas

Davies–Bouldin index boosts in the case of dense clusters.

The following is the formula for calculating the DB index:

$$D = \frac{min_{1\leq i<j\leq n} p(i,j)}{max_{1\leq i<k\leq n} q(k)}$$

Here, *i*, *j*, *k* are the indices for clusters, *p* is the inter-cluster distance, and *q* is the intra-cluster distance.

Conclusion

In this chapter, we learned about Clustering, which is one of the most useful areas of unsupervised machine learning. We learned that the clustering model allows us to find the similarity and relationship patterns among data samples and after detecting the relationship patterns, based on similar features, it clusters the data samples into groups.

Despite forming the clusters in spherical form, there are four other methods, namely, Density-based method, Hierarchical-based method, Partitioning method, and Grid method to form the clusters. All these methods have been covered in this chapter.

We also covered the three commonly used clustering algorithms, namely, k-means, mean-shift, and hierarchical clustering along with their advantages and disadvantages. You also learned their implementation in the Python programming language.

In a nutshell, the basic pipeline of clustering task works as follows:

- The task of clustering starts with the data points, assigned to a cluster of their own.
- Once the data points are accessed, the model will compute all the centroids.
- The process will be repeated iteratively, and the model will move to the highest density of data points that is, cluster centroids.

The output of clustering tasks are the isolated groups of similar data points, and these are dissimilar from the data points of other groups.

The various performance matrices, namely Silhouette Analysis, Davies–Bouldin index, and Dunn index, for clustering have also been covered.

However, like supervised learning, in the case of unsupervised learning, we don't have that luxury because we are dealing with unlabeled data. That's the reason it is not easy to measure the performance of clustering tasks and the idea of testing seems a flawed premise. But even when labeled data is unavailable, there are numerous metrics that examine the quality of clustering results. Rather than measuring the validity of the ML model's prediction, these metrices can give us insight into how the clusters might change. The metrics we covered will also evaluate the comparative performance between the two ML models.

In the next chapter, you will learn about logic programming and how we can solve problems by using it.

Questions

1. What is clustering? Explain various methods to form clusters.
2. What are the various steps of the k-means clustering algorithm? Write a code to implement it in the Python programming language.
3. What are the various steps of the mean-shift clustering algorithm? Write a code to implement it in the Python programming language.
4. What is a hierarchical clustering algorithm? Explain.
5. What is the role of dendrograms in the agglomerative hierarchical clustering algorithm?
6. Explain various performance evaluation metrics for clustering algorithms.

---

[1] IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 14, NO. 5, Page 1003-1016, SEPTEMBER/OCTOBER 2002

[2] STING : A Statistical Information Grid Approach to Spatial Data Mining, VLDB 97 Conference

[3] Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications, 94-105 ACM SIGMOD Conference 1998: Seattle, Washington

CHAPTER 5

Solving Problems with Logic Programming

Introduction

If you have done coding before, there are chances that you are comfortable with some of the imperative programming languages like C++, Java, or Python. We know, in such a programming paradigm, every program is an organized list of instructions that modifies its state when executed. Isn't it the simplest and the most common way of computer programming? But it is not the focus of this chapter.

Instead, in this chapter, we are going to learn about a different computer programing paradigm in which the program uses facts and rules to express the problem. This kind of programming paradigm is called logic programming. You will also get to know about the building blocks of logic programming. Furthermore, with the help of some examples, we will also implement logic programing in the Python language.

Structure

In this chapter, we will discuss the following topics:

- Logic programming
- Building blocks of logic programming
- Python packages for logic programming
- Implementation examples

Objectives

After studying this chapter, you should be able to implement logic programming in the Python programming language. With the help of two useful implementation examples, namely, Checking and generating prime numbers and solving Zebra puzzle, you will learn to solve problems in the logic programming domain. You will also be able to install two useful python packages, named Kanren and SymPy.

Logic programming

We all are familiar with the term logic or more precisely formal logic in one or another sense. In layman language, logic may be defined as the study of what comes after what. On the other hand, in technical terms, logic may be defined as the study of principles of correct reasoning. For example, if A = B and B = C then we can easily infer that A = C.

Let's now talk about logic programming. Logic programming, a combination of two individual words logic and programming, is a programming paradigm in which a program is a database of relations, that is, knowledge made of facts and rules. In simple words, in logic programming, a program is a set of organized instructions expressing facts and rules about a particular problem domain. A graphical representation of logic programming is given as follows:

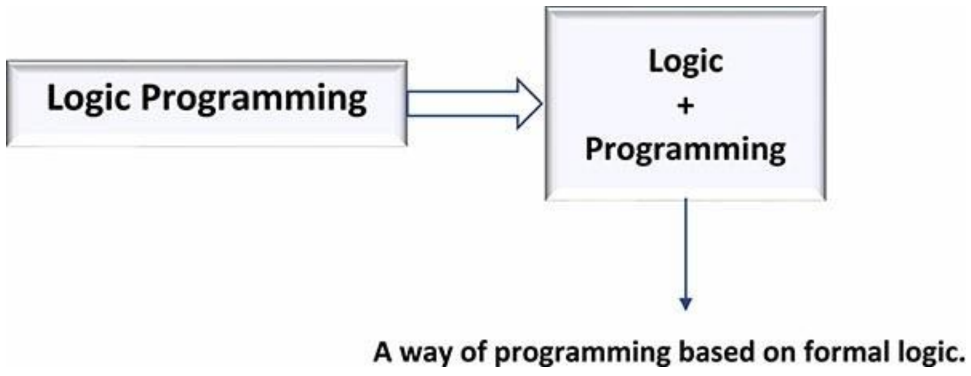


Figure 5.1: *Logic Programming*

Building blocks of logic programming

In order to solve problems, logic programming uses facts and rules. These facts and rules, hence, are called the building blocks of logic programming. Let's know more about facts and rules in detail:

- **Facts**: The simplest definition of facts is that these are true statements. For example, the national game of India is Hockey. In the case of logic programming, the facts are the true statements about the program and the data itself.
- **Rules**: To achieve the given goal, along with facts, logic programming also needs constraints that can lead us to conclusions about the problem domain. Rules, written as logical clauses to express facts, are such constraints in logic programs.

Syntax of rules: The following statement is the syntax of rules:

X->Y1,Y2 …,Ym

We can read the preceding statement as:

X if Y1 and Y2 and …and Ym

Here, *X* is the head of the preceding rule and *Y1, Y2, …, Ym* is the body of that rule.

A rule that does not have any body is called fact. For example, *X* from the preceding stated rule.

In a nutshell, facts and rules are the backbones of logic programs with the help of which it achieves the given goal and leads to a conclusion.

Useful Python packages for logic programming

The following are the two useful Python packages with the help of which we can start logic programming:

- **Kanren**: Kanren is one of the important Python packages used for logic programming. We can easily express logic as facts and rules. It helps the programmer to simplify its code for business logic as well. The following is the Python command to install it:

  ```
  pip install kanren
  ```

- **SymPy**: SymPy, a library for symbolic mathematics, is another useful python package for logic programming. It's almost a full-featured **CAS** (**Computer Algebra System**). The following is the python command to install it:

  ```
  pip install sympy
  ```

Implementation examples

In this section, we will implement two logic programming examples in the Python programming language.

Checking and generating prime numbers

Logic programming can help us find prime numbers from a list of numbers as well as generate them. The following steps show how to check and generate prime numbers:

First, by using Kanren Python library, import the required packages:

```
from kanren import isvar, run, membero
from kanren.core import success
from kanren.core import fail
from kanren.core import goaleval
from kanren.core import condeseq
from kanren.core import eq
from kanren.core import var
from sympy.ntheory.generate import prime, isprime
import itertools as it
```

Next, let us define a function for checking prime numbers from the given list of numbers:

```
def prime_check(M):
  if isvar(M):
     return condeseq([(eq,M,p)] for p in map(prime, it.count(1)))
  else:
     return success if isprime(M) else fail
```

Now, declare a variable and use this to find the prime numbers from the list of numbers:

```
M = var()
print((set(run(0,M,(membero,M,(1,3,4,5,6,7,9,10,11,12,14,15,20,21,22,
23,29,30,41,44,52,55,59,61,89)),(prime_check,M)))))
```

Output:

```
{3, 5, 7, 41, 11, 61, 23, 89, 59, 29}
```

We can also generate prime numbers. Let's generate the first 15 prime numbers:

```
print((run(15,M,prime_check(M))))
```

```
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47)
```

Solving the puzzles

Many puzzles like Sudoku, N-queen, 8-puzzles, Zebra puzzle can be solved by using logic programming. Here, we will solve a variant of Zebra Puzzle, which was published by Life International magazine in 1962. In this puzzle, they provided a series of clues about 5 houses containing people of different nationalities and animals along with other things too. The readers need to figure out which house had the animal Zebra in it.

First, let us check the variant of the puzzle to be solved:

```
There are five houses.
The English man lives in the red house.
The Swede has a dog.
The Dane drinks tea.
The green house is immediately to the left of the white house.
They drink coffee in the green house.
The man who smokes Pall Mall has birds.
In the yellow house they smoke Dunhill.
In the middle house they drink milk.
The Norwegian lives in the first house.
The man who smokes Blend lives in the house next to the house with cats.
In a house next to the house where they have a horse, they smoke Dunhill.
The man who smokes Blue Master drinks beer.
The Japanese smokes Prince.
The Norwegian lives next to the blue house.
They drink water in a house next to the house where they smoke Blend.
```

Programmatically, we can solve such a problem in a number of ways. If we can make a table to solve it, we can brute force it. Isn't it? But then it's going to be a big table. For this, we can dump that table into a database and query it with Structured Query Language (SQL). But, it's not that easy due to the following kind of statements:

```
The man who smokes Blend lives in the house next to the house with cats.
```

Such kind of statements declare a relationship between some *X* and some other *X*. Here, *X* represents the house. *X* can be anything but the most important thing in these kinds of statements is the word 'some'. Because it is the logic variable that acts as a placeholder for a value without even specifying a single actual value. That's the reason, the solver can try out different possible solutions before landing on a final one.

On the other hand, we have a logical solution with the help of logic programming. For this, the solver should think of each clue as a rule. Consider all the rules and tell what values will satisfy all of them. The following are the two examples given:

```
(eq, (var(), var(), var(), var(), var()), houses)
```

The preceding example sets up **`houses`** as a list of logic variables, that is, **`houses`** equals a list of five logic variables. In other words, it indicates that each of the houses will itself contain a list of logic variables. Such an example does not represent an assignment but it actually declares equality.

```
(membero,('Englishman', var(), var(), var(), 'red'), houses)
```

The preceding example shows that one of them has two properties, namely, **`Englishman`** and **`red`**. The other three out of five should be filled by the puzzle solver and are left blank as logic variables.

In this way, we need to input all the constraints/rules. Once the input is complete, you can run the solver to find out what houses meet the requirements in rules for Zebra.

The solution of this puzzle using the Python logic programming are given as follows:

First, by using kanren Python library, import the required packages:

```
from kanren import *
from kanren.core import lall
import time
```

Next, let us define a function, namely, **`left_to_house()`** to check whose house is left to whose house and another function, namely, **`next_to_house()`** to check whose house is next to whose house:

```
def left_to_house(q, p, list):
  return membero((q,p), zip(list, list[1:]))
def next_to_house(q, p, list):
  return conde([left_to_house (q, p, list)], [left_to_house (p, q, list)])
```

Now, declare the variable named houses and define the rules/constraints as follows:

```
houses = var()
rules_for_zebrahouse = lall(
  (eq, (var(), var(), var(), var(), var()), houses),

  (membero,('Englishman', var(), var(), var(), 'red'), houses),
  (membero,('Swede', var(), var(), 'dog', var()), houses),
  (membero,('Dane', var(), 'tea', var(), var()), houses),
  (left_to_house,(var(), var(), var(), var(), 'green'), (var(), var(), var(),
  var(), 'white'), houses),
  (membero,(var(), var(), 'coffee', var(), 'green'), houses),
  (membero,(var(), 'Pall Mall', var(), 'birds', var()), houses),
  (membero,(var(), 'Dunhill', var(), var(), 'yellow'), houses),
  (eq,(var(), var(), (var(), var(), 'milk', var(), var()), var(), var()),
  houses),
  (eq,(('Norwegian', var(), var(), var(), var()), var(), var(), var(), var()),
  houses),
  (next_to_house,(var(), 'Blend', var(), var(), var()), (var(), var(), var(),
  'cats', var()), houses),
  (next_to_house,(var(), 'Dunhill', var(), var(), var()), (var(), var(), var(),
  'horse', var()), houses),
  (membero,(var(), 'Blue Master', 'beer', var(), var()), houses),
  (membero,('German', 'Prince', var(), var(), var()), houses),
  (next_to_house,('Norwegian', var(), var(), var(), var()), (var(), var(), var(),
  var(), 'blue'), houses),
  (next_to_house,(var(), 'Blend', var(), var(), var()), (var(), var(), 'water',
  var(), var()), houses),
  (membero,(var(), var(), var(), 'zebra', var()), houses)
)
```

Finally, we need to run the solver with the following constraints:

```
solutions = run(0, houses, rules_for_zebrahouse)
```

The following line of code will get the output from the solver:

```
Zebra_house = [house for house in solutions[0] if 'zebra' in house][0][0]
print ('\n'+ Zebra_house + ' had zebra.')
```

Output:

```
German had zebra.
```

Conclusion

In this chapter, we learned about the basic concepts of logic programming and implemented some of its examples in the Python programming language. We got to know that logic programming is a kind of programming paradigm that is based on "Logic" or more precisely "Formal Logic". We also learned about the building blocks of logic programming, which are facts and rules. Facts, we understood, are the true statements about the program and data, whereas rules are the constraints, which lead the solver to some conclusions. If we talk about the mathematical structure of these building blocks, rules have both head and the body, whereas facts only have the head and do not contain the body.

For implementing logic programming in the Python programming language, we discussed two useful packages, named Kanren and SymPy. You also learned the commands to install these packages in your computer system. You can use logic programming to solve a variety of problems such as N-queen, 8-puzzles, and Sudoku, and so on. With the help of two useful implementation examples, namely, *Checking and Generating Prime Numbers* and *Solving Zebra Puzzle*, we learned how to solve problems in this domain.

From our Zebra puzzle, we understood why logic programming is important to solve such puzzles. Some of the reasons are:

- To create tables to solve such puzzles is time consuming.
- It is difficult to solve and understand the statements that declare a relationship between two X variables.
- Logic variable acts as a placeholder for a value without even specifying a single value.

AI is the capacity of an artificial machine to act smartly, and Logic on the other side is important for AI as the intelligent agents need to know the facts about the environment in which they operate. The facts contain both declarative and procedural knowledge. For example, procedural knowledge – don't put the finger in the fire; declarative knowledge – putting your finger in the fire will burn it and you don't want to get hurt.

In the next chapter, you will learn about the basic concepts of **Natural Language Processing** (**NLP**) logic programming along with its implementation in the Python programming language.

Questions

1. What do you mean by Formal Logic? Also, explain logic programming.
2. What are facts and rules? How are they useful in solving problems using logic programming?

3. What are the uses of Kanren and SymPy Python packages? How can you install them?
4. Write a program in Python programming language to generate the first 10 prime numbers by using the logic programming paradigm.
5. What is a Zebra puzzle? Write a Python program to solve this puzzle using the logic programming paradigm.

CHAPTER 6

Natural Language Processing with Python

Introduction

What is our method of communication with others? It is our language, isn't it? It is the language in which we can speak, read, and write. In other words, we do things like making plans, making decisions, thinking, and so on, all in our natural language. But here one of the most important questions is that in this era of **Artificial Intelligence** (**AI**), can we similarly communicate with machines as we do with other human beings, that is, in our natural language?

In this chapter, we are going to learn about one of the hottest topics in the field of data science that concerns enabling machines to process and understand human language. You will also get to know about the working and phases of NLP. We will also cover important concepts like tokenization, stemming, lemmatization, chunking, Bag-of-Words (BoW) model, stop words, vectorization, and transformers. Furthermore, from an implementation perspective, we will be learning how to install and work with **Natural Language Toolkit** (**NLTK**) Python package.

Structure

This chapter is structured as follows:

- Natural Language Processing (NLP)
- Installing Python's NLTK package
- Understanding tokenization, stemming, and lemmatization
- Understanding chunking
- Understanding Bag-of-Words (BoW) model
- Understanding stop words
- Understanding vectorization and transformers
- Implementation examples

Objective

After studying this chapter, you should be able to install and use Python's NLTK package. You will learn some important NLP concepts such as tokenization, stemming, lemmatization, chunking, **Bag-of-Words** (**BoW**) model, stop words, vectorization, and transformers. You will also be able to implement those NLP concepts using the Python programming language.

Natural Language Processing (NLP)

Natural Language Processing (**NLP**) is a field of computer science, more precisely a field of AI that concerns enabling machines to comprehend and process the language in which we (humans) communicate. In technical terms, the main task of NLP is to program machines for understanding, analyzing, and processing the huge amount of natural language data.

Working of NLP

To understand the working of NLP, we first need to understand how we use our natural language. In our day-to-day life, we usually say a hundred or thousands of words. Similarly, other human beings infer those words and answer accordingly. Isn't it simple communication for us? It is simple for us because we humans can derive a context from what is said and how it is said. In the same way, rather than focusing on voice modulation, natural language processing does draw on contextual patterns.

Look at the following examples:

Question: Man is to women as king is to what?

You can easily interpret the preceding question and answer it. The question shows that man relates to the king, hence women can relate to the **queen**.

Answer: queen

We got the answer from our experience, but can the machine behave in the same way. Check the following steps to understand how the machines learn in the same way as we learn:

1. The first basic step is to feed enough data to machines so that they can use it and learn from experience.
2. Once the machine will get enough data, it will create word vectors using that data. It uses a deep learning algorithm for this task.
3. To provide answers like humans, the machine will perform simple algebraic operations on the word vectors that are created earlier from data.

Phases/logical steps in NLP

The following figure represents the phases/logical steps in NLP:

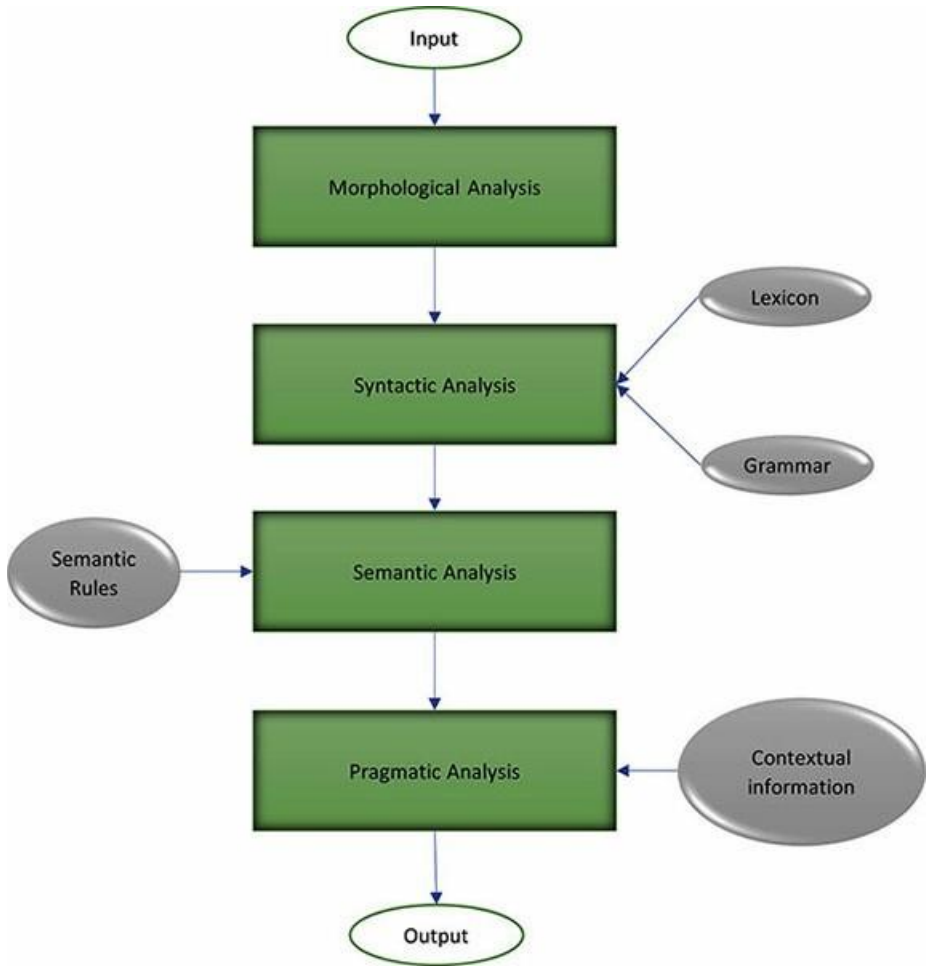


Figure 6.1: *NLP phases*

- **Morphological analysis**: It is the first phase of natural language processing. It explores the structure of words by breaking down the chunks of input language sentences into sets of tokens. For example, unfriendly = un-friend-ly. Morphological analysis is important for information retrieval, language modeling, and machine translation.
- **Syntactic analysis/parsing**: It is one of the most important phases/components of NLP. The following are the two main purposes of this phase:
    - Checking whether a sentence is well-formed or not.
    - Breaking up the sentence into a structure that gives us the syntactic relationship between various words.
- **Semantic analysis**: Drawing the exact dictionary meaning from the input text is an important phase of NLP. This phase is called semantic analysis. In this phase, the input text is checked for its meaningfulness. The output of this phase would be the object references. For example, the sentence hot ice cream does not have any meaning, hence it would be rejected by a semantic analyzer.
- **Pragmatic analysis**: In this phase, NLP fits the actual objects existing in each context with object references produced by a semantic analyzer. For example, the sentence *Keep*

the book in the rack on the table is having two semantic interpretations. From such sentences, the pragmatic analyzer needs to choose between possible semantic interpretations.

Implementing NLP

To implement NLP and build applications with it, one needs to have a great understanding of language along with a specific skill set. The tools will also play an important role to process the language efficiently. NLTK, Mallet, GATE, UIMA, Gensim, Open NLP, and Standford toolkit are some open-source tools, and some are developed by organizations to develop their applications.

In this chapter, we will be using NLTK. In comparison with the NLP tools, which are stated earlier, NLTK is much easier to use. It is written in Python hence the learning curve is fast.

Installing Python's NLTK Package

Prerequisites for installing are:

To install NLTK, we need to first install Python (if not yet installed) on our computer system. To install Python, first, go to the link **https://www.python.org/downloads/** and then select the OS on which you are running your computer system:

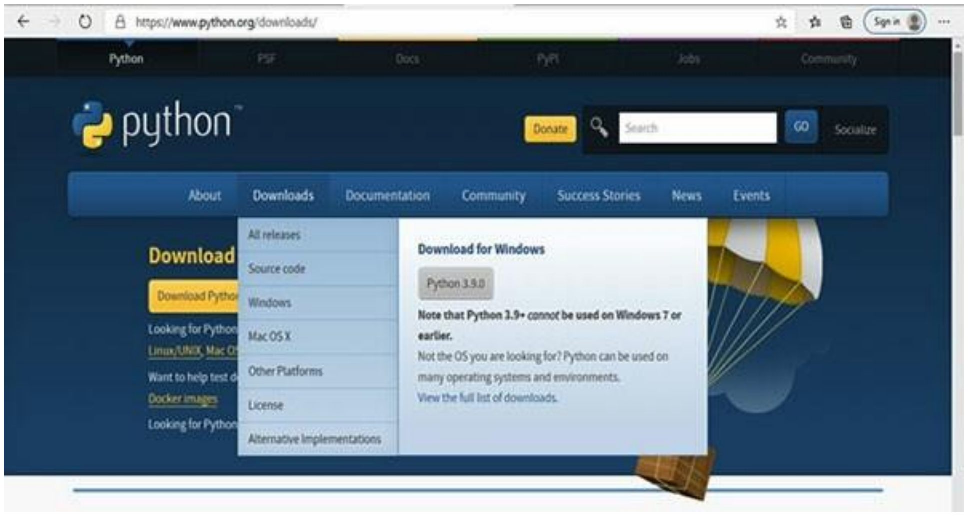

https://www.python.org/downloads/
Python
PSF
Docs
PyPI
Jobs
Community
python
Donate
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Download
All releases
Source code
Windows
Mac OS X
Other Platforms
License
Alternative Implementations
Download for Windows
Python 3.9.0
Note that Python 3.9+ cannot be used on Windows 7 or earlier.
Not the OS you are looking for? Python can be used on many operating systems and environments.
View the full list of downloads.

Figure 6.2: *Downloading Python*

Installing NLTK

Let us understand how we can install NLTK on various operating systems:

Windows OS

The following are the steps to install NLTK on MS Windows:

1. Open the Command Prompt (cmd) and go to the `pip` folder location.

2. Next, use the following command:

   ```
   pip3 install nltk
   ```

3. To verify the installation, open the PythonShell and type the following command:

   ```
   importnltk
   ```

If you have got no error after running the preceding command, means the installation is successful.

Mac/Unix OS

Use the following command to install NLTK on Mac/Unix OS:

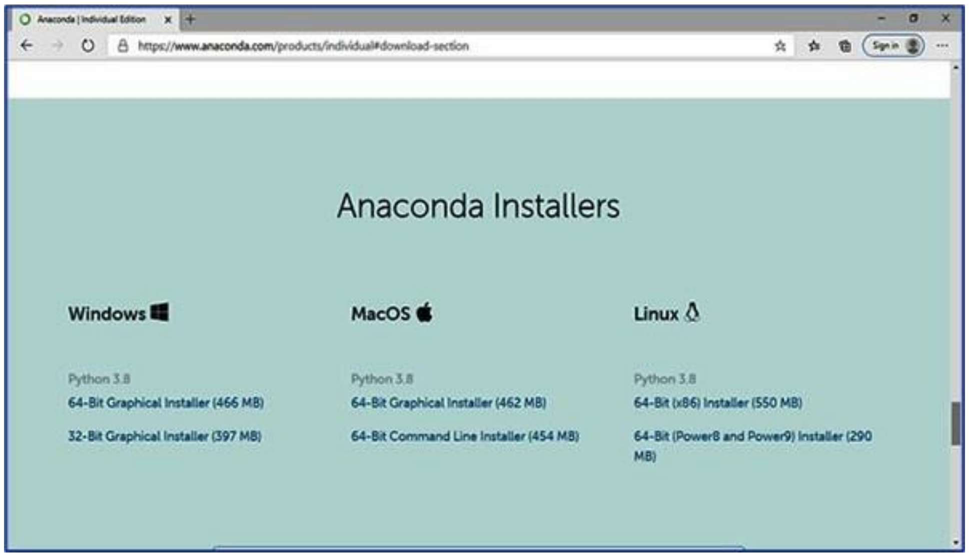
```
pip install -user -U nltk
```

In case if you do not have pip installed on your computer system, you first need to install that also.

Through Anaconda distribution installer

An alternative to install NLTK is the Anaconda distribution installer. The following are the steps to install NLTK:

1. Step1: Use this link https://www.anaconda.com/products/individual#download-section to install Anaconda. You can choose as per your OS.



Anaconda | Individual Edition

https://www.anaconda.com/products/individual#download-section

Anaconda Installers

Windows

Python 3.8

64-Bit Graphical Installer (466 MB)

32-Bit Graphical Installer (397 MB)

MacOS

Python 3.8

64-Bit Graphical Installer (462 MB)

64-Bit Command Line Installer (454 MB)

Linux

Python 3.8

64-Bit (x86) Installer (550 MB)

64-Bit (Power8 and Power9) Installer (290 MB)

Figure 6.3: *Installing Anaconda distribution*

2. Open the Anaconda Command Prompt run the following command:

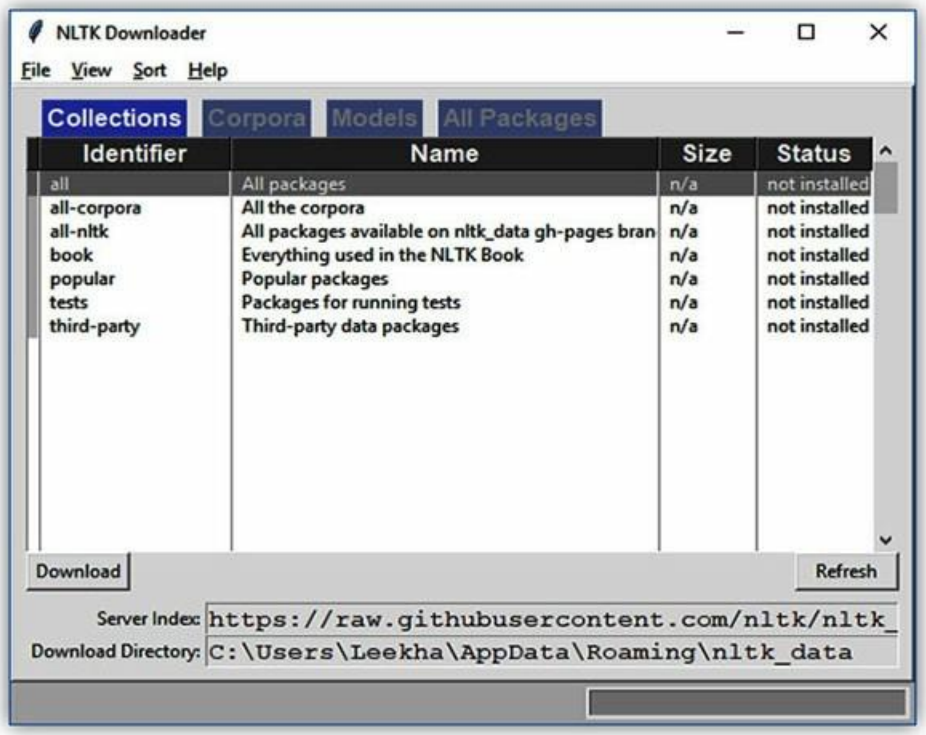
   ```
   conda install -c anaconda nltk
   ```

Downloading NLTK corpus

We learned how to install NLTK on various operating systems but to work with it we also need to download its corpus/datasets. Open the Python Shell and type the following commands:

```
importnltk
nltk.download()
```

After running the commands, we will get the following window:



Identifier	Name	Size	Status
all	All packages	n/a	not installed
all-corpora	All the corpora	n/a	not installed
all-nltk	All packages available on nltk_data gh-pages bran	n/a	not installed
book	Everything used in the NLTK Book	n/a	not installed
popular	Popular packages	n/a	not installed
tests	Packages for running tests	n/a	not installed
third-party	Third-party data packages	n/a	not installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_
Download Directory: C:\Users\Leekha\AppData\Roaming\nltk_data

Figure 6.4: *Downloading NLTK corpus*

Click on the `Download` button and you will get all the NLTK datasets/corpus on your computer system.

Understanding tokenization, stemming, and lemmatization

In this section, we will discuss some useful NLP concepts, such as, tokenization, stemming, and lemmatization. First, let’s get started with tokenization.

Tokenization

Language translation, QA systems, chatbots, sentiment analysis, and voice systems are some of the valuable applications that can be built by using NLP. To build such applications, the most important step is to understand the pattern in the text. Tokens (the smaller parts of the text) play a vital role in finding and understanding the pattern in the text.

In this way, we can define tokenization as the process of breaking up a piece of text into smaller parts. These smaller parts, which can be sentences and words, are called tokens. For example, sentences are made up of words, hence words are tokens in a sentence. Similarly, paragraphs are made up of sentences, hence sentences are tokens in a paragraph.

To achieve the process of tokenization in Python, the NLTK provides a package named **`nltk.tokenize`**.

As we have discussed before, we can have tokens from sentences (in the form of words) and paragraphs (in the form of sentences). Let's see how we can do this by using the **`nltk.tokenize`** package:

- **Tokenizing sentences**: One of the important text processing activities is to split the sentences into words. The Python package **`nltk.tokenoze`** provides the following modules to tokenize sentences into words:
    - **`Word_tokenize()`**: This module is used for the basic tokenization of sentences into words. Let us understand it's working with the help of the following example:

      ```
      import nltk
      from nltk import word_tokenize
      word_tokenize("This module can be used for basic tokenizing of
      sentences into words.")
      ```

      Output:

      ```
      ['This',
        'module',
        'can',
        ‘be’,
        'used',
        'for',
        'basic',
        'tokenizing',
        'of',
        'sentences',
        'into',
        'words',
        '.']
      ```
    - **TreebankWordTokenizer**: This method is invoked by **`word_tokenize()`** module with an assumption that the text has already been segmented into different sentences. The following are some characteristics of the treebank tokenizer method:
        - It splits standard contractions. For example, can't -> ca n't.
        - When followed by whitespaces, this method also splits off commas and single quotes.
        - It separates periods that appear at the end of the line.

      Let us understand it's working with the help of the following example:

      ```
      import nltk
      from nltk.tokenize import TreebankWordTokenizer
      sentence = '''Good vegan pizza cost $12.25\ninPheonix, Arizona. Please
      buy me\ntwo of them.\nThank you.'''
      TreebankWordTokenizer().tokenize(sentence)
      ```

      Output:

```
['Good',
  'vegan',
  'pizza',
  'cost',
  '$',
  '12.25',
  'in',
  'Pheonix',
  ',',
  'Arizona.',
  'Please',
  'buy',
  'me',
  'two',
  'of',
  'them.',
  'Thank',
  'you',
  '.']
sentence1 = "He'll save and invest for his retirement."
TreebankWordTokenizer().tokenize(sentence1)
```

Output:

```
['He', "'ll", 'save', 'and', 'invest', 'for', 'his', 'retirement',
'.']
sentence2 = "Hello, he can't go to market,"
TreebankWordTokenizer().tokenize(sentence2)
```

Output:

```
['Hello', ',', 'he', 'ca', "n't", 'go', 'to', 'market', ',']
```

- **WordPunctTokenizer**: As the name implies, this tokenizer splits punctuations into separate tokens. Let's understand it's working with the help of the following example:

```
import nltk
from nltk.tokenize import WordPunctTokenizer
sentence = "He'll save and invest for his retirement."
WordPunctTokenizer().tokenize(sentence)
```

Output:

```
['He', "'", 'll', 'save', 'and', 'invest', 'for', 'his', 'retirement',
'.']
```

You can see the difference in the output of **`TreebankWordTokenizer()`** and **`WordPunctTokenizer()`.**

- **RegexpTokenizer**: As the name implies, this method uses a regular expression for tokenizing sentences into words. Let's understand it's working with the help of the following example:

```
import nltk
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer("[\w']+")
sentence = "He'll save and invest for his retirement."
tokenizer.tokenize(sentence)
```

Output:

```
["He'll", 'save', 'and', 'invest', 'for', 'his', 'retirement']
```

- **Tokenizing paragraphs**: We understood word tokenization, that is, tokenizing sentences into words with the help of the `word_tokenizer()` module. Now, we will tokenize paragraphs, that is, tokenizing paragraphs into sentences. For this, NLTK provides us the `sent_tokenize()` module.

  But here the question arises if we have a word tokenizer then why do we need a sentence tokenizer? Both are important in one or another way. For example, if you would need to count average words in sentences, then you need both a sentence tokenizer as well as a word tokenizer. Let us understand its working and how sentence tokenizer is different from word tokenizer with the help of the following example:

  ```
  import nltk
  from nltk.tokenize import sent_tokenize
  text = "It shows the difference between word tokenizer and sentence
  tokenizer. It's a simple example."
  sent_tokenize(text)
  ```

  Output:

  ```
  ['It shows the difference between word tokenizer and sentence tokenizer.',
  "It's a simple example."]
  ```

Stemming

Stemming, an important part of the pipelining process in NLP and NLU, may be defined as the process of reducing a word to its root/base after removing suffixes and prefixes. In simple words, stemming produces morphological variants of a word. For example, the stemming algorithms or stemmers reduce the words eating, eats, and eaten to eat.

One of the most important applications of stemming is in information retrieval systems in search engines. Stemming reduces the size of the index because a search engine can store only the stems and it does not need to store all forms of the word.

- **Stemming algorithms**: The following figure has all the stemming algorithms/stemmers, which we will discuss in this section:

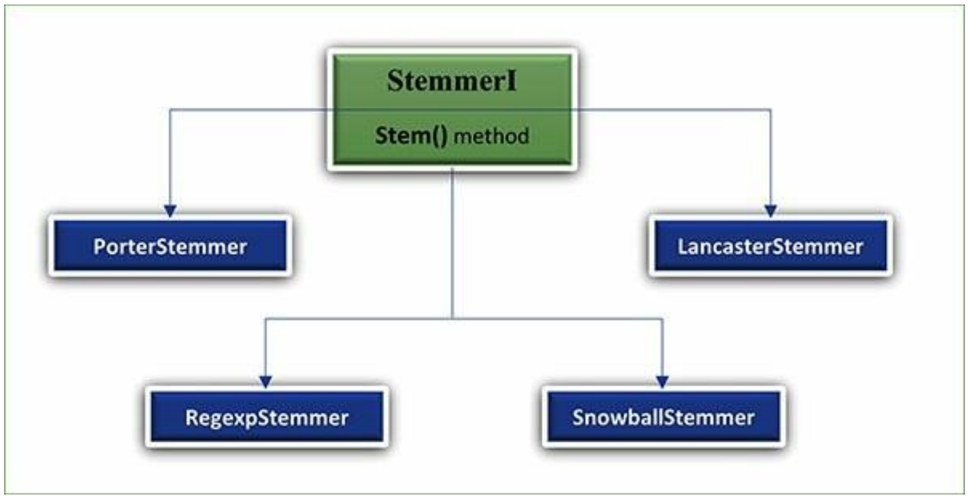


Figure 6.5: *Various stemming algorithms*

- **PorterStemmer**: NLTK provides us **`PorterStemmer`** class with the help of which we can easily implement a porter stemming algorithm, which is designed to remove as well as replace suffixes of English words. Check the following example:

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

stemming_word.stem('writing')
```

Output:

```
'write'
stemming_word.stem('working')
```

Output:

```
'work'
```

- **LancasterStemmer**: Another common stemming algorithm is the Lancaster Stemming algorithm, which is developed at Lancaster University. NLTK provides us the **`LancasterStemmer`** class with the help of which we can easily implement this algorithm. Check the following example:

```
import nltk
from nltk.stem import LancasterStemmer
stemming_Lanc = LancasterStemmer()
stemming_Lanc.stem('reads')
```

Output:

```
'read'
stemming_Lanc.stem('sweets')
```

Output:

```
'sweet'
```

- **RegexpStemmer**: Another useful stemming algorithm is the Regular Expression Stemming algorithm, which takes a single RE and removes prefix or suffix matching that expression. NLTK provides us **`RegexpStemmer`** class with the help of which we can easily implement this algorithm. Check the following example:

```
import nltk
from nltk.stem import RegexpStemmer
Regexp_stemmer = RegexpStemmer('ing')
Regexp_stemmer.stem('enjoying')
```

Output:

```
'enjoy'
```

```
Regexp_stemmer.stem('ingenjoy')
```

Output:

```
'enjoy'
```

- **SnowballStemmer**: The Snowball Steaming algorithm supports 15 non-English languages. To work with this algorithm, we first need to create an instance of the language and then call the method **`stem()`**. NLTK provides us **`SnowballStemmer`** class with the help of which we can easily implement this algorithm. Check the following example:

```
import nltk
from nltk.stem import SnowballStemmer
SnowballStemmer.languages#Languge supported by Snowball Stemmer
```

Output:

```
 ('arabic',
  'danish',
  'dutch',
  'english',
  'finnish',
  'french',
  'german',
  'hungarian',
  'italian',
  'norwegian',
  'porter',
  'portuguese',
  'romanian',
  'russian',
  'spanish',
  'swedish')
Language_French = SnowballStemmer('french')
Language_French.stem ('Bonjoura')
```

Output:

```
'bonjour'
Language_English = SnowballStemmer('english')
Language_English.stem ('Eating')
```

Output:

'eat'

```
Language_English.stem ('Reading')
```

Output:

```
'read'
```

Lemmatization

The lemmatization technique is similar to the stemming technique that we have discussed before but the difference is that lemmatization gives us the root word as output, rather than root stem, which we usually get after stemming. The technical name of the root word is lemma.

To achieve lemmatization, NLTK provides us **`WordNetLemmatizer`** class. This class is a thin wrapper around the wordnet corpus, and to find lemma, it uses **`morphy()`** function to the **`WordNet CorpusReader`**. Check the following example:

```
import nltk

from nltk.stem import WordNetLemmatizer

ex_lemmatizer = WordNetLemmatizer()
ex_lemmatizer.lemmatize('reading')
```

Output:

```
'reading'

ex_lemmatizer.lemmatize('sweets')
```

Output:

```
'sweet'
```

Difference between lemmatization and stemming

Although both techniques look similar, technically there are some differences between them, which are given in the following table:

Lemmatization	Stemming
Lemmatization always provides us a valid word as output. It means that lemma is an actual language word.	Stemming chops off the suffix or prefix, hence it might not provide a valid word as output always. It means that stem might not be an actual word.
Lemmatization looks at the meaning of the word.	Stemming looks at the form of the word.

Table 6.1: *Difference between lemmatization and stemming*

With the help of the following programming example, we will be able to understand the difference between both the techniques more clearly:

```
#Implementing Stemming
```

```
import nltk
from nltk.stem import PorterStemmer
ex_wordstemmer = PorterStemmer()
ex_wordstemmer.stem('believe')
```

Output:

```
Believ
```

```
#Implementing Lemmatization
import nltk
from nltk.stem import WordNetLemmatizer
ex_lemmatizer = WordNetLemmatizer()
ex_lemmatizer.lemmatize(' believe ')
```

Output:

```
believe
```

Understanding chunking

Chunking, one of the important processes in NLP, is used to extract phrases from the unstructured text. In other words, chunking is used to analyze the structure of a sentence to identify constituents such as noun groups, verb groups, verbs, and so on. Chunking is also called partial parsing and works on top of the POS tagging.

Importance of chunking

Do you think for text processing, simply breaking sentences into words is going to help? We must understand that a sentence involves various entities like a person, place, a date, a day, and so on. and an entity alone is of no use. That’s why phrases are more useful than individual words. Chunking does this job and breaks sentences into phrases to yield some meaningful results.

Let’s understand this concept with the help of the hierarchical structure of a sentence. The hierarchical structure of a sentence consists of the following components:

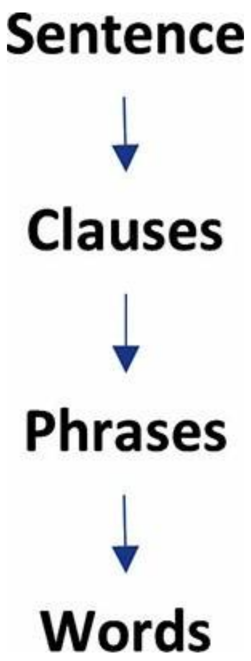


Figure 6.6: *Hierarchical structure of a sentence*

The last component is words that make up phrases. These are the following five categories of phrases:

- **Noun Phrase** (**NP**)
- Verb **Phrase** (**VP**)
- **Adjective Phrase** (**ADJP**)
- **Adverb Phrase** (**ADVP**)
- **Prepositional Phrase** (**PP**)

Once you know the categories of phrases, it is also important to understand the phrase structure. Phrase structure is also called as constituency grammar because it is based on the constituency relation. The following is an example of phrase structure, which can be understood in terms of NP and VP:

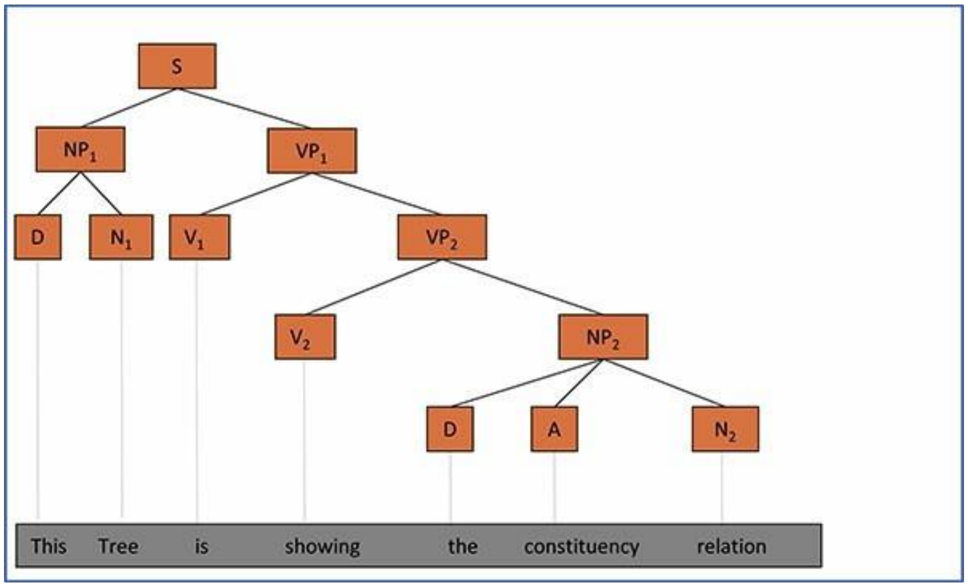


Figure 6.7: *Example of phrase structure*

Example:

Follow the steps to implement noun–phrase chunking in Python:

1. **Chunk grammar definition**: It is the first step for implementing noun–phrase chunking. Here we need to define the grammar for chunking, which would be containing the rules that need to be followed.
2. **Chunk parser creation**: Once you define the chunk grammar, it is time to create a chunk parser, which will parse the grammar and produce the output in tree format.

The following is an easy-to-understand Python recipe of chunker based on regular expression pattern:

```
import nltk

S = [("This", "DT"),("book", "NN"),("has","VBZ"),("ten","JJ"),("chapters","NNS")]

chunker=nltk.RegexpParser(r'''

NP:{<DT><NN.*><.*>*<NN.*>}
}<VB.*>{
''')

chunker.parse(S)

Output=chunker.parse(S)

Output.draw()
```

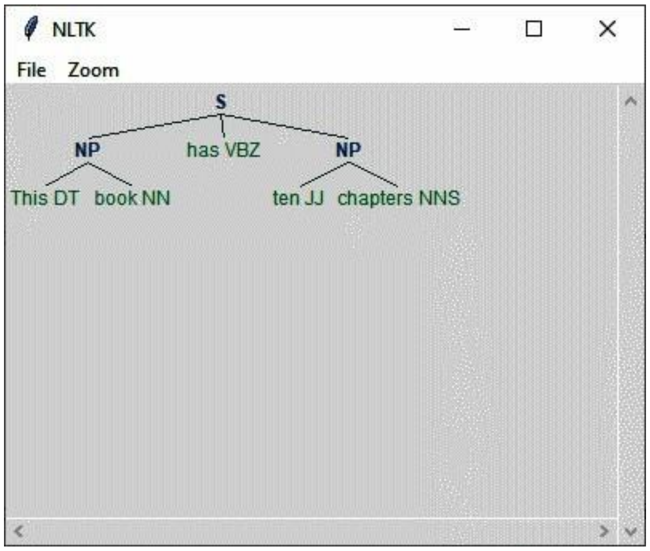


Figure 6.8: *Tree-like structure of the sentence after chunking*

Understanding Bag-of-Words (BoW) model

Bag-of-Words (**BoW**) is an NLP technique of text modeling, which is used to extract the features from the text. The BoW is a representation of the text describing the existence of words inside a document. It is called a bag of words because it is only concerned with the occurrence of the words in the document and any kind of information about the structure of words is discarded.

Why the BoW algorithm?

Why do we need BoWs? Is there any issue with simple and easy text?

- We know that the text is unstructured, and **Machine Learning** (**ML**) algorithms require structured and fixed-length input data. With the help of the BoW technique, we can easily convert raw data (variable-length texts) into a fixed-length vector.
- We also know that ML algorithms work well with numeric data rather than textual data. With the help of the BoW technique, we can easily convert the textual data into its equivalent vector of numbers.

Example:

The following is an example to understand how the BoW technique converts text into vectors:

- **Sentence 1**: Bag-of-Words model is a very useful NLP technique.
- **Sentence 2**: Bag-of-Words model is used to extract the features from the text.

Considering the preceding sentences, we have the following 16 distinct words:

- bag
- of
- words

- model
- is
- very
- useful
- NLP
- technique
- used
- to
- extract
- the
- features
- from
- text

We see that the vocabulary has 16 distinct words. We can use a fixed-length representation of 16 with one position in the vector for scoring each word. The scoring method is very simple, use 1 for the presence of each word otherwise use 0.

The following table will represent the scoring of the Sentence 1:

Word	Frequency
Bag	1
Of	1
Words	1
Model	1
Is	1
Very	1
Useful	1
NLP	1
Technique	1
Used	0
To	0
Extract	0
the	0
Features	0
from	0
Text	0

Table 6.2: *Scoring of the Sentence 1*

We can write the preceding frequencies in vector as follows:

Sentence 1 - [1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0]

Similarly, the following table will represent the scoring of Sentence 2:

Word	Frequency
Bag	1
Of	1
Words	1
Model	1
Is	1
Very	0
Useful	0
NLP	0
Technique	0
Used	1
To	1
Extract	1
the	1
Features	1
from	1
Text	1

Table 6.3: *Scoring of the Sentence 2*

We can write the preceding frequencies in vector as follows:

Sentence 2 - [1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1]

Implementing the BoW algorithm using Python

To implement the preceding BoW model, we will use the function named **`CountVectorizer()`** from the Scikit-learn Python library:

```
from sklearn.feature_extraction.text import CountVectorizer

Sentences=['Bag of Words model is very useful NLP technique.', 'Bag of Words
model is used to extract the features from text.']

vector_count = CountVectorizer()

text_feature = vector_count.fit_transform(Sentences).todense()
print(vector_count.vocabulary_)
{'bag': 0, 'of': 7, 'words': 15, 'model': 5, 'is': 4, 'very': 14, 'useful': 13,
'nlp': 6, 'technique': 8, 'used': 12, 'to': 11, 'extract': 1, 'the': 10,
'features': 2, 'from': 3, 'text': 9}

print(text_ feature)
[[1 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1]
  [1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 1]]
```

The preceding output represents the feature vectors, that is, text to numeric form. They can now be used in ML algorithms.

Understanding stop words

What is one of the major tasks of pre-processing? It filters out useless data from our dataset. Isn't it? In NLP, such useless words are called stop words. Stop words usually refer to the most common words in a language that does not add much meaning to the sentence. For example, words such as *a*, *an*, *the*, *in*, *is*, *at*.

When to remove stop words?

There is no hard and fast rule on when to keep unwanted words out of our corpus. But it is recommended to remove stop words if you want to perform tasks such as language classification, text classification, sentiment analysis, spam filtering, auto-tag generation, and caption generation because while performing these tasks, stop words do not provide any information to our model. On the contrary, if your task is related to machine translation, question—answer problems, language modeling, or text summarization, it is better to keep these unwanted words because they may be a crucial part of these applications.

Removing stop words using the NLTK library

With the help of the following example, let's see how we can remove the stop words using the NLTK library:

```
#Importing the NLTK Python library stopwords
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

#Sentence with stop words
sample_sentence = "Hey, this is pretty cool. We will do more such cool things."

stop_words = set(stopwords.words('english'))

text_tokens = word_tokenize(sample_sentence)

#Filtering the stop words
filtered_sentence = [w for w in text_tokens if not w.lower() in stop_words]

filtered_sentence = []

for w in text_tokens:
  if w not in stop_words:
     filtered_sentence.append(w)

#Printing the sentence with stop words
print(f"Tokenized text with stop words: \n{text_tokens}")

#Printing the sentence without stop words
print(f"Tokenized text without stop words: \n{filtered_sentence}")
```

The output of the preceding Python script is as follows:

```
Tokenized text with stop words:
['Hey', ',', 'this', 'is', 'pretty', 'cool', '.', 'We', 'will', 'do', 'more',
'such', 'cool', 'things', '.']
Tokenized text without stop words:
['Hey', ',', 'pretty', 'cool', '.', 'We', 'cool', 'things', '.']
```

Understanding vectorization and transformers

Vectorization is a methodology to map words or phrases from the vocabulary to its corresponding vector of real numbers. These vectors of real numbers further used to find word similarities and word findings. In simple words, the process of converting words into numbers is called vectorization.

Vectorization techniques

The following are some of the useful vectorization techniques:

- **Count vectorization**: This is one of the simplest techniques to perform text vectorization. In this technique, a document term matrix is created, which is a set of dummy variables indicating whether a particular word appears in the document or not. Each individual cell in the document term matrix denotes the frequency (known as term frequency) of the word in a particular document, whereas the columns represent each word in the corpus. Let's understand it with an example:

  Example

  To implement this example, we will be using the **`CountVectorizer`** package of Scikit-learn Python library. This package is available under **`sklearn.feature_extraction.text`**.

```
from sklearn.feature_extraction.text import CountVectorizer
Data_corpus = [
   'This book is on Artificial Intelligence.',
   'This book implements programs in Python.',
   'And Python is one of the best programming language.',
   'Is this the first book?',
]
Count_vectorizer = CountVectorizer()
M = Count_vectorizer.fit_transform(Data_corpus)
print(Count_vectorizer.get_feature_names())

print(M.toarray())
['and', 'artificial', 'best', 'book', 'first', 'implements', 'in',
'intelligence', 'is', 'language', 'of', 'on', 'one', 'programming',
'programs', 'python', 'the', 'this']
[[0 1 0 1 0 0 0 1 1 0 0 1 0 0 0 0 0 1]
  [0 0 0 1 0 1 1 0 0 0 0 0 0 0 1 1 0 1]
  [1 0 1 0 0 0 0 0 1 1 1 0 1 1 0 1 1 0]
  [0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1]]

Count_vectorizer2 = CountVectorizer(analyzer='word', ngram_range=(2, 2))
M2 = Count_vectorizer2.fit_transform(corpus)
print(Count_vectorizer2.get_feature_names())
['and python', 'artificial intelligence', 'best programming', 'book
implements', 'book on', 'first book', 'implements programs', 'in python',
'is book', 'is one', 'is this', 'of the', 'on artificial', 'one of',
'programming language', 'programs in', 'python is', 'the best', 'the
first', 'this book', 'this is', 'this the']
```

- **N-Grams**: This is like the count vectorization technique because in this technique a document term matrix is created, and each individual cell denotes the frequency of the word in a particular document. The difference in the N-gram method is that the columns in the document term matrix represent all the columns of adjacent words of length n. In simple words, we can say that count vectorization is N-Gram where n =1.

  For example, “I like Artificial Intelligence” has four words, and n =4.

  For n = 2, that is, bigram, the columns would be- [“I like”, “like Artificial”, “Artificial Intelligence”].

  For n = 3, that is, trigram, the columns would be- [“I like Artificial”, “like Artificial Intelligence”].

  For n = 4, that is, four-gram, the columns would be- [“I like Artificial Intelligence”].

  The trade-off is between the number of N values because the smaller value of N may not be sufficient to provide the most useful information. In contrast, the high value of N will yield a huge matrix with loads of features.

- **Term Frequency–Inverse Document Frequency (TF–IDF)**: This technique is also similar to the count vectorization technique because in this technique a document term matrix is created, and each column represents an individual unique word. TF–IDF method is different in the sense that along with the term frequency, each cell also contains a weight value signifying the importance of that word for the document. Rather than taking the consideration of a word in a single document, it takes into consideration a word in the entire corpus. Let’s understand **Term Frequency** (**TF**) and inverse document frequency (IDF):

  - **Term Frequency (TF)**: As the name implies, it is the frequency of a word in a document. For a specified word, TF is defined as the percentage of the number of times a word(m) occurs in a particular document (n) divided by the total number of words in the document. The following is the formula for finding TF:

    tf (‘word’) = Frequency of ‘word’ appears in the document D/total number of words in the document

    For example, consider the following document:

    Mohan loves to play cricket.

    The term frequency value for the word Mohan will be tf(‘Mohan’) = 1/5.

  - **Inverse Document Frequency (IDF)**: IDF measures the importance of a particular word in the corpus, that is, how common that word is across all the documents in the corpus. For a specified word, IDF is defined as the logarithmic ratio of the number of total documents to the number of a document with a particular word. The following is the formula for finding IDF:

    idf(‘word’) =log(Total number of documents in corpus/number of document with ‘word’ in it)

    For example, suppose the word *the* is present in all the documents in a corpus of 500 documents. Then the *idf* for word *the* would be:

idf('the') = log(500/500) i.e., log(1) = 0

In this way, TF–IDF(term) = TF(term) * IDF(term).

The formula for finding TF–IDF is as follows:

$$W_{m,n} = tf_{m,n} * log\left(\frac{X}{df_m}\right)$$

Here,

W_{mn} is word m within document n

$tf_{m,n}$ is the frequency of m in n

df_m is the number of documents containing m

X is the total number of documents

Let's understand it with an example:

Example

To implement this example, we will use the **`TfidfVectorizer`** package of Scikit-learn Python library. This package is available under **`sklearn.feature_extraction.text`**.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
corpus = ["this is the first document."," this document is the second
document."," and this is the third one."," is this the first document."]
tfidfvectorizer = TfidfVectorizer()
vectors_list = tfidfvectorizer.fit_transform(corpus)
feature_names = tfidfvectorizer.get_feature_names()
print(f"Feature Names are: \n{feature_names}")
matrix = vectors_list.todense()
list_dense = matrix.tolist()
df=pd.DataFrame(list_dense, columns=feature_names)
print(df)
```

The following is the output of the preceding Python script:

```
Feature Names are:
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
        and  document     first        is       one    second       the  \
0  0.000000  0.469791  0.580286  0.384085  0.000000  0.000000  0.384085
1  0.000000  0.687624  0.000000  0.281089  0.000000  0.538648  0.281089
2  0.511849  0.000000  0.000000  0.267104  0.511849  0.000000  0.267104
3  0.000000  0.469791  0.580286  0.384085  0.000000  0.000000  0.384085

      third      this
0  0.000000  0.384085
1  0.000000  0.281089
2  0.511849  0.267104
3  0.000000  0.384085
```

Transformers

The paper titled *Attention is All You Need*[1] introduces a novel concept in NLP called Transformers. It aims to solve seq2seq (sequence-to-sequence) tasks along with handling the long-range dependencies with ease.

Read the following quotation from the previously mentioned paper:

"The Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution."

The transduction here means the conversion of the input sequence to the output sequence; hence, we can say that the basic concept of a Transformer is to handle the dependencies between input and output with self-attention and recurrence.

Let's understand the meaning of self-attention. According to the previously mentioned paper:

"Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence."

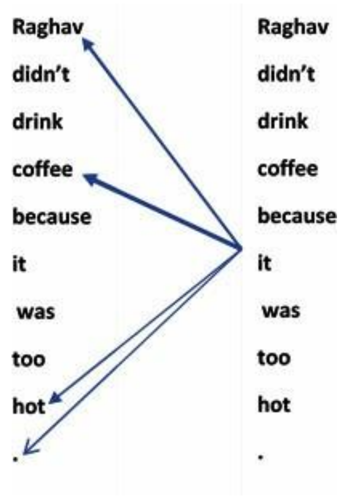


Figure 6.9: *Understanding self-attention*

Look at *Figure 6.9* and try to figure out what the term *it* refers to? Is it referring to the *coffee* or *Raghav*? It's simple for human beings but not for an algorithm to answer. When the NLP model processing the word *it*, self-attention tries to associate it with *coffee*.

In simple words, to get a better understanding of a specified word in the sequence, self-attention allows the NLP model to check other words in the input sequence.

We will use the `TfidfTransformer` package of Scikit-learn Python library for our examples in the next section. This package will transform a count matrix to normalized **Term Frequency** (**TF**) or **Term Frequency–Inverse Document Frequency** (**TF–IDF**) representation.

Some examples

In this section, we are going to solve two simple but useful NLP examples, namely, *Predicting the category* and *Gender finding* by using the Python programming language.

Predicting the category

Every word is important but categorizing the document is equally important. Categorizing a document means in which category of text a word falls. For example, we want to predict the given sentence falls in which category like email, sports, business, and so on. For our following example, we will be using 20 newsgroup datasets from the Scikit-learn Python library:

```
#Import the required packages
from sklearn.datasets import fetch_20newsgroups
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import CountVectorizer

#Defining five different category maps
c_map = {'talk.religion.misc': 'Religion', 'rec.autos':
'Autos','rec.sport.hockey':'Hockey','sci.electronics':'Electronics', 'sci.space':
'Space'}

#Creating the training set
t_data = fetch_20newsgroups(subset='train',
     categories=c_map.keys(), shuffle=True, random_state=5)

#Building a count vectorizer and extracting the term counts
v_count = CountVectorizer()
train_tc = v_count.fit_transform(t_data.data)
print("\nDimensions of training data:", train_tc.shape)

#Creating tf-idf transformer
tfidf = TfidfTransformer()
train_tfidf = tfidf.fit_transform(train_tc)

#Defining the test data
input_data = [
  'Columbia is the name of a space shuttle',
  'Hindu, isai, Sikh, Muslim all are religions',
  'We should drive safely',
  'Puck is a round disk made of hard rubber',
  'Television, Microwave, Mixer Grinder, Refrigerator, all uses electricity']

#Multinomial Naïve Bayes classifier training
classifier = MultinomialNB().fit(train_tfidf, t_data.target)

#Transforming input data by using count vectorizer
input_tc = v_count.transform(input_data)

#Transforming vectorized data by using tf-idf transformer
input_tfidf = tfidf.transform(input_tc)

#Predicting output categories
predictions = classifier.predict(input_tfidf)
for sent, category in zip(input_data, predictions):
  print('\nThe Input Data is:', sent, '\n Category:', \
  c_map[t_data.target_names[category]])
```

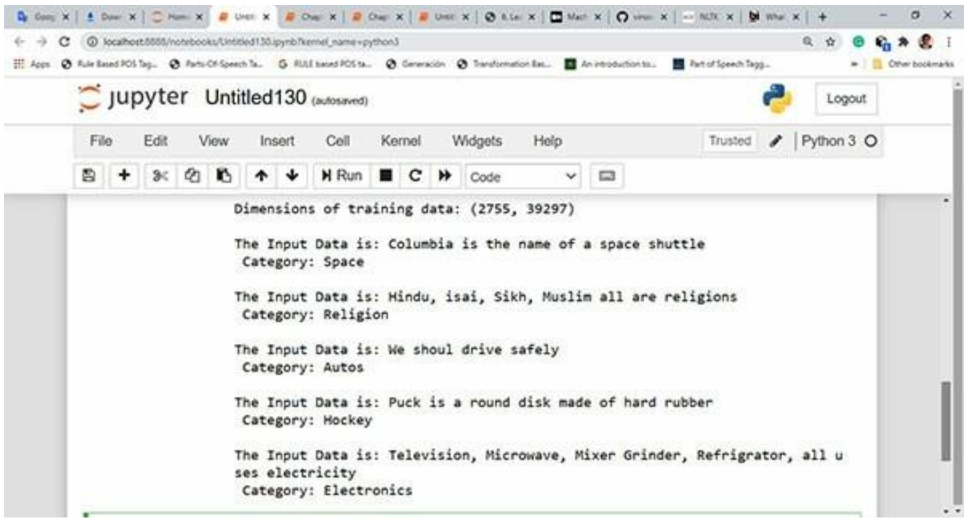
In the preceding program, first, we fetch the **`Sklearn 20newsgroup`** dataset and other useful packages. As we will be predicting the categories, we need to define different categories. We are defining five categories for this program. After that, we are creating the training set from our dataset.

Once done with the training set, we need to extract the term counts and for this, we build a count vectorizer by using **`Sklearn CountVectorizer()`** package. We now need to define the transformer and we are using the Sklearn TF-–IDF transformer. Along with the training set, the classifier also needs the test data. Hence, next, we will define the test dataset.

Now we need to fit the model and here, for this task, we are using a multinomial Naïve Bayes classifier. Next, we need to vectorize our input data by using a count vectorizer. After this, we need to pass this vectorize data to TF–IDF transformer to transform this data. At last, we will use the **`predict()`** function to predict the output categories.

Output:

The following is the output of the preceding Python script:


jupyter Untitled130 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
Dimensions of training data: (2755, 39297)

The Input Data is: Columbia is the name of a space shuttle
 Category: Space

The Input Data is: Hindu, isai, Sikh, Muslim all are religions
 Category: Religion

The Input Data is: We shoul drive safely
 Category: Autos

The Input Data is: Puck is a round disk made of hard rubber
 Category: Hockey

The Input Data is: Television, Microwave, Mixer Grinder, Refrigrator, all u
ses electricity
 Category: Electronics
```

Figure 6.10: *Predicted Output Categories*

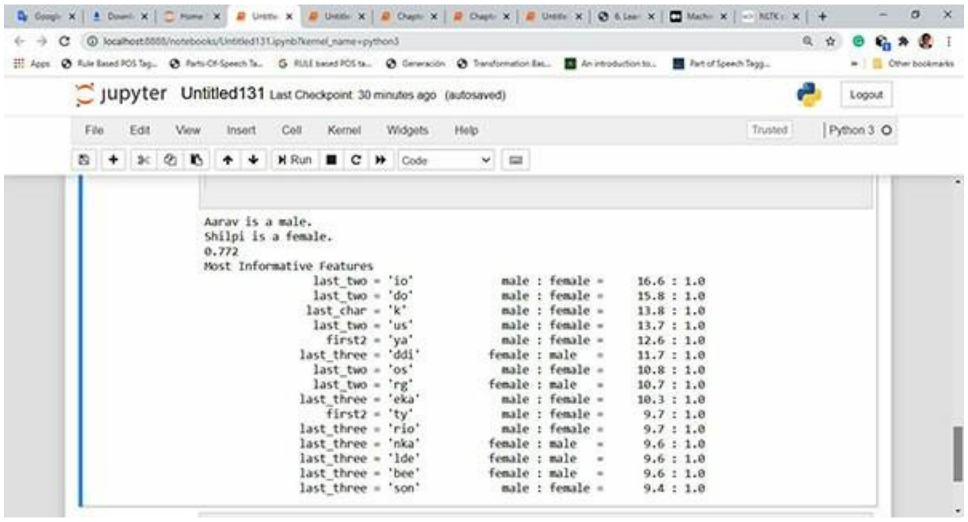
Gender finding

This example is to train a classifier that will predict the gender by providing names of males and females. First, the classifier will decide what features of the input are valid. Second, the classifier will decide how it can encode those features. We need to create a feature extractor function that will build a dictionary containing appropriate data about a given name (male or female).

```
#Import the required packages
import random
from nltk.corpus import names
from nltk import NaiveBayesClassifier
from nltk.classify import accuracy

#Defining the function to calculate features
def features(name):
  name = name.lower()
  return {
      'last_char': name[-1],
      'last_two': name[-2:],
      'last_three': name[-3:],
      'first': name[0],
      'first2': name[:2]
  }
names_M = [(name, 'male') for name in
names.words(r"C:/Users/Leekha/Desktop/malenames.txt")]
names_F = [(name, 'female') for name in
names.words(r"C:/Users/Leekha/Desktop/femalenames.txt")]
```

```
names_labels = names_M + names_F
random.shuffle(labeled_names)

# Splitting the dataset into training set and testing set.
train_set, test_set = featuresets[500:], featuresets[:500]

# Training the Naive Bayes classifier
classifier = NaiveBayesClassifier.train(train_set)
male_gender = classifier.classify(features('Aarav'))
female_gender = classifier.classify(features('Shilpi'))
print("Aarav is a {}.".format(male_gender))
print("Shilpi is a {}.".format(female_gender))

#Getting the accuracy
print(accuracy(classifier, test_set))

#Printing first 15 feature sets
classifier.show_most_informative_features(15)
```

Output:



```
Aarav is a male.
Shilpi is a female.
0.772
Most Informative Features
               last_two = 'io'           male : female =     16.6 : 1.0
               last_two = 'do'           male : female =     15.8 : 1.0
              last_char = 'k'            male : female =     13.8 : 1.0
               last_two = 'us'           male : female =     13.7 : 1.0
                 first2 = 'ya'           male : female =     12.6 : 1.0
             last_three = 'ddi'        female : male   =     11.7 : 1.0
               last_two = 'os'           male : female =     10.8 : 1.0
               last_two = 'rg'         female : male   =     10.7 : 1.0
             last_three = 'eka'          male : female =     10.3 : 1.0
                 first2 = 'ty'           male : female =      9.7 : 1.0
             last_three = 'rio'          male : female =      9.7 : 1.0
             last_three = 'nka'        female : male   =      9.6 : 1.0
             last_three = 'lde'        female : male   =      9.6 : 1.0
             last_three = 'bee'        female : male   =      9.6 : 1.0
             last_three = 'son'          male : female =      9.4 : 1.0
```

Figure 6.11: *Predicted Genders*

Conclusion

In this chapter, we learned about the basic concept of Natural Language Processing (NLP) and implement some of its examples in the Python programming language. We got to know that NLP, a field of Artificial Intelligence, concerns with enabling machines to comprehend and process the language in which we communicate, that is, our natural language. We also came to know about the working of NLP. We understood four phases of NLP, namely, morphological analysis, syntactic analysis, semantic analysis, and pragmatic analysis.

From the implementation perspective in the Python programming language, we discussed an extremely useful package called NLTK. You also learned the commands to install this package and its corpus/dataset on various operating systems. We also discussed some of the important concepts, such as tokenization, stemming, and lemmatization. We also implanted them by using the Python's NLTK package.

We also understood one important process of NLP called chunking, which is used to analyze the structure of a sentence to identify the constituents such as noun groups, verb groups, verbs, and so on. We also discussed phrase structure with the help of an example and implemented it in the Python programming language.

The BoW model, an important and most used model in NLP, is also discussed. It is used to extract the features from the text so that we can provide structured and fixed-length input data to ML algorithms. We also implemented an example of the BoW model by using the Python's Scikit-learn library.

You can use NLP to solve a variety of problems and build a variety of applications. With the help of two useful implementation examples, namely, *Predicting the category* and *Gender finding*, we learned how to solve problems in this domain.

In the next chapter, you will learn about the basic concept of speech recognition along with its implementation in the Python programming language.

Questions

1. What is Natural Language Processing (NLP)? How does it work?
2. Explain various phases of NLP.
3. What is tokenization? How do you implement it by using the NLTK package?
4. What is stemming? Explain along with examples the various stemming algorithms provided by the NLTK package?
5. What is lemmatization? Explain the difference between Stemming and Lemmatization.
6. Define chunking. How can we implement it in Python?
7. What is Bag-of-Words (BoW) model? Explain its importance with the help of an example.
8. What are stop words in NLP? How can we remove them by using the NLTK python package?
9. What is Vectorization? Explain various vectorization techniques with examples.
10. Explain the concept of Transformers in NLP.

---

1 https://arxiv.org/abs/1706.03762

CHAPTER 7

Implementing Speech Recognition with Python

Introduction

Other than our natural language, what else plays a prominent role in human–human interaction? It is speech, isn’t it? That is why it is quite natural for us to expect speech interfaces with machines. Over the last four decades, to ease the communication barrier between humans and machines, speech technology has come into existence. But over the past 6 to 7 years, I am sure everyone has doubtlessly noticed a quantum jump in the quality of a wide range of speech-enabled personal assistants like Amazon’s Alexa, Apples’ Siri, Google Assistant, or Microsoft’s Cortana, and voice-activated appliances as well as other similar technology around us. Likewise, soon, more than half of all the web searches will be done by voice.

In this chapter, we are going to learn about speech recognition, a technology that has unbelievable achievements from the first laboratory model to the commercial products available in the market today. You will also get to know about building a speech recognizer using the Python programming language. We will also discuss about the difficulties that one can face while developing a speech recognition system.

Structure

This chapter is structured as follows:

- Basics of speech recognition
    - Working of the speech recognition system
- Building a speech recognizer
    - Difficulties while developing a speech recognition system
    - Visualization of audio signals
    - Characterization of the audio signal
    - Monotone audio signal generation
    - Extraction of features from speech
    - Recognition of spoken words

Objective

The main objective of this chapter is to make you understand how you can develop a speech recognition system in the Python programming language. But before that, you must know about the basics and working of the speech recognition system. This chapter will fulfill this objective too. Last but not the least, this chapter will also make you aware of the difficulties you can face while developing an ASR system.

Basics of speech recognition

In the present era, among the tasks with the help of which we can interact with machines in our spoken language, automatic speech recognition that is also known as computer speech recognition is one of the fastest-growing and most commercially promising techniques. Speech recognition that allows the machines to identify the words, phrases, and sentences human beings speak is the first task among the three tasks of speech processing. The other two tasks consist of Natural Language Processing (allows machines to read, understand, and make sense of human languages) and Speech Synthesis (allows machines to convert normal language text into speech). In this chapter, we will focus on speech recognition.

Working of the speech recognition system

The following figure depicts the structure of a speech recognition system:

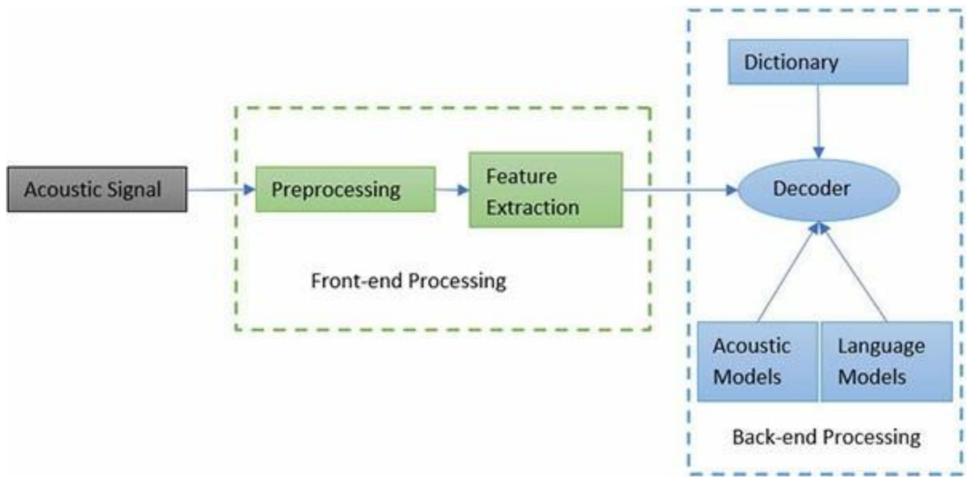


Figure 7.1: *Structure of speech recognition system*

The speech recognition system consists of the following two parts:

- Front-end processing
- Back-end processing

Front-end processing

Front-end processing consists of two techniques, namely, preprocessing and feature extraction. Let's discuss them in detail.

Preprocessing

The first part of a speech recognition system is preprocessing, which covers the following tasks:

- Analog-to-digital conversion
- Background noise filtering
- Pre-emphasis
- Blocking
- Windowing

Typically, a speech signal is a stream of 8-bit numbers at the rate of 10,000 numbers per second. It is a large amount of data and one of the biggest challenges of a speech recognition system is to reduce this huge data to some manageable representation. After the conversion of this electric signal conversion, next, it will filter the background noise and keep the **SNR** (**Speech-to-Noise Ratio**) say greater than 40 decibels.

Once background noise filtration is completed, it spectrally flattens the signal. This is called pre-emphasis, which actually amplifies the important areas of the spectrum. For example, in the spectrum region of 1KHz to 5KHz, hearing is more sensitive. By assisting the spectral analysis algorithm, pre-emphasis will amplify this area of the spectrum.

Feature extraction

As the name implies, feature extraction is used to find features of an utterance having acoustic correlations in the speech signal. These features can be computed on a frame-by-frame basis by using several feature extraction techniques. The following are some of the most used feature extraction techniques:

- **Linear predictive cepstral coefficient (LPCC)**: In the time domain, the LPCC model of speech production is given in following equation:

  $$s[n] \approx \sum_{k=1}^{n} a[k]s[n-k]$$

  In the preceding equation,

  s[n] denotes the speech signal samples,

  a[k] are the predictor coefficients,

  The total squared prediction error is given in the following equation:

  $$E = \sum\nolimits_{n} \left(s[n] - \sum_{k=1}^{p} a[k]s[n-k] \right)^2$$

  Here *p* is the order of the predictor.

  The main objective of linear predictive analysis is to determine the coefficients *a[k]* for every speech frame in such a way that error (*E*) is minimized.

  Now, given the LPC coefficients *a[k]*, the LPCC coefficients are computed using recursion as follows:

$$c[n] = a[n] + \sum_{k=1}^{n-1} \frac{k}{n} c[k]\, a[n-k] \qquad 1 \le n \le p$$

$$c[n] = \sum_{k=n-p}^{n-1} \frac{k}{n} c[k]\, a[n-k] \qquad n > p$$

These coefficients for every window of the speech will be used as features of the speech recognizer.

- **Mel frequency cepstral coefficient (MFCC)**: It may be defined as the real cepstrum of a windowed short-term signal derived from the fast Fourier transform of that signal. Block figure of MFCC is depicted in the following figure:

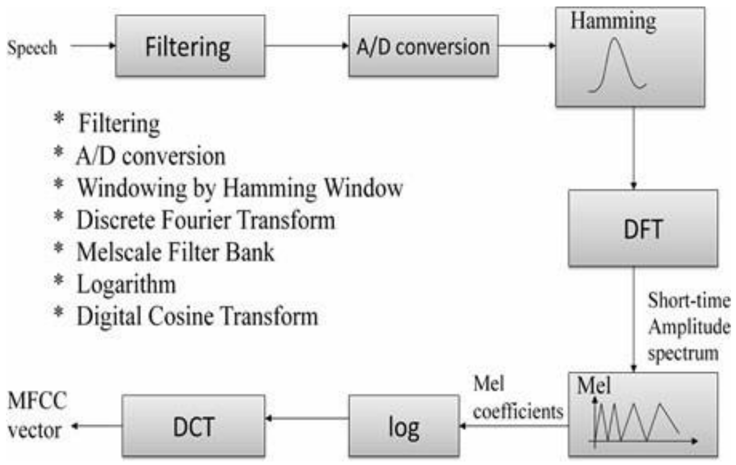


Figure 7.2: *Block diagram of MFCC*

The following are the given steps to compute MFCC feature vectors:

A 39-dimensional MFCC feature vector is computed from 25 ms of a window with 51 ms overlap using the following steps:

1. Pre-emphasize and weight the speech signal by a Hamming window. The Hamming window is defined as:

$$\left(0.54 - 0.46 \cos\left(\frac{2\pi i}{n-1}\right)\right)$$

   Where n is the total number of samples in an interval.
2. Now, take the Fourier transform of the weighted signals.
3. Next, average the spectral magnitude values using a triangular window at uniform spaces on the Mel-scale to consider auditory characteristics. The Mel-scale is defined in the following equation:

$$Mel(f) = 2595 \log_{10}\left(1 + \frac{f}{100}\right)$$

   Where f is the frequency in hertz.
4. Take a logarithm of the averaged spectral values. The convolution between sound source (pitch) and articulation (vocal tract impulse response) becomes addition due to the logarithm operations.

5. Take the inverse Fourier transform of the logarithmic spectral values. Remove the first coefficient and weight the next 12 cepstral coefficients using the following formula:

$$x_i = \left(1 + \frac{l}{2} sin\left(\frac{\pi i}{l}\right)\right) x_i$$

   Where x_i is the ith cepstral coefficient and l is the liftering coefficient.

6. Append normalized frame energy, producing a 13-dimensional feature vector.
7. Compute the first- and the second-order time derivatives of the 13 coefficients using the following regression formula:

$$\frac{\partial x_i}{\partial_T} = \frac{\Sigma_t T\left(x_i^{(t)} - x_i^{(-t)}\right)}{2\,\Sigma_t t^2}$$

$$\frac{\partial^2 x_i}{\partial_{T^2}} = \frac{\Sigma_t T\left(\frac{\partial x_i}{\partial_T}^{(t)} - \frac{x_i^{(-t)}}{\partial_T}\right)}{2\,\Sigma_t t^2}$$

   Where t is the time, and $x_i^{(t)}$ and $x_i^{(-t)}$ represents the tth following and previous cepstral coefficients in the time frame, respectively. The derivatives are appended to the original MFCCs, producing a 39-dimensional feature vector for every frame.

- **Perceptual linear prediction (PLP)**: PLP technique is based on the variation of LPC, which we discussed before. Considering human auditory perceptions, in PLP the critical filter bank consists of 17 filters spaced by one Bark on the Bark scale along with a frequency range of 0-5 kHz. The carrier frequency is defined by the following equation:

$$z = 6log\left(\frac{f}{600} + \sqrt{\left(\frac{f}{600}\right)^2 + 1}\right)$$

  Here, f is the frequency in Hz and z covers the range of 0-5 kHz by the 17 band pass filters *i.e.*, $0 \leq z \leq 17$ Bark. Each band is simulated by a spectral weighting, as shown in the following equation:

$$M_k = \begin{cases} 10^{z-x_k}\,; & z \leq x_k \\ 1\,; & y_k < z < x_k + 1 \\ 10^{-2.5(z-x_k-1)}\,; & z \geq x_k + 1 \end{cases}$$

  Here, $x_k = z_k - 0.5$ and z_k are the center frequencies. The benefit of this analysis is that it significantly improves recognition accuracy, especially in multi-speaker recognition.

Back-end processing

Back-end processing mainly involves pattern recognition, which will compare the feature vectors extracted in the front end with the machine's knowledge of speech. For pattern recognition, back-end processing constructs a language and an acoustic model. Let's discuss both of these models in detail.

Acoustic modeling

Acoustic modeling is used to establish a connection between acoustic information and phonetics. As we know that speech is a temporal signal, the speech unit is mapped to its acoustic counterpart using temporal models. Some of the most used models for acoustic modeling are **HMM** (**Hidden Markov Model**), **ANN** (**Artificial Neural Network**), and **DBN** (**Dynamic Bayesian Network**).

Language modeling

The main goal of language modeling is to generate the probabilities of a word W for which it uses the structural constraints available in the language. Language modeling uses the following two approaches:

- **Grammar-based approach**: For small vocabulary constraints tasks such as phone dialing, it uses the grammar-based approach.
- **Stochastic approach**: On the other hand, for large vocabulary constraints tasks such as broadcast news transcription, it uses the stochastic approach.

Building a speech recognizer

One of the centers of attention for **artificial intelligence** (**AI**) projects like robotics is ASR (automatic speech recognition). Without speech recognition, we cannot even imagine the cognitive interaction of humans and machines, especially robots. In this section, we will learn how to develop a speech recognizer in the Python programming language. But building a speech recognizer is not an easy task. It has a lot of difficulties and before deep diving into building a speech recognizer, we will first discuss about the difficulties.

Difficulties while developing a speech recognition system

The following are some of the difficulties the developer might face while developing a speech recognition system:

- **Vocabulary size**: Vocabulary size plays an important role in the success of a speech recognizer. The larger the size of the vocabulary, the harder it would be to recognize that. For example, for a voice-menu system, we need a small vocabulary size containing 2-100 words and on the other hand for a database retrieval task, we need a large vocabulary size containing around 10,000 words.
- **Channel quality**: Another important dimension that plays an important role is the characteristics of the channel. If we talk about human speech, we need high bandwidth with full frequency, whereas in the case of telephone speech, we need low bandwidth with limited frequency. It is tough to recognize telephonic speech.
- **Mode of speech**: We have three modes of speech-isolated, connected, and continuous speech. Among these three, continuous speech is tough to recognize.
- **Style of speech**: We have three styles of speech-formal, spontaneous, and conversational speech. Among these three, conversational speech is tough to recognize.

- **Noise**: Speech recognition is affected by signal-to-noise ratio (SNR), which can be high (>30 dB), medium (between 30 and 10 dB), and low (< 10dB). It is also affected by the type of background noise like stationary, and crosstalk by other speakers.

Now, let’s understand various steps to build a speech recognizer:

Visualization of audio signals

This is the first step in building a speech recognizer. With the help of this step, we can understand the structure of an audio signal. Recording and sampling are the two sub-steps to be followed while visualizing audio signals. A recording is needed in case if you want to read the audio signal from a file. Sampling is required to convert digitized signals into discrete numerical form. We should do sampling at a certain frequency because high frequency makes it feel as a continuous audio signal.

In the following example, by using the Python programming language, we will analyze an audio signal that is stored in a file.

```
#importing necessary packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

#Reading the stored audio file, returning sampling frequency and audio signal
freq_sampling, audio_sig =
wavfile.read("C:/Users/Leekha/Desktop/audio_Harvard.wav")

#Displaying various parameters of the audio signal
print('\nSignal shape:', audio_sig.shape)
print('Signal Datatype:', audio_sig.dtype)
print('Signal duration:', round(audio_sig.shape[0] / float(freq_sampling), 2),
'seconds')

#Normalizing the audio signal
audio_sig = audio_sig / np.power(2, 15)

#To visualize the signal, extracting first 200 values from it
audio_sig = audio_sig [:200]
time_axis = 1000 * np.arange(0, len(audio_sig), 1) / float(freq_sampling)

#Now, let’s visualize the stored audio signal
plt.plot(time_axis, audio_sig, color='red')
plt.xlabel('Time (ms)')
plt.ylabel('Amplitude')
plt.title('Audio Signal')
plt.show()
```

Output:

```
Signal shape: (445699,)
Signal Datatype: int16
Signal duration: 55.71 seconds
```

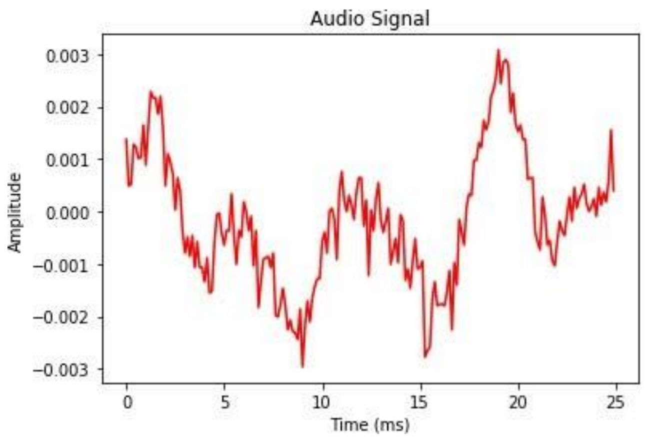


Figure 7.3: *Visualization of the audio signal*

Characterization of the audio signal

Characterization of the audio signal is another important step in which we will convert time-domain signal into a frequency domain. For such transformation, we use a mathematical tool named Fourier Transform.

Now, let's characterize the audio signal we have used previously for visualization purposes:

```
#importing necessary packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile

#Reading the stored audio file, returning sampling frequency and audio signal
freq_sampling, audio_sig =
wavfile.read("C:/Users/Leekha/Desktop/audio_Harvard.wav")

#Displaying various parameters of the audio signal
print('\nSignal shape:', audio_sig.shape)
print('Signal Datatype:', audio_sig.dtype)
print('Signal duration:', round(audio_sig.shape[0] / float(freq_sampling), 2),
'seconds')

#Normalizing the audio signal
audio_sig = audio_sig / np.power(2, 15)

#Extracting length and half length of the signal
len_signal = len(audio_sig)
half_len = np.ceil((len_signal + 1) / 2.0).astype(np.int)

#Using Fourier Transform
signal_frequency = np.fft.fft(audio_sig)

#Normalizing frequency domain signal
signal_frequency = abs(signal_frequency[0:half_len]) / length_signal
signal_frequency **= 2

#Extracting length and half length of the frequency transformed signal
len_fts = len(signal_frequency)

#Adjusting Fourier transformed signal for both even and odd case:
if len_signal % 2:
```

```
signal_frequency[1:len_fts] *= 2
else:
signal_frequency[1:len_fts-1] *= 2

#Extracting the power in dB and measuring frequency in kHz for x-axis
signal_power = 10 * np.log10(signal_frequency)
x_axis = np.arange(0, half_len, 1) * (freq_sampling / len_signal) / 1000.0

#Visualizing characterized signal
plt.figure()
plt.plot(x_axis, signal_power, color='green')
plt.xlabel('Frequency (kHz)')
plt.ylabel('Signal power (dB)')
plt.show()
```

Output:

```
Signal shape: (445699,)
Signal Datatype: int16
Signal duration: 55.71 seconds
```

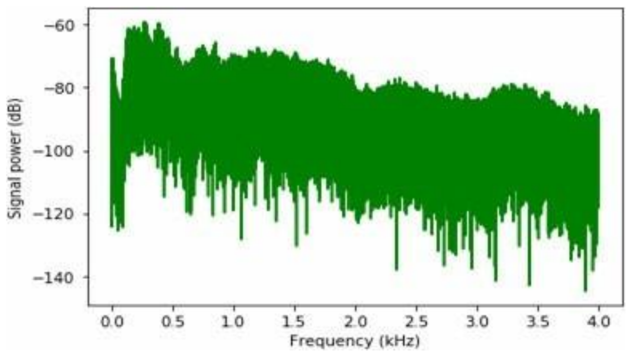


Figure 7.4: *Characterization of an audio signal*

Monotone audio signal generation

Here we will be generating the audio signal with some predefined parameters:

```
#importing necessary packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.io.wavfile import write

#File for saving the output audio signal
file_output = "C:/Users/Leekha/Desktop/audio_Harvard_monotone.wav"

#Specifying the parameters
duration = 35 # in seconds
freq_sampling = 44100 # in Hz
freq_tone = 784
min_val = -4 * np.pi
max_val = 4 * np.pi

#Generating the audio signal
t = np.linspace(min_val, max_val, duration * freq_sampling)
audio_sig = np.sin(2 * np.pi * freq_tone * t)
```

```
#Saving the audio signal file in the output file
write(file_output, freq_sampling, audio_sig)

#Extracting first 200 values for our graph
audio_sig = audio_sig[:200]
time_axis = 1000 * np.arange(0, len(audio_sig), 1) / float(freq_sampling)

#Visualizing the generated audio signal
plt.plot(time_axis, audio_sig, color='red')
plt.xlabel('Time in milliseconds')
plt.ylabel('Amplitude')
plt.title('Generated audio signal')
plt.show()
```

Output:

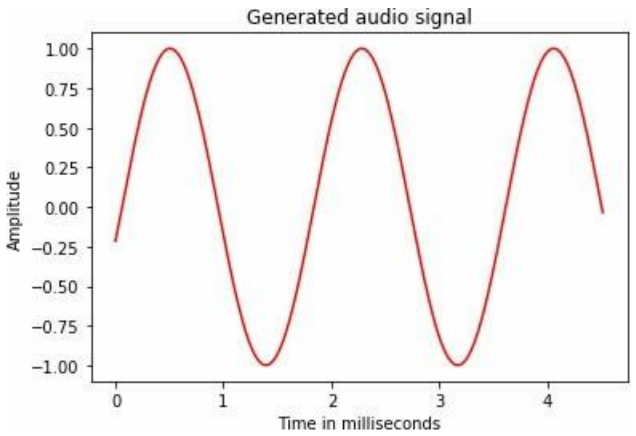


***Figure 7.5:** Monotone audio signal*

Extraction of features from speech

After converting the speech signal into the frequency domain, we need to extract its features. Feature extraction is one of the most important steps in building the speech recognizer. There are various techniques such as MFCC, PLP, PLP–RASTA, which can be used for this task. For our example, we will use the MFCC feature extraction technique.

```
#importing necessary packages
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
from python_speech_features import mfcc, logfbank

#Reading the stored audio file, returning sampling frequency and audio signal
freq_sampling, audio_sig =
wavfile.read("C:/Users/Leekha/Desktop/audio_Harvard.wav")

#Taking first 15000 samples for analysis
audio_sig = audio_sig[:15000]

#Exatracting MFCC features and printing its parameters
mfcc_features = mfcc(audio_sig, freq_sampling)
print('\nMFCC:\nNumber of windows =', mfcc_features.shape[0])
print('Length of each feature =', mfcc_features.shape[1])
```

Output:

```
MFCC:
Number of windows = 186
Length of each feature = 13
#Plotting and visualizing the MFCC features
mfcc_features = mfcc_features.T
plt.matshow(mfcc_features)
plt.title('MFCC')
```

Output:

```
Text(0.5, 1.05, 'MFCC')
```

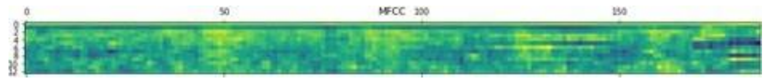


Figure 7.6: *MFCC features*

```
#Exatracting Filter bank features and printing its parameters
filterbank_features = logfbank(audio_sig, freq_sampling)
print('\nFilter bank:\nNumber of windows =', filterbank_features.shape[0])
print('Length of each feature =', filterbank_features.shape[1])
```

Output:

```
Filter bank:
Number of windows = 186
Length of each feature = 26

#Plotting and visualizing the Filterbank features
filterbank_features = filterbank_features.T
plt.matshow(filterbank_features)
plt.title('Filter bank')
plt.show()
```

Output:

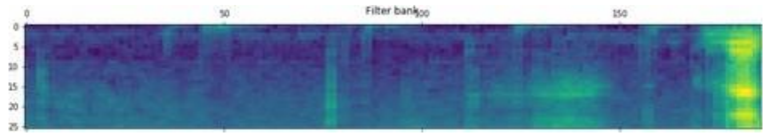


Figure 7.7: *Filter bank features*

Recognition of spoken words

For recognizing the spoken words, we will use Google Speech API in Python. For our example, we need to install the following Python packages:

- **PyAudio**: The command to install `PyAudio` package is `pip install Pyaudio`.
- **SpeechRecognition**: The command to install `SpeechRecognition` package is `pip install SpeechRecognition`.
- **Google-Speech-API**: The command to install Google-Speech-API package is `pip`

```
install google-api-python-client.

#importing necessary packages
import speech_recognition as srec

#creating an object
recording = srec.Recognizer()

#voice will be taken by Microphone() module as input
with srec.Microphone() as source:
recording.adjust_for_ambient_noise(source)
print("Say something:")
   audio = recording.listen(source)

#Google API will recognize the voice and provide output

try:
print("What you said is: \n" + recording.recognize_google(audio))
except Exception as e:
   print(e)
```

Output:

```
Say something:
What you said is:
speech recognition example
```

Conclusion

In this chapter, we learned about the basics of speech recognition and how to build a speech recognizer in the Python programming language. We got to know that speech recognition, which is one of the fastest-growing and most commercially promising techniques, is the first task among the three tasks of speech processing. It allows the machines to identify the words, phrases, and sentences human beings speak. We also came to know about the working of speech recognition system under which we understood its two parts, namely, front-end processing and back-end processing.

From the implementation perspective, we learned how to develop a speech recognizer in the Python programming language. We discussed the difficulties one can face while developing the speech recognition system. Vocabulary size, channel quality, speaking mode, speaking style, and type of noise are some of the common difficulties. In the next chapter, you will learn about the basic concepts of **Artificial Neural Network** (**ANN**) along with its implementation in the Python programming language.

Questions

1. What is automatic speech recognition (ASR)? How does it work?
2. What is feature extraction? Explain along with any two feature extraction techniques.
3. What is the role of acoustic modeling and language modeling in speech recognition? Explain.
4. What are some of the common difficulties one can face while building a speech recognition system?

5. Write down the Python program to extract features from a speech by using the MFCC feature extraction technique.
6. Write down the Python program for recognizing spoken words by using Google-Speech-API.

CHAPTER 8

Implementing Artificial Neural Network (ANN) with Python

Introduction

Hubots – are they some kind of humans or machines?

Human beings, with their intelligence, can quickly find the answer to the previous question. In fact, we humans can easily tell the difference between the two of them. Do you think a machine can tell the right answer?

Yes, machines can also tell the right answer but for making such predictions correctly, it must rely on algorithms like the **Artificial Neural Network** (**ANN**), which is inspired by the way our brain processes information by managing nonlinear relationships between inputs and outputs. There are both surprising similarities and differences in how humans think and how machines learn.

In this chapter, we will learn about the ANN, an efficient computing system whose central theme is borrowed from the biological nervous system. You will also get to know about the Python packages, which are useful for constructing the ANNs. With the help of some examples, we will understand how to construct the ANNs.

Structure

In this chapter, we will cover the following topics:

- Understanding of **Artificial Neural Network** (**ANN**)
- Installing useful Python packages for ANN
- Examples of building some neural networks:
    - Perceptron-based classifier
    - Single-layer neural networks
    - Multi-layer neural networks
    - Vector quantization

Objective

The main objective of this chapter is to make the learner understand how they can construct ANN in the Python programming language. The learner will understand the basic concepts of ANN along with some useful Python packages for building ANNs. The learner will also be able to build neural networks such as Perceptron-based classifiers, single-layer neural networks, multi-layer neural networks, and vector quantization.

Understanding of Artificial Neural Network (ANN)

ANNs are those computational models that are inspired by the way the human brain processes information using the biological nervous system. In other words, we can say that the ANN is a biologically inspired network of artificial neurons configured to process the nonlinear relationship between inputs and outputs in parallel like our brain does. Therefore, the ANNs are sometimes termed as **Artificial Neural Systems** and **Parallel Distributed Processing Systems**.

To understand the working of the ANN, we first need to understand the working of the human brain, that is, how our brain processes information using biological neurons.

A biological neuron

Does the following figure come into your mind when you hear the word neural network?

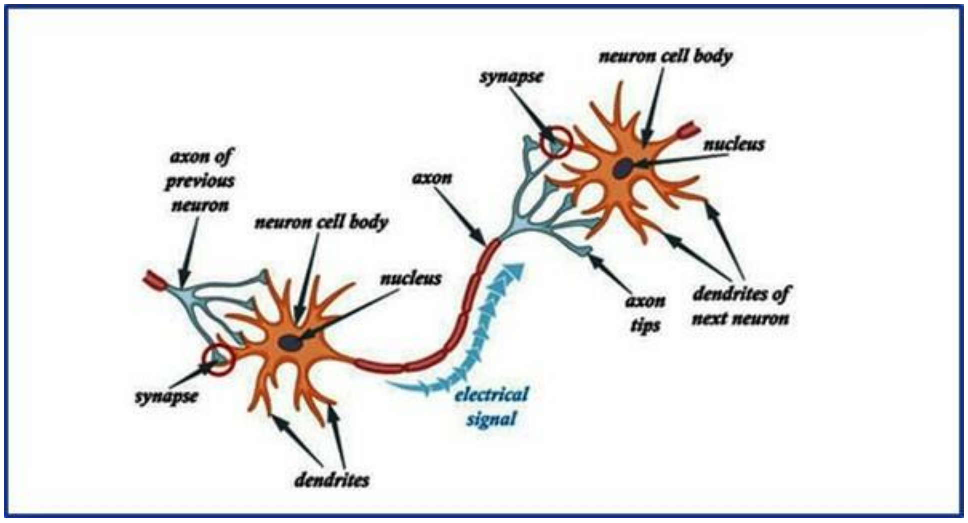


Figure 8.1: *Biological neuron credit: https://towardsdatascience.com*

As you can see in the preceding figure, a biological neuron consists of the following four parts:

- **Dendrites**: Dendrites, which look like branches of a tree, are responsible for receiving signals from other neurons they are connected to.
- **Soma**: It is located inside the cell body of a neuron. Its key task is to sum all the signals received from dendrites and generate the input.
- **Axon**: It is just like a wire through which a signal travels from one neuron to another.
- **Synapse**: It is the point of interconnection between two neurons. The higher the synaptic weight of the connections, the higher the amount of signal that will be transmitted between neurons. Based on weights, there are two types of synapses:
    - **Excitatory synapse**: A synapse is known as excitatory if the corresponding synaptic weight is positive.
    - **Inhibitory synapse**: A synapse is known as inhibitory if the corresponding synaptic

weight is negative.

For better understanding, the figure depicting the model of a biological neuron is shown here:

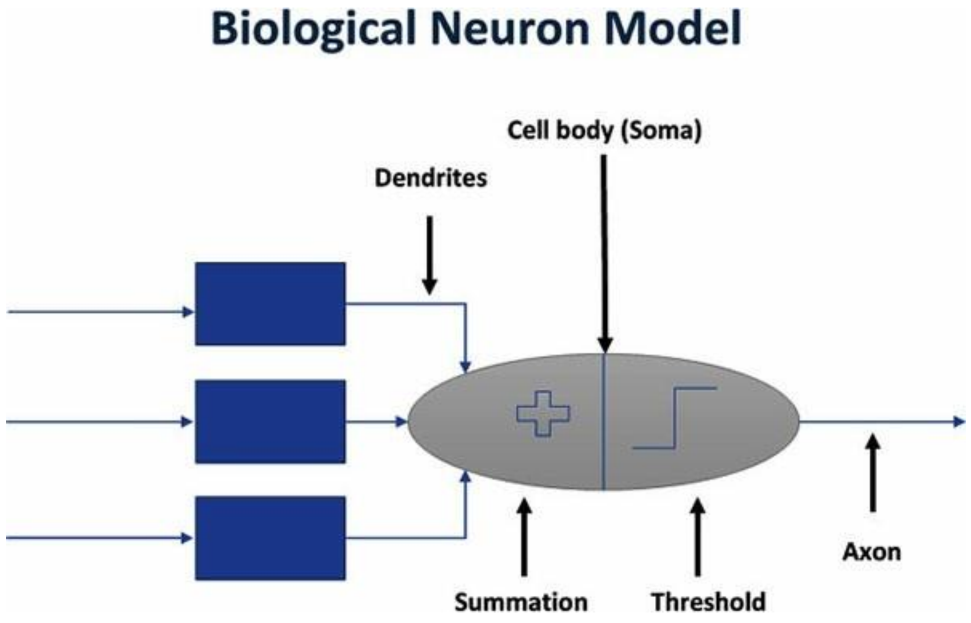


Figure 8.2: *Model of a biological neuron*

[Working of ANN]()

The following is the diagram of an ANN (refer *[figure 8.3]()*):

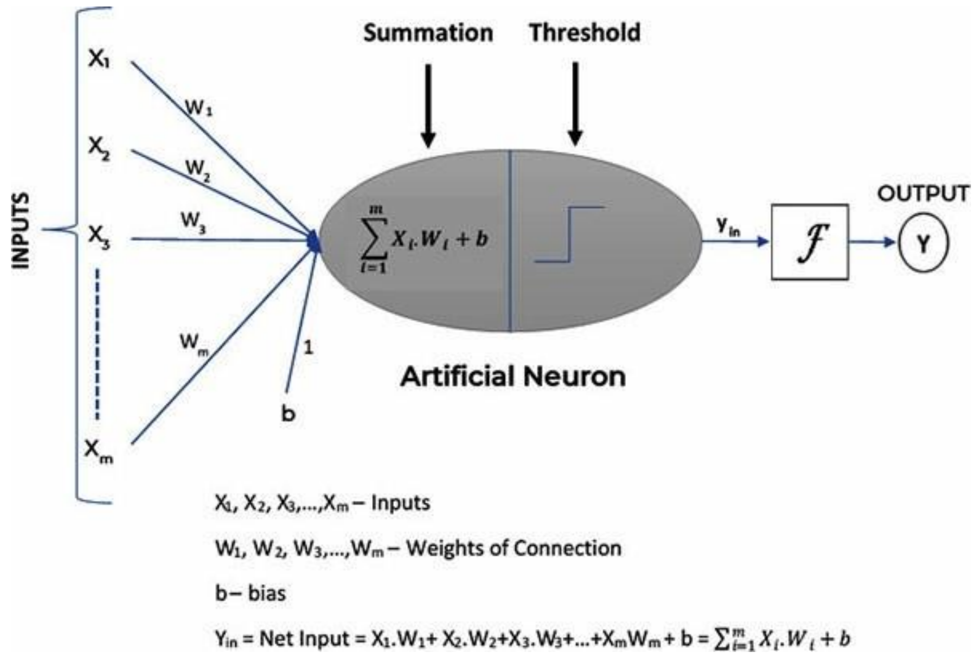

Summation
Threshold
INPUTS
X_1
X_2
X_3
X_m
W_1
W_2
W_3
W_m
1
b
$\sum_{i=1}^{m} X_i.W_i + b$
Y_{in}
$\mathcal{F}$
OUTPUT
Y
Artificial Neuron
$X_1, X_2, X_3,\ldots,X_m$ – Inputs
$W_1, W_2, W_3,\ldots,W_m$ – Weights of Connection
b – bias
Y_{in} = Net Input = $X_1.W_1 + X_2.W_2 + X_3.W_3 + \ldots + X_mW_m + b = \sum_{i=1}^{m} X_i.W_i + b$

Figure 8.3: *Artificial Neural Network*

From the preceding figure, we can see that the ANNs are a kind of weighted directed graph. The nodes are formed by artificial neurons and directed edges are connections between neuron inputs and outputs. These directed edges are having weights (representing the strength of interconnection among artificial neurons) on them.

Now let us understand the working of the ANNs with the help of the following steps:

1. First, it receives the input signal in various forms of information.
2. Once received, each input then is multiplied by its corresponding weight.
3. Now, all the weighted inputs will be summed up inside the artificial neuron, that is, the computing unit. Here, suppose if the weighted sum is zero, a bias will be added to make the output non-zero. The weight of the bias is always equal to 1.
4. At last, the sum of weighted inputs will be passed through an activation function.

Here the question arises as to what is an activation function?

To obtain the desired output, we need to apply some set of transfer functions applied over the input. These sets of transfer functions are called activation functions.

Let's see some of the most commonly used activation functions (broadly divided into two categories, linear and nonlinear):

Linear

Such activation functions are also called identity activation functions because the equation they have is similar to that of a straight line.

- **Equation**: F(x) = x
- **Range**: infinity to +infinity
- **Uses**: It is used only at the output layer that is, the last layer of neurons in ANN. The output layer produces the given outputs for the program.

The following figure depicts the linear activation function:

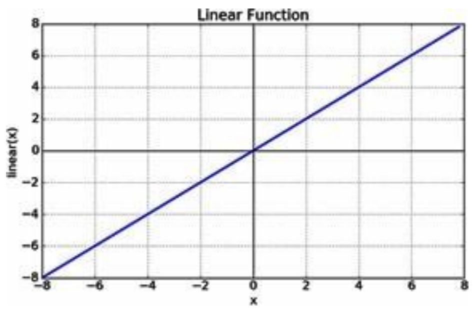


Figure 8.4: *Linear activation function*

Nonlinear

The following are some of the nonlinear activation functions:

- **Sigmoid (logistic)**: It is a kind of nonlinear activation function having an S-shaped curve, as shown in *figure 8.5*.

- **Equation**: F(x) = sigm(x) = 1/(1+e^{-x})
- **Range**: 0 to 1 (another commonly used range for sigmoid activation function is from -1 to 1).
- **Uses**: It is mainly used in the output layer for binary classification, which refers to predicting one of two classes.

The following figure depicts the sigmoid activation function:

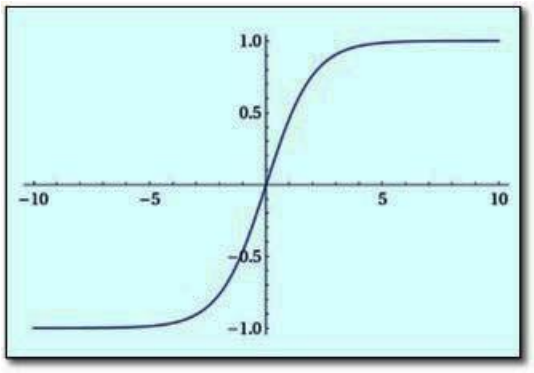


Figure 8.5: *Sigmoid activation function*

- **Tangent hyperbolic** (Tanh): It is also a nonlinear activation function having an S-shaped curve. It is basically a mathematically shifted version of the sigmoid (logistic) activation function with the additional value ranging from 0 to -1.0. This additional value is useful in the case when the input values to the network are negative.
  - **Equation**: F(x) = tanh(x) = $\frac{1-e^{-2x}}{1+e^{-2x}}$
  - **Range**: -1 to 1
  - **Uses**: It is mainly used in the hidden layer for classification between two classes.

The following figure depicts the Tanh activation function:

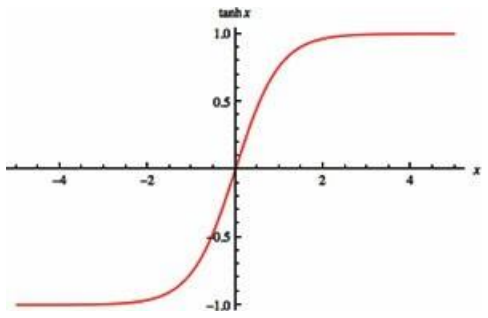


Figure 8.6: *Tanh activation function*

- **RELU (Rectified Linear Unit)**: RELU, also a nonlinear activation function, is the most widely used activation function because of having a less computationally expensive nature than the Tanh and the sigmoid activation functions.
  - **Equation**: F(x) = max(0, x), that is, it gives an output = x if x is positive an 0

otherwise.

  - **Range**: (0, infinity).
  - **Uses**: It is used in almost all **Convolutional Neural Networks** (**CNNs**).

The following figure depicts the RELU activation function:

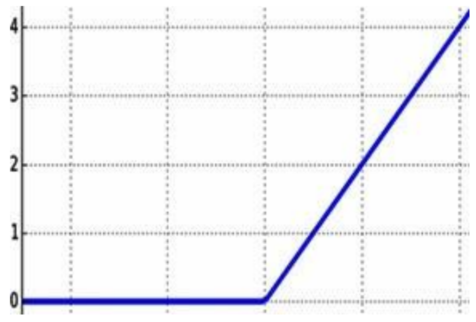


Figure 8.7: *RELU activation function*

- **Leaky ReLU activation function**: Leaky ReLU function is an improved version of the ReLU activation function, which addresses one of the biggest problems of the latter. As we know, in the ReLU activation function, the gradient is 0 for all the negative values of inputs(x), which further may lead to a dead ReLU problem, that is, the neurons get dead in that region.

  Rather than defining the function as 0, the leaky ReLU activation function defines the function as an extremely small linear component of input(x). It means this activation function returns output for negative values as well. Therefore, we would not encounter the problem of dead neurons in that region. The following is the formula for leaky ReLU function:

$$f(x) = \max(0.01*x, x)$$

The following table depicts the analogies between **Biological Neuron Model** (**BNN**) and **Artificial Neuron model** (**ANN**):

Biological Neural Network (BNN)	**Artificial Neural Network (ANN)**
Cell body (Soma)	Artificial neuron unit/node
Dendrites	Weighted input
Synapse	Weights
Axon	Output unit

Table 8.1: *BNN versus ANN*

The basic structure of ANN

ANNs are composed of layers of artificial neurons called units, organized in interconnected layers, where each artificial neuron can make simple decisions and feed those decisions to other neurons.

The following figure depicts the working of interconnected layers comprised in an ANN framework:

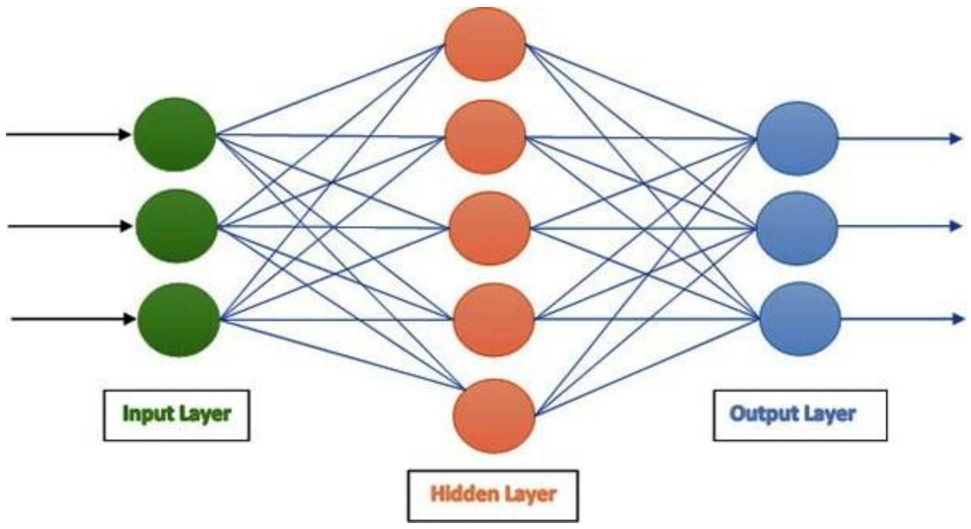


Figure 8.8: *Basic structure of ANN*

The explanation of three interconnected layers comprised in an ANN framework is as follows:

- **Input layer**: As the name implies, this layer contains those neurons/units that receive input in various forms of information from the outside world.
- **Hidden layer**: Hidden layer is an intermediate layer between input and output layers. It contains the neurons/units that process the inputs obtained by its previous layer. The activation function is also applied over this layer.
- **Output layer**: As the name implies, the neurons/units in the output layer collect and transmit the information exactly as it has been designed to give.

The one thing that should be kept in mind is the connection between units/neurons of adjacent layers having 'weights' associated with them.

Types of ANNs

The following table consists of various types of ANNs. The factors used are network topology, number of hidden layers, weights, and memory unit:

Factor	Types of ANN
Network topology	• **Feedforward Neural Networks (FNNs)**: As the name implies, they do not have any feedback loop. In these kinds of neural networks, the signal can flow only in one direction, that is, from input to output. • **Recurrent Neural Networks (RNNs)**: As the name implies, they consist of the feedback loop. In these kinds of neural networks, the signal can flow in both directions, that is, from input to output as well as output to input.
Number of hidden layers	• **Single-layer NNs**: Single-layer neural networks, the simplest form of the neural network, having only one layer of input units/nodes to send the weighted inputs

	to an adjacent layer of receiving units/nodes. The adjacent layer is called the output layer. There is no hidden layer in such networks. For example, single-layer Perceptron. • **Multi-layer NNs**: Unlike single-layer NNs, in multi-layer neural networks there is more than one layer of artificial neurons. It means they would have one or more hidden layers between input and output layers. For example, multi-layer Perceptron.
Weights	• **Fixed weight neural networks**: As the name implies, such neural networks have fixed weight. Once trained, these networks can adapt without any change of weights. • **Adaptive weight neural networks**: As the name implies, the weights in these networks are updated and changed during training.
Memory unit	• **Static neural networks**: Static neural networks are the networks in which the current output depends upon the current input only because they have memoryless units. For example, feedforward neural networks are static in nature. • **Dynamic neural networks**: Dynamic neural networks are the networks in which the current output depends upon the current input as well as current output because the units have memories. For example, recurrent neural networks are dynamic in nature.

Table 8.2: *Types of ANN*

Optimizers for training the neural network

Optimizers may be defined as the algorithms or methods to update various parameters, such as weights and learning rate, of your NN to minimize an error function (loss function).

This section will walk you through various types of optimizers:

Gradient descent

Gradient descent, the most basic but most used optimization method, is a first-order optimization algorithm that depends on the first-order derivative of a loss function. This approach reduces a loss function by moving in the direction opposite to that of the steepest ascent and achieve the minima. Gradient descent is also adopted in backpropagation in NN where the loss is transferred from one layer to another and the parameter i.e., weights are updated depending upon when the minimum loss is achieved. The following is the mathematical equation of gradient descent:

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

Where *W* represents the weight and α represents the learning rate.

Advantages of Gradient descent:

- Easy to understand.
- The computation is easy.

Disadvantages of Gradient descent:

- It may trap at local minima.
- As this method calculates the gradient for the whole datasetin one update, it may take months or years to converge to the minima if the dataset is too large.
- It requires large memory.

Stochastic Gradient Descent (SGD)

SGD is a variant of gradient descent. It updates the model's parameters one by one and more frequently. For example, if our dataset is having 10000 rows, the stochastic gradient descent algorithm will update the model's parameters 10000 times in one cycle of dataset rather than in one time in the gradient descent.

Advantages of Stochastic gradient descent:

- It converges in less time.
- It requires less memory.

Disadvantages of gradient descent:

- The frequent update of the model parameter may result in a noisy gradient.
- It becomes computationally expensive because of frequent updates.
- High variance.

Mini-Batch Gradient Descent

Mini-Batch gradient descent, a combination of the concepts of batch gradient descent and SGD, splits the training dataset into small batches and performs an update after every batch.

Advantages of Mini-Batch gradient descent:

- Less variance.
- It requires a medium amount of memory.
- It performs more efficient gradient calculations.

Disadvantages of Mini-Batch gradient descent:

- It does not guarantee good convergence.
- The smaller the learning rate the slower will be the convergence rate.

Stochastic Gradient Descent with Momentum

As the name entails, it is a stochastic optimization method that adds a term called momentum to regular SGD. Momentum accelerates the convergence towards the relevant direction and increases the stability to a certain extent. The following is the formula for SGD with momentum:

$$V_t = \beta V_{t-1} + (1 - \beta)\, \nabla_w L(W,X,y)$$
$$W = W - \alpha V_t$$

Here, L is the loss function.

β is another hyperparameter that takes the value from 0 to 1.

α is the learning rate.

∇_w is the gradient with respect to weight.

Advantages of SGD with Momentum:

- It reduces the noise and high variance of parameters.
- It converges faster than the gradient Descent method.

Disadvantages of SGD with Momentum:

- An extra hyperparameter is added that needs to be selected manually and accurately.

Adam (Adaptive Moment Estimation)

Adam optimizer, one of the most popular and famous gradient descent optimization methods, works with momentums of first and second order. Adam computes adaptive learning rates for each parameter to decrease the velocity so that we do not jump over the minimum. It stores the decaying average of the past gradients as well as the decaying average of the past squared gradients as follows:

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_t - \varepsilon}}} * V_{dw_t}$$
$$b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{dw_t - \varepsilon}}} * V_{db_t}$$

Advantages of Adam:

- It converges rapidly.
- It reduces the high variance of parameters.

Disadvantages of Adam:

- It is computationally costly.

Regularization

Neural networks are complex models, and it makes them prone to overfitting i.e., they perform well on the training dataset but not so good on the test dataset. In other words, we can say that neural network models have a high variance, and they cannot generalize well on the dataset they have not been trained on.

Getting more data and using regularization are the two ways to address overfitting in neural networks. The first way i.e., getting more data for training is quite impossible and expensive too. That's why regularization is the most common method to reduce overfitting.

Regularization techniques

The following are the given two commonly used regularization techniques applied in neural networks:

L1 and L2 regularization

L1 and L2, the most common types of regularization, update the general cost function by penalizing the complex models that is, adding another term known as the **Complexity term** or **Regularization term**.

Cost function = Loss + Regularization term

The benefit of adding the regularization term is that the values of weight matrices decrease. Now the question arises as to how it reduces overfitting with this? After adding the regularization term, the model assumes that an NN with smaller weight matrices leads to simpler models and hence reduces the overfitting to quite an extent.

The regularization term in both L1 and L2 differs as shown in the following equation:

- In L1:

  $$Cost\ function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

  Here, λ(lambda) is the regularization parameter whose value is optimized for better results.

  As seen, the absolute value of weights is penalized, hence the weights may be reduced to zero here.

- In L2:

  $$Cost\ function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

  L2 regularization forces the weights to decay towards zero but not exactly zero, hence it is very useful when we are not trying to compress our model. Otherwise, prefer L1 over L2.

Dropout

As the name implies, this regularization technique sets the probability of keeping a certain node in the NN to make it much smaller and simpler. We only decide the threshold but the probability of keeping each node is set at random. For example, there is a 25% probability of removing a node from the network if we set the threshold to 0.75.

To regularize an NN, it might seem a bad idea to randomly remove nodes from it. Yet, dropout regularization is a widely used method. But why does it work so well?

In dropout regularization, each node has a random probability of being removed, hence the NN cannot rely on any input node. That's the reason, the NN will be reluctant to provide high weights to some features because they might vanish. Consequently, the weights are spread across all the features and shrink the model to regularize it.

Installing useful Python package for ANN

For building ANNs, we will be using a powerful Python package for neural networks called NeuroLab. This package consists of basic neural networks algorithms along with flexible network configurations as well as learning algorithms for Python. The following command is used to install the NeuroLab:

```
pip install NeuroLab
```

In case you are using an anaconda environment, the following would be the command to install the NeuroLab:

```
conda install -c labfabulous neurolab
```

Examples of building some neural networks

Here, we will be creating some neural networks by using the Python NeuroLab package.

Perceptron-based classifier

Perceptron is the building block of neural networks. The following Python code is used to build a simple Perceptron-based classifier:

```
#Importing the required python packages:
import matplotlib.pyplot as plt
import neurolab as nl

#Providing the input values. We also need to provide the target value because it
is of supervised learning:
input = [[0.5, 0.3], [0, 1.5], [1.6, 0.8], [0.7, 2.1]]
target = [[0], [0], [0], [1]]

#Building the neural network with 2 inputs and 1 neuron:
net = nl.net.newp([[0, 1],[0, 1]], 1)

#Train the network using Delta rule:
error = net.train(input, target, epochs=200, show=15, lr=0.1)

#Visualizing the output and plotting the graph:
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.grid()
plt.show()
```

The output of the preceding Python script is given as follow:

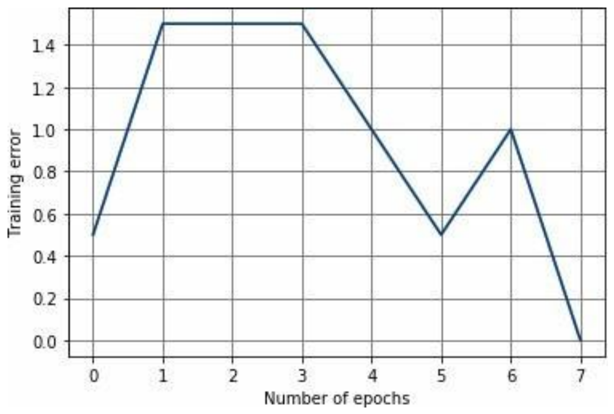


Figure 8.9: *Training error versus number of epochs (Perceptron-based classifier)*

Single-layer neural networks

With the help of the following written Python script, we will create a single-layer neural network. This single-layer NN consists of independent neurons acting on input data to produce the output. We are using the text file named **`NN_single.txt`** as our input whose first two columns are features and the last two columns are labels.

```
#Importing the required python packages:
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

#Loading the dataset from saved text file in local directory:
data_input = np.loadtxt("{Your Directory Path}/NN_single.txt")

#Separating the four columns of input data into 2 data columns and 2 labels
data = data_input[:, 0:2]
labels = data_input[:, 2:]

#Plotting the input data:
plt.figure()
plt.scatter(data[:,0], data[:,1])
plt.xlabel('Dimension_1')
plt.ylabel('Dimension_2')
plt.title('Input-Data')

#Defining the minimum as well as maximum values for each dimension
dimension1_min, dimension1_max = data[:,0].min(), data[:,0].max()
dimension2_min, dimension2_max = data[:,1].min(), data[:,1].max()

#Defining the number of neurons in the output layer
nn_output_layer = labels.shape[1]

#Defining a single-layer NN
dim1 = [dimension1_min, dimension1_max]
dim2 = [dimension2_min, dimension2_max]
neural_net = nl.net.newp([dim1, dim2], nn_output_layer)

#Training the NN with number of epochs and learning rate
error = neural_net.train(data, labels, epochs=100, show=10, lr=0.01)

#Visualizing and plotting the training progress
```

```
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Training error')
plt.title('Training error progress')
plt.grid()
plt.show()

#Using the test data-points to test the classifier
print('\nThe Test Results are:')
data_test = [[2.5, 4.5], [2.9, 3.8], [3.6, 4.7],[4.5, 7.8]]
for item in data_test:
  print(item, '-->', neural_net.sim([item])[0])
```

The output of the preceding Python script is given as follow:

```
Epoch: 10; Error: 8.0;
Epoch: 20; Error: 4.0;
Epoch: 30; Error: 4.0;
Epoch: 40; Error: 4.0;
Epoch: 50; Error: 4.0;
Epoch: 60; Error: 4.0;
Epoch: 70; Error: 4.0;
Epoch: 80; Error: 4.0;
Epoch: 90; Error: 4.0;
Epoch: 100; Error: 4.0;
The maximum number of train epochs is reached
```

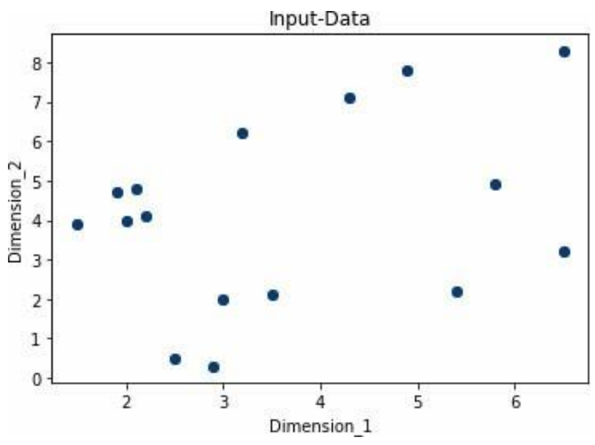


Figure 8.10: *Input data (single-layer neural network)*

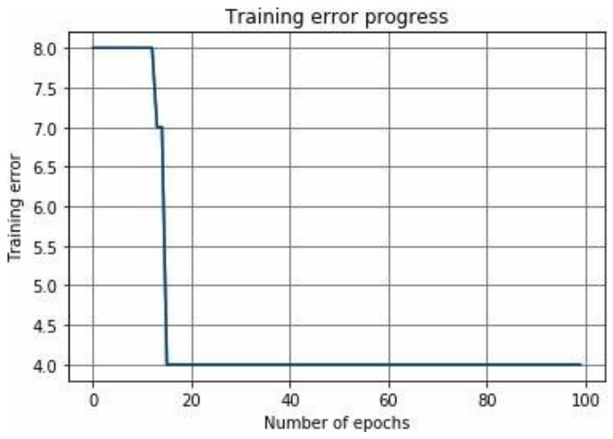


Figure 8.11: *Training error versus number of epochs (single-layer neural network)*

```
The Test Results are:
[2.5, 4.5] --> [1. 1.]
[2.9, 3.8] --> [1. 1.]
[3.6, 4.7] --> [1. 1.]
[4.5, 7.8] --> [1. 1.]
```

Multi-layer neural networks

With the help of the following written Python script, we will create a multi-layer neural network. This multi-layer NN, which works like a regressor, consists of more than one layer to extract the underlying patterns in the training data. We will generate the data points based on the equation:.

```
#Importing the required python packages:
import numpy as np
import matplotlib.pyplot as plt
import neurolab as nl

#Generating some data point based on the equation: y= 2X^2+8.
minimum_val = -35
maximum_val = 35
num_points = 160
x = np.linspace(minimum_val, maximum_val, num_points)
y = 2 * np.square(x) + 8
y /= np.linalg.norm(y)

#Reshape the input dataset
data = x.reshape(num_points, 1)
labels = y.reshape(num_points, 1)

#Visualizing and plotting the input dataset
plt.figure()
plt.scatter(data, labels)
plt.xlabel('Dimension_1')
plt.ylabel('Dimension_2')
plt.title('Data_points')

#Creating the NN having two hidden layers-10 neurons in the first hidden layer
and 6 in the second hidden layer. The output layer consists of 1 neuron.
NN = nl.net.newff([[min_val, max_val]], [10, 6, 1])

#Using the gradient-descend training algorithm to train the NN
```

```
NN.trainf = nl.train.train_gd

#Training the NN on previously generated data
error = NN.train(data, labels, epochs=100, show=10, goal=0.01)

#Running the NN on the training data-points
output = NN.sim(data)
y_pred = output.reshape(num_points)

#Plotting and visualizing
plt.figure()
plt.plot(error)
plt.xlabel('Number of epochs')
plt.ylabel('Error')
plt.title('Training error progress')

#Plotting visualizing the actual versus predicted output
x_dense = np.linspace(min_val, max_val, num_points * 2)
y_dense_pred=NN.sim(x_dense.reshape(x_dense.size,1)).reshape(x_dense.size)
plt.figure()
plt.plot(x_dense, y_dense_pred, '-', x, y, '.', x, y_pred, 'p')
plt.title('Actual versus predicted')
plt.show()
```

Output:

```
Epoch: 10; Error: 15.477455279786286;
Epoch: 20; Error: 13.528289284270963;
Epoch: 30; Error: 0.7008856414498756;
Epoch: 40; Error: 2.5499794643999127;
Epoch: 50; Error: 0.609645167038151;
Epoch: 60; Error: 1.4625867712043874;
Epoch: 70; Error: 0.12571106553919292;
Epoch: 80; Error: 0.22895324068235945;
Epoch: 90; Error: 0.5779960386023401;
Epoch: 100; Error: 0.4679757574525619;
The maximum number of train epochs is reached
```

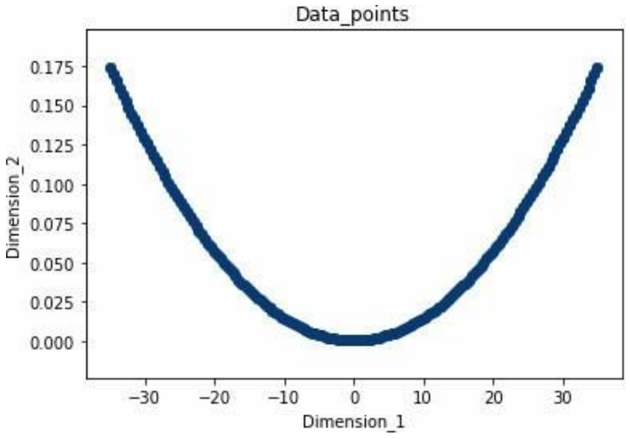


Figure 8.12: *Input data (multi-layer neural network)*

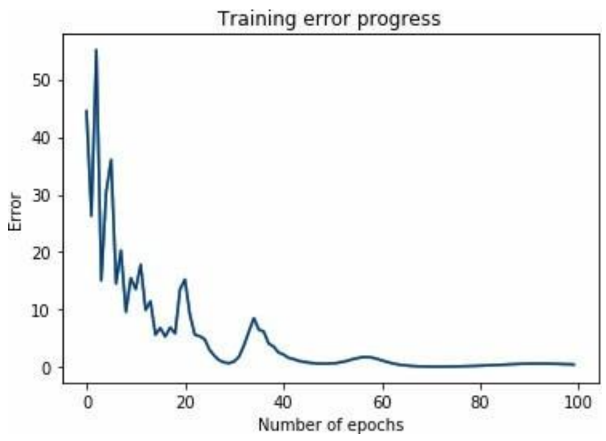


Figure 8.13: *Training error versus number of epochs (multi-layer neural network)*

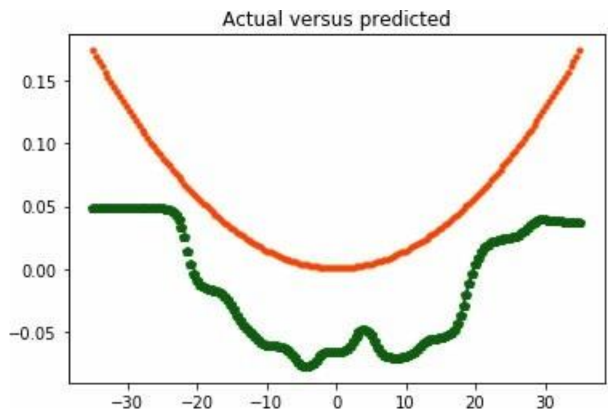


Figure 8.14: *Actual versus predicted output (multi-layer neural network)*

Vector quantization

Vector quantization, an N-dimensional version of rounding off, is commonly used in **Natural Language Processing** (**NLP**), **Computer Vision** (**CV**), and **Machine Learning** (**ML**). We can use neural networks to create a vector quantizer. The following Python script will be implemented using the NeuroLab library:

```
#Importing required Python packages
import numpy as np
import neurolab as nl
import matplotlib.pyplot as plt

#Creating the train samples
input_data = np.array([[-3, 0], [-2, 1], [-2, -1], [0, 2], [0, 1], [0, -1], [0,
-2], [2, 1], [2, -1], [3, 0]])
labels = np.array([[1, 0], [1, 0], [1, 0], [0, 1], [0, 1], [0, 1], [0, 1], [1,
0], [1, 0], [1, 0]])

# Creating NN with 2 layers (4 neurons in input layer and 2 neurons in output
layer)
NN = nl.net.newlvq(nl.tool.minmax(input_data), 4, [.5, .5])

# Training the neural network
error = NN.train(input_data, labels, epochs=500, goal=-1)
```

```
# Plotting and visualizing the result
xx, yy = np.meshgrid(np.arange(-3, 3.4, 0.2), np.arange(-3, 3.4, 0.2))
xx.shape = xx.size, 1
yy.shape = yy.size, 1
i = np.concatenate((xx, yy), axis=1)
o = net.sim(i)
grid1 = i[o[:, 0]>0]
grid2 = i[o[:, 1]>0]

class1 = input_data[target[:, 0]>0]
class2 = input_data[target[:, 1]>0]

plt.plot(class1[:,0], class1[:,1], 'cs', class2[:,0], class2[:,1], 'ko')
plt.plot(grid1[:,0], grid1[:,1], 'b.', grid2[:,0], grid2[:,1], 'gx')
plt.axis([-3.3, 3.3, -3, 3])
plt.xlabel('Input_data[:, 0]')
plt.ylabel('Input_data[:, 1]')
plt.legend(['class 1', 'class 2', 'detected class 1', 'detected class 2'])
plt.title('Vector quantization using neural networks')
plt.show()
```

Output:

```
Epoch: 100; Error: 0.0;
Epoch: 200; Error: 0.0;
Epoch: 300; Error: 0.0;
Epoch: 400; Error: 0.0;
Epoch: 500; Error: 0.0;
The maximum number of train epochs is reached
```

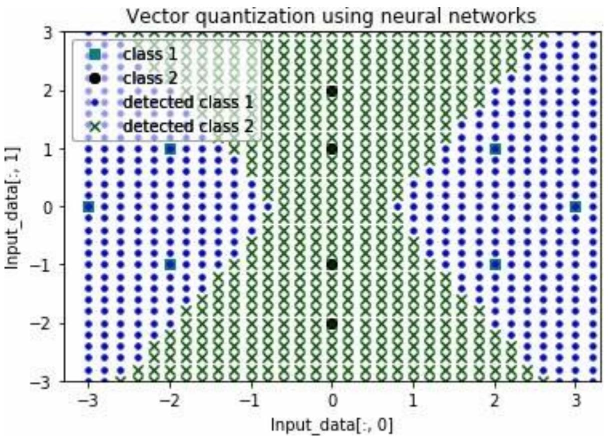


Figure 8.15: *Vector quantization using neural networks*

Conclusion

In this chapter, we learned about the basics of **Artificial Neural Networks** (**ANNs**) and build some neural networks in the Python programming language. We got to know that the ANNs, also termed as Artificial Neural Systems and Parallel Distributed Processing Systems, are those computational models that are inspired by the way the human brain processes information using the biological nervous system. We also discussed the working of the biological neuron so that we can understand how our brain works.

From the implementation perspective, we learned how we can build some neural networks, namely, Perceptron-based classifier, single-layer neural network, multi-layer neural network, and vector quantization in the Python programming language by using the NeuroLab library.

In the next chapter, you will learn about the basic concepts of reinforcement learning along with its implementation in the Python programming language.

Questions

1. How does the biological neuron work?
2. What is Artificial Neural Network (ANN) and how does it work?
3. What are the various types of ANN? Explain.
4. What is an activation function? Explain various types of activation functions.
5. How to build a Perceptron-based classifier in the Python programming language?
6. How can you create a single-layer neural network using the Python programming language?
7. How can you create a multi-layer neural network using the Python programming language?
8. How to build a vector quantizer using the Python programming language?

CHAPTER 9

Implementing Reinforcement Learning with Python

Introduction

Machine Learning (**ML**) is a large field of study that focuses on learning, that is, acquiring skills or knowledge from experience. As a practitioner in the field of ML, you may encounter different types of learning. Among them, one is reinforcement learning, the coolest branch of **Artificial Intelligence** (**AI**), which has already proven its prowess by beating the world champions in games of Chess, Go, and even DotA 2.

We humans went through the learning enforcement when we were children. In childhood, when we started crawling and tried to get up, we fell over and over, but our parents or elders were there to lift us and teach us. **Reinforcement Learning** (**RL**) is a teaching based on experience, that is, the machine must deal with what went wrong before and look for the correct outlook. In simple words, reinforcement learning is the concept where machines can teach themselves based on the results of their own actions.

In this chapter, we will learn about reinforcement learning and its building blocks, namely, environment and agent. With the help of some examples, we will understand how to construct an environment and an agent using the Python programming language.

Structure

This chapter is structured as follows:

- Understanding reinforcement learning
- Markov Decision Process (MDP)
- Building blocks of reinforcement learning
    - Environment
    - Agent
- Constructing an environment using Python
- Constructing an agent using Python

Objective

After studying this chapter, the reader will be able to construct a reinforcement learning environment and agent in the Python programming language. The reader will also be able to install and use OpenAI Gym, an open-source Python library, to develop, compare, and construct reinforcement learning algorithms.

Understanding reinforcement learning

Reinforcement learning methods are a bit different from supervised, unsupervised, and semi-supervised learning methods. In these kinds of learning algorithms, a trained agent interacts with a specific environment. The job of the agent is to interact with the environment and once observed, it takes actions regarding the current state of that environment.

Workflow of reinforcement learning

The workflow of reinforcement learning is as follows (refer *figure 9.1*):

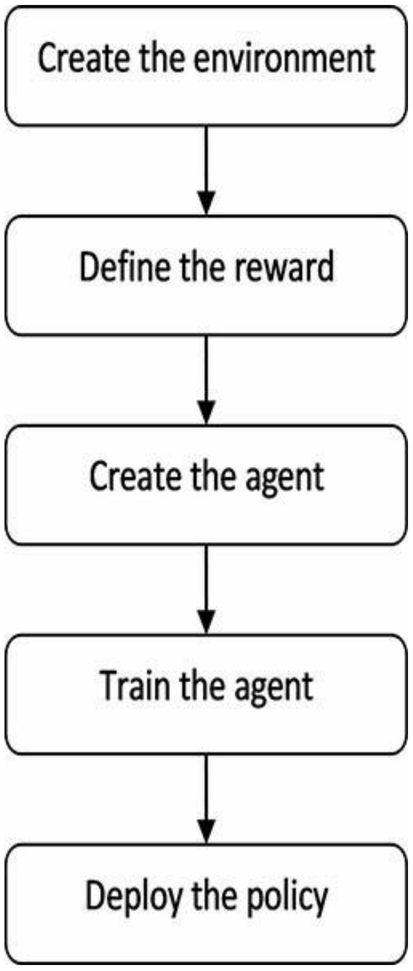


Figure 9.1: *Reinforcement learning workflow*

Let's understand the working of reinforcement learning methods in the following steps:

1. Prepare an agent with some set of strategies.
2. Observe the environment's current state.
3. Regarding the current state of the environment, select the optimal policy and perform

suitable action accordingly.

4. Agent gets reward or penalty based on the action it took according to the current state of the environment.
5. If needed update the set of strategies.
6. Repeat the process until the agent learns and adopts the optimal policy.

Markov Decision Process (MDP)

Let's understand the basic elements of a reinforcement learning problem:

- **Agent**: The learner in reinforcement learning is called an agent. It is the sole decision-maker.
- **Environment**: It represents a physical world where an agent learns. It also decides what actions are to be performed by an agent.
- **Action**: It consists of a list of actions that can be performed by an agent.
- **State**: It represents the current situation of the agent.
- **Reward**: It is the feedback given by the environment for every selected action by the agent. It is a scalar value.
- **Policy**: As the name implies, it is the strategy that the agent prepares to map situations to actions.
- **Value function**: It is the value of the state, that is, the current situation of the agent. It shows up the rewards achieved by the agent from starting of the state until the policy/strategy is executed.

The following figure depicts the terminologies used in reinforcement learning:

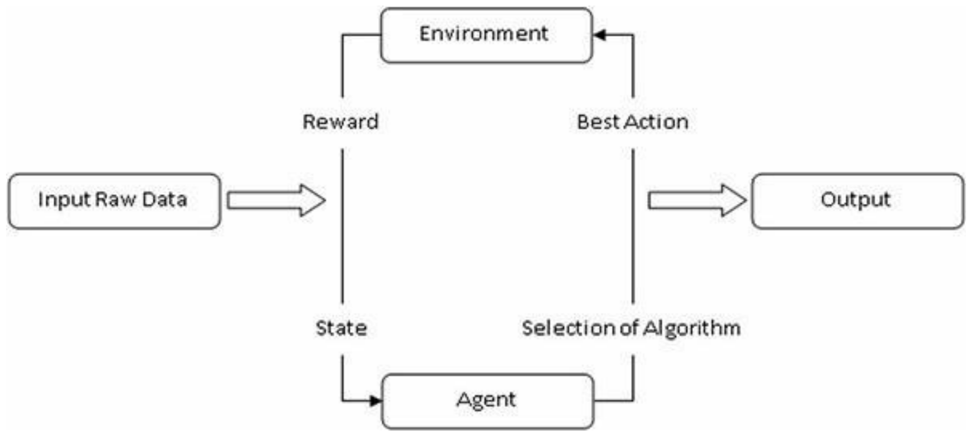


Figure 9.2: *Terminologies used in reinforcement learning*

Markov Decision Process (**MDP**) may be defined as the mathematical framework to describe an environment in RL. The following diagram depicts the interaction between agent and environment in MDP:

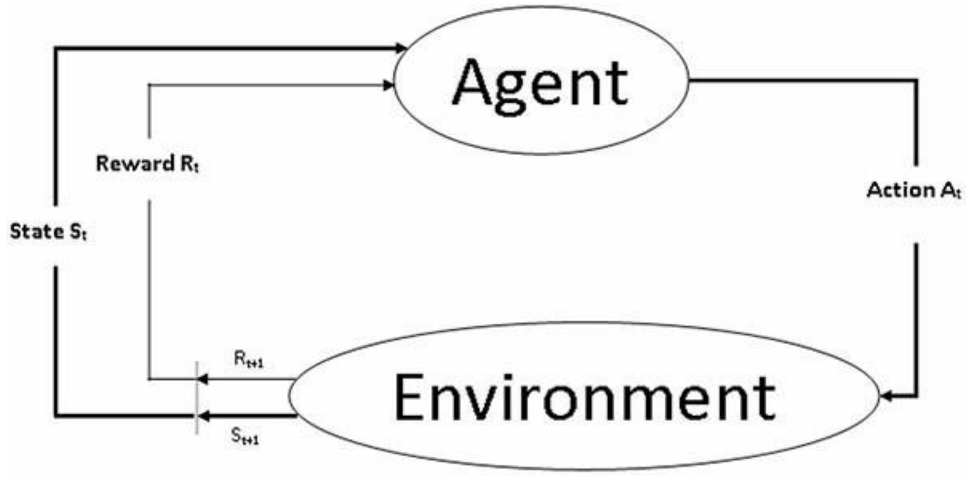


Figure 9.3: *Basic diagram of reinforcement learning*

As shown previously, an MDP model contains the following:

- A set of possible world states denoted by *S*.
- A set of possible actions denoted by *A*.
- A set of models.
- A reward function denoted by *R(s, a)*.

Working of Markov Decision Process (MDP)

First, at each discrete time step $t = 0, 1, 2, 3, 4,\ldots$, the agent and the environment interact with each other. Next, at each time step, the agent gets the information about the environment state S_t and based on that state it will choose an action A_t. Based on the actions, the agent will also get a numerical reward signal say R_{t+1}. In this way, we will get the sequence like S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, and so on.

The random variables R_t and S_t have a well-defined discrete probability distributions, which by the virtue of Markov property, are dependent only on the preceding state. Assume *S*, *A*, and *R* as the sets of states, actions, and rewards, then the following is given the probability that the values of S_t, R_t, and A_t are taking values of *s'*, *r*, and *a* along with the previous state:

$$p(S',r|s,a) = P\{S_t=s',R_t=r|S_{t-1} = s, A_{t-1}= a\}$$

Here *p* is the function that controls the dynamics of the process.

Difference between reinforcement learning and supervised learning

The following table shows the main differences between reinforcement learning and supervised learning:

Criteria	Supervised learning	Reinforcement learning

Definition	The model is trained with labeled data.	An agent interacts with its environment and once observed, it takes actions regarding the current state of that environment. The agent learns from its experience.
Types of problems	There are two types of supervised learning: regression and classification.	The problems in reinforcement learning are reward-based.
Type of data	The data used by supervised learning is labeled data.	There is no predefined data in reinforcement learning.
Training	External supervision, that is, known outputs are there for training purposes.	There is no supervision.
Approach	It maps the labeled inputs to the known outputs.	It follows the trial-and-error method and has three implementation approaches, namely, value-based, policy-based, and model-based.
Example	Object detection	Chess game

Table 9.1: *Reinforcement Learning versus Supervised Learning*

Implementing reinforcement learning algorithms

The following are the three approaches to implement reinforcement learning algorithms:

- **Value-based approach**: This approach is all about finding the optimal value function, i.e., the maximum value at a state under any policy. That's the reason, in this approach, the agent expects the long-term return at any state(s) under policy *X*.
- **Policy-based approach**: This approach is all about finding the optimal policy for the maximum future rewards. Opposite to the value-based approach, it does not use the value function. There are mainly two types of policies in the policy-based approach:
  - **Deterministic**: As the name implies, in deterministic policy, the same action is produced by policy *X* at any state.
  - **Stochastic**: As the name implies, in stochastic policy, the action is determined by probability.
- **Model-based approach**: In the model-based approach, to make the agent learn, a virtual model is created for the environment. As the model representation is different for each environment, there will be no specific solution or algorithm for a model-based approach.

Reinforcement learning algorithms

Reinforcement learning algorithms are mainly used in AI and gaming applications. Some of the mainly used reinforcement learning algorithms are given as follows:

- **Q-learning**: It is a value-based off-policy reinforcement learning algorithm for supplying information to intimate which action an agent should perform. Q-learning algorithm learns the value function *Q(S, a)*. This value function means how good to act *a* at a specific state *S*.

  The working of the Q-learning algorithm is explained in the following flowchart in *figure*

9.4:

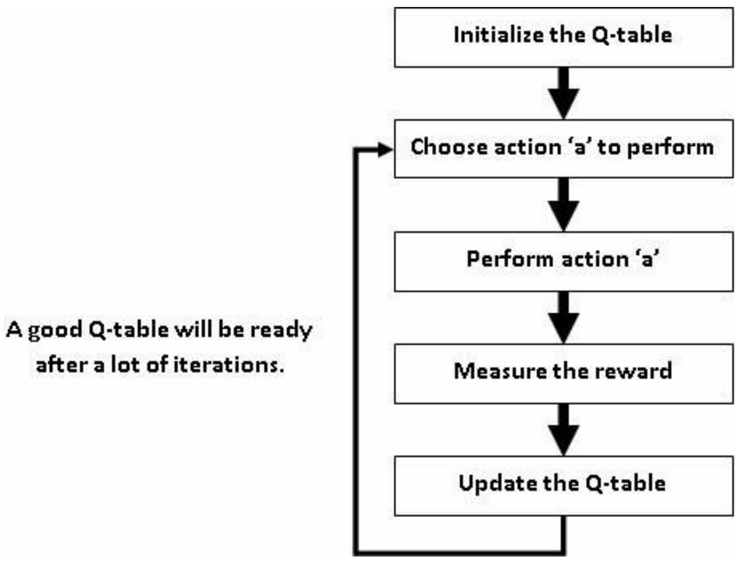

Initialize the Q-table
Choose action 'a' to perform
Perform action 'a'
Measure the reward
Update the Q-table
A good Q-table will be ready after a lot of iterations.

Figure 9.4: *Working of Q-learning*

- **State Action Reward State Action (SARSA)**: It is an on-policy reinforcement learning algorithm that selects the action for each state while learning using a specific policy. In SARSA, unlike Q-learning, the maximum reward for the next state is not required for updating the Q-value in the table. In other words, new actions and rewards are selected using the same policy that determined the original action.

  It is named SARSA because it uses *Q(s, a,r, s', a')* quintuple as described here:

  - **s**: Original state.
  - **a**: Original action.
  - **r**: Reward that is observed while following the state(s).
  - **s'**: New state.
  - **a'**: New action.

- **Deep Q Neural Network (DQN)**: DQN, as the name implies, is a Q-learning algorithm using neural networks. The use of DQN is for a big state environment where updating a Q-table is a challenging and complex task. In such cases, instead of updating the Q-table, the neural network approximates the Q-value for every action and state.

Types of reinforcement learning

The following are the two types of reinforcement learning:

- **Positive reinforcement**: It may be defined as an event that occurs due to a specific behavior and has a positive effect on that behavior. In other words, it increases the strength and the frequency of that behavior.

  The advantage of positive reinforcement is that it maximizes the performance as well as sustains the change for a long period of time. On the other hand, the disadvantage of positive reinforcement is that it can lead to the overload of states, which can diminish the results.

- **Negative reinforcement**: It is opposite to positive reinforcement learning as it strengthens the specific behavior by avoiding the negative condition.

  The advantage of negative reinforcement is that it increases the behavior as well as provides a decent to a minimum standard of performance. On the other hand, the disadvantage of negative reinforcement is that it only provides enough to meet up a minimum behavior.

Benefits of reinforcement learning

Reinforcement learning, which applies to complex problems that cannot be tackled with other ML algorithms, is closer to **Artificial General Intelligence** (**AGI**). RL explores various possibilities autonomously as well as seeks long-term goals. Various benefits of RL include:

- **Focuses on the whole problem rather than dividing it into subproblems**: Reinforcement learning, rather than dividing the problem into subproblems, focuses on the whole problem to maximize the long-term reward. RL understands the final goal and is capable of trading off short-term rewards for long-term rewards.
- **No need to perform the data collection step**: In reinforcement learning, instead of a separate collection of data for the algorithms, training data is obtained via the direct interaction of the agent with the environment. In other words, training data is the experience of a learning agent. Therefore, in RL, there is no need to perform the data collection step.
- **Innovative**: Reinforcement learning algorithms are innovative because they can come up with completely new solutions that were never even considered by human beings.
- **Resistant to bias**: As we know that supervised learning algorithms will pick up the bias, if there is any in the way data is labeled. But on the other hand, reinforcement learning is resistant to bias and gives us solutions that are free from bias or discrimination.
- **Adaptable**: RL is adaptable, that is, it adapts to new environments automatically hence does not require retraining and redeployment to accomplish.

Challenges with reinforcement learning

Despite being successful in solving complex problems in diverse simulated environments, the adoption of reinforcement learning in the real world is slow. The following are the various challenges that have made the uptake of RL difficult:

- **RL agent's experience**: As discussed, training data is obtained via the direct interaction of the agent with the environment, that is, training data is the experience of the learning agent. Therefore, the rate of data collection is limited by the dynamics of the environment. For example, the environment having high latency will slow down the learning curve. On the other hand, the environment having high-dimensional state spaces will need extensive exploration for a good solution.
- **Delay in rewards**: The foundation principle of reinforcement learning is that the agent

can trade off short-term rewards for long-term gains. This principle makes RL useful. But on the other hand, this principle makes it difficult for the agent to find the optimal policy. It is true in the case of those environments where the outcome is unknown until many sequential actions are taken. For example, in the game of chess, the outcome is unknown until both the players have finished all their moves.

- **Inadequacy of interpretability**: The RL agent, deployed in the environment, takes actions based on its experience. Due to the lack of interpretability, the external observer sometimes cannot understand the reasons for those actions taken by the agent. In such a scenario, it is difficult to develop trust between the agent and the external observer.

Building blocks of reinforcement learning

There are two building blocks of reinforcement learning: agent and environment. Let's understand them in detail, as follows:

Agent

The learner in reinforcement learning is called an agent. It uses sensors and effectors for learning. Through sensors, it can perceive the environment, whereas through effectors it can act upon that environment. Following are some of the agents:

- **Human agent**: As the name implies, here the learners are human beings. The eyes, ears, nose, skin, and tongue play the role of sensors to perceive its environment. Other organs, such as hands, mouth, and legs play the role of effectors for acting upon the environment.
- **Robotic agent**: As the name implies, here the learners are robots. Camera and IR range finders play the role of sensors to perceive its environment. Other instruments such as motors and actuators play the role of effectors for acting upon the environment.
- **Software agent**: As the name implies, here the learners are software programs. For sensors and effectors, it has encoded bit strings.

Agent terminology

The following are some of the agent terminologies:

- **Performance measure**: The performance measure of an agent is the criteria that determines how successful that agent is.
- **Behavior**: The behavior of an agent is the action performed by any agent after any given sequence of percepts.
- **Percept**: Percept is the perceptual inputs of agents.
- **Percept sequence**: It represents the history of all perceptual inputs that have been received by an agent.
- **Agent function**: It may be defined as the map from the percept sequence to an action taken by the agent.

Environment

It represents a physical world where an agent learns. It also decides what actions to be performed by an agent. In simple words, the environment is the whole world for the agent where it lives as well as interacts. The agent interacts with its environment by means of some action, but is confined to the rules of that environment and cannot influence those rules by its actions.

Environment action space

Action space, as the name implies, is a set of actions that are allowed to be an agent in each environment. There are two types of environment action space as follows:

- **Discrete action space**: In discrete action space, as the name implies, all the actions are discrete in nature. For example, the *Atari Wall Breaker* game has a discrete action space of [Left, Right], whereas the *Pac-Man* game has a discrete action space of [Left, Right, Up, Down].
- **Continuous action space**: In continuous action space, as the name implies, all the actions are continuous in nature. For example, the environment of the *self-driving car* has a continuous action space of [steering wheel rotation, velocity].

Types of environments

Different types of environments can be categorized as follows:

- **Deterministic versus stochastic environment**: In deterministic environments, the next state of that environment can always be determined by the current state and the actions of an agent. For example, while driving a car if the agent performs an action of steering right, the car will move right only.

  On the other hand, in a stochastic environment, the next state of that environment cannot always be determined by the current state and the actions of the agent. For example, in case the agent's world of driving a car is not perfect and the agent tries to accelerate the car, then there is a small probability that the car may just stop.
- **Episodic versus sequential environment**: In episodic environments, the actions of agents do not depend upon any previous action. The actions are limited to the specific episode only. For example, in the game of archery, the action of the agent is independent of its previous attempts.

  On the other hand, in sequential environments, the actions of agents depend upon the previous actions. For example, in the game of chess, all the future actions will be dependent on its previous actions, that is, the history of sequences.
- **Fully observable versus partially observable environment**: In fully observable environments, as the name implies, the agent is always aware of the complete state of the environment at any point in time. For example, the game of chess is fully observable because at any given time the agent can always see the position of itself as well as its opponent.

  On the other hand, in partially observable environments, as the name implies, the agent cannot be aware of the complete state of the environment at any point in time. For example, the game of poker is partially observable because at any given time the agent cannot see the hands of its opponent.

- **Single-agent versus multi-agent environment**: In single-agent environments, as the name implies, there is only one agent that interacts with that environment. For example, only one person driving a car from one point to another.

  On the other hand, in multi-agent environments, as the name implies, there is more than one agent that interacts with that environment. For example, multiple cars are controlled by different agents.
- **Discrete versus continuous environment**: In discrete environments, there is a limited number of distinct and clearly defined states. For example, the game of chess.

  On the other hand, in continuous environments, the action space of the environment is continuous in nature. For example, self-driving cars.

Constructing an environment using Python

For constructing reinforcement learning environments, we will be using an open-source library called OpenAI Gym. We can use the following command to install it:

```
pip install gym
```

Once installed we can access various environments, such as `Cartpole-v0`, `Hopper-v1`, `MsPacman-v0`, and so on. You can learn more about OpenAI Gym at **https://gym.openai.com/docs/#environments**.

`Env interface`

The `Env` interface is used to create an environment. It provides `make()` method, which can be used to create an environment, as shown in the following Python script:

```
import gym
#creating cartpole-v0 environment
env = gym.make('CartPole-v0')

#intializing the environment
env.reset()
for _ in range(1000):
  env.render()#render the environment for visual representation
  env.step(env.action_space.sample())
env.close()#closing the environment for necessary cleanup
```

After running the preceding Python script, we will get a window showing a cartpole moving to the right. The initial position of the cartpole is depicted in the following figure:

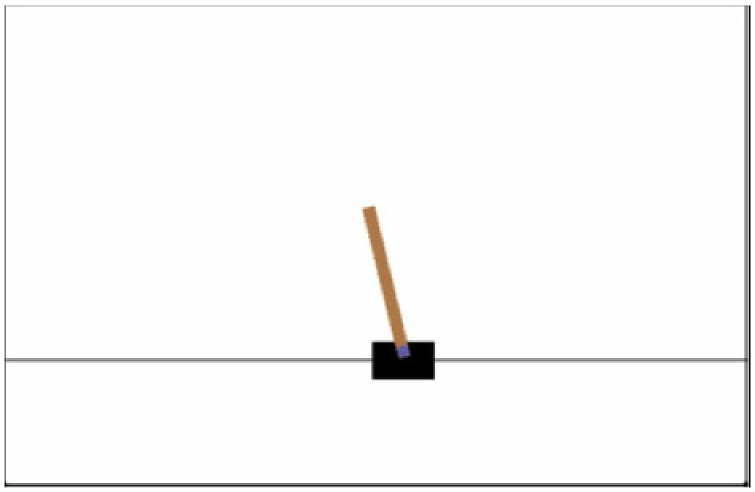


Figure 9.5: *CartPole environment-screenshot1*

Next, we will see the cartpole moving. This is depicted in the following figure:

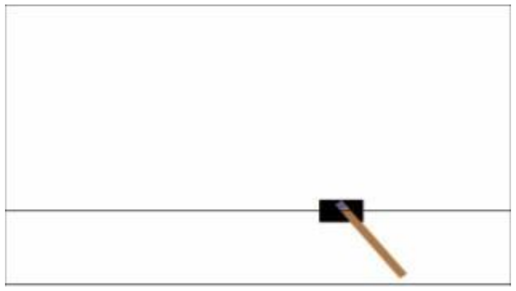


Figure 9.6: *CartPole environment-screenshot2*

In the end, we will see the cartpole going out of the window. This is depicted in the following figure:

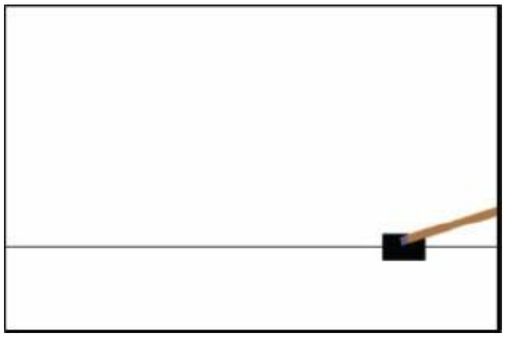


Figure 9.7: *CartPole environment-screenshot3*

Constructing an agent using Python

For constructing reinforcement learning agents, we will use the open-source library called OpenAI Gym as we did in the previous section:

```
import gym
```

```
#Creating cartpole-v0 environment
env = gym.make('CartPole-v0')
for _ in range(20):
  obs = env.reset()
  for i in range(100):
    env.render()
    print(obs)
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)
    if done:
      print("Episode finished after {} timesteps".format(i+1))
    break
```

When we run the preceding Python script, we will observe that the cartpole balances itself. The following figure depicts it:

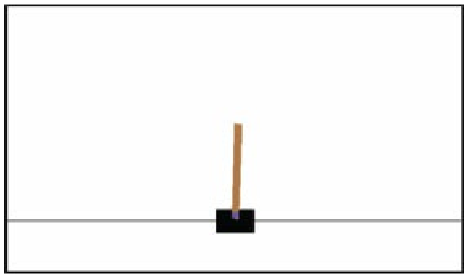


Figure 9.8: *Creating an agent-screenshot1*

If we let the script run for few seconds, we will see that the cartpole is still standing in balance. The following figure depicts it:

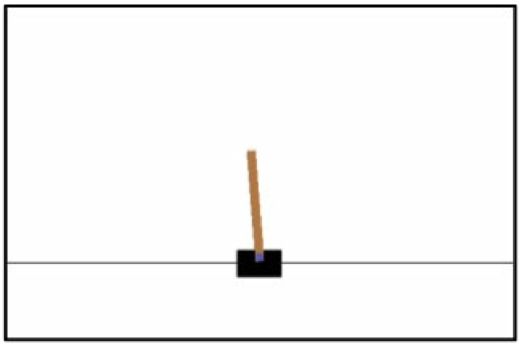


Figure 9.9: *Creating an agent-screenshot2*

Conclusion

In this chapter, we learned about the basics of reinforcement learning and construct its two building blocks, agent and environment, in the Python programming language. We got to know that in reinforcement learning methods, which are a bit different from supervised, unsupervised, and semi-supervised learning methods, a trained agent interacts with a specific environment. The job of the agent is to interact with the environment and once observed, it takes actions regarding the current state of that environment.

From the implementation perspective, we learned how to build the **`CartPole-v0`** environment and a learning agent for this environment that balance the cartpole in the Python programming language by using the OpenAI Gym package. In the next chapter, you will learn about the basic concept of **Convolutional Neural Network** (**CNN**) along with its implementation in the Python programming language.

Questions

1. What is reinforcement learning? How does it work?
2. What are the various reinforcement learning algorithms? Explain them in detail.
3. How reinforcement learning is different from supervised learning?
4. What is the role of the environment in reinforcement learning? What are the various types of environments?
5. What is the role of an agent in reinforcement learning? Explain in detail.
6. How to build a reinforcement learning environment in the Python programming language?
7. How can you create a reinforcement learning agent in the Python programming language?

CHAPTER 10

Implementing Deep Learning and Convolutional Neural Network

Introduction

Over the past decade, without any doubt, we have noticed a quantum jump in the quality of a wide range of everyday technologies. Let's have a look at the applications around us like self-driving cars, speech-enabled personal assistants, Gmail's smart reply, image recognition, chatbots, humanoid robots. All of these applications get the power from **Deep Learning** (**DL**). Apart from these applications, DL also powers some of the other interesting applications in the world like Amazon Go, Kitty Hawk's Cora, and Google Tulip. The ability of deep learning to handle a large amount of data makes it possible.

In this chapter, we will learn about deep learning and also how it is different from machine learning. We will also learn about **Convolutional Neural Network** (**CNN**), which is one of the most popular deep neural networks. We will understand how to construct an image classifier using CNN in the Python programming language.

Structure

This chapter is structured as follows:

- Understanding deep learning
    - Machine learning versus deep learning
- Elucidation of convolutional neural networks
- Architecture of convolutional neural networks
- Localization and object detection with deep learning
- Image classification using CNN in Python.

Objective

After studying this chapter, the reader will be able to construct an image classifier using convolutional neural networks in the Python programming language. The reader will also be able to install and use Keras – a deep learning API written in python for developing and evaluating deep learning models.

Understanding Deep Learning

Deep learning sets to be the most transformative technology existing over the next decade. It is a subset of **Machine Learning** (**ML**), which on the other hand is a subset of **Artificial Intelligence** (**AI**). These three are a set of tracking dolls nested within each other as shown in the following figure:

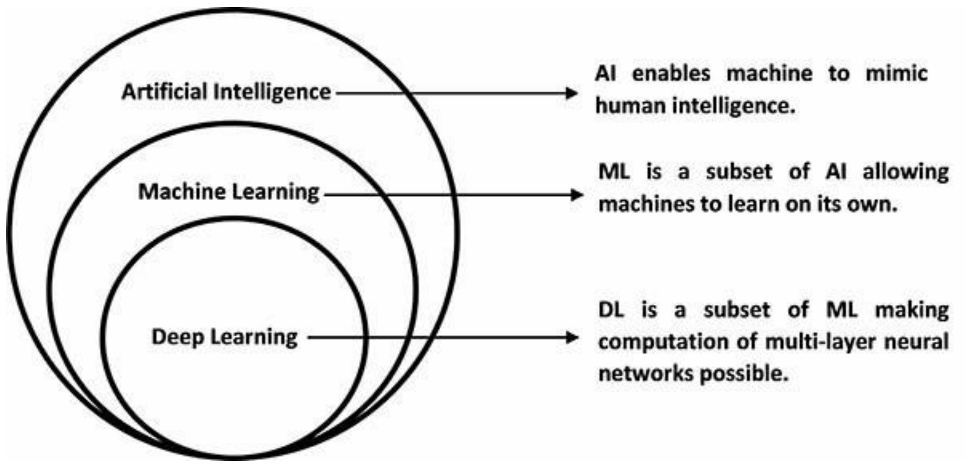

Artificial Intelligence
Machine Learning
Deep Learning
AI enables machine to mimic human intelligence.
ML is a subset of AI allowing machines to learn on its own.
DL is a subset of ML making computation of multi-layer neural networks possible.

Figure 10.1: *AI, ML, and DL*

AI is a technique that enables machines to mimic human intelligence. Whereas ML represents a set of algorithms that learn from data and make all this possible. On the other hand, DL is a type of ML inspired by the structure and function of the human brain. Deep learning uses a multi-layered structure of algorithms called **Artificial Neural Network** (**ANN**) to draw similar conclusions as human beings.

We have already discussed ANN and its implementation in the Python programming language in *chapter 8*. Just to recall, the basic structure of ANN is shown here:

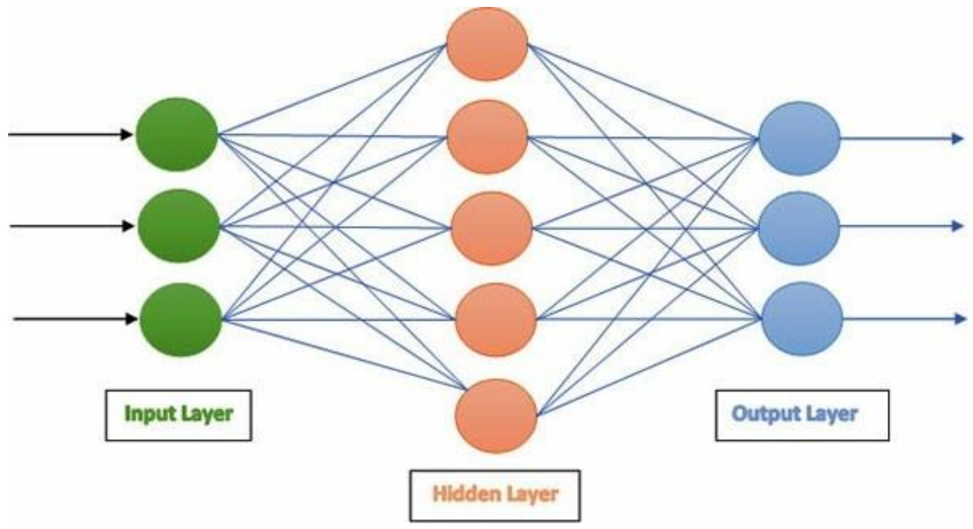


Figure 10.2: *Basic structure of ANN*

Machine learning versus deep learning

Although there are many differences between machine learning and deep learning, the following table gives five most important differences between these two subsets of AI:

Criteria	Machine learning	Deep learning
Human intervention	To get results, it requires more ongoing human intervention.	More complex to set up but requires less human intervention thereafter.
Machine dependency	ML algorithms can work on low-end machines.	DL algorithms require powerful hardware and resources to work perfectly in case of complex tasks.
Time of execution	ML algorithms have fewer parameters than DL algorithms, hence they can operate quickly. But ML algorithms may be limited in the power of their results.	DL algorithms take less time for testing and hence can generate results instantaneously.
Feature extraction	A machine learning expert is required to identify most of the features extracted by ML algorithms.	DL algorithms can extract high-level features and can self-learn from the same as well.
Approach	It tends to require structured data. It uses traditional ML algorithms such as logistic regression, linear regression, **K-Nearest Neighbors** (KNN), and so on.	DL algorithms are built to accommodate large volumes of unstructured data. For implementation, it employs neural networks.

Table 10.1: *Machine Learning versus Deep Learning*

Elucidation of Convolutional Neural Networks

Convolutional Neural Networks (**CNN**) and ordinary neural networks are similar to each other in the manner that they both are made up of neurons having weights and biases. If we talk about the working of ordinary neural networks, each neuron receives one or more inputs, does the sum of the weights, and passes that weighted sum through an activation function to produce the final output. Here the question arises that if CNNs and ordinary NNs are so much similar then what makes them different?

The difference between CNNs and ordinary NNs is in the types of layers along with how they both treat the input data. As we know that ordinary NNs ignore the structure of the input data and convert all the input data into one-dimensional data before feeding it into the network for final processing. Whereas, while processing, CNN architecture considers the two-dimensional structure of images. It can also take any other two-dimensional input such as speech signals. The CNN architecture is designed in such a way that it extracts the properties specific to images.

In contrast to standard multilayer neural networks, CNNs have one or more convolutional layers and pooling layers followed by one or more fully connected layers. That's the reason we can think of CNN as a special case of fully connected networks. Isn't it interesting?

The Architecture of Convolutional Neural Network

CNN is a list of layers that transforms the 3-dimensional (image having a width, height, and depth) image volume into a 3-dimensional output volume. The working seems similar as we overlay an filter on the input image because each neuron in the current layer of CNN is connected to a small patch of the output from the previous layer. It uses M filters, that is, feature extractors to get all the details like edges, corners, and so on. Before deep diving into the architecture of CNN, let's understand the layers [INPUT-CONV-RELU-POOL-FC] used to construct CNNs, which is given as follows:

- **INPUT layer**: The input layer holds the raw pixel values of an input image. By raw pixel values, we mean the data of the image as it is. For example, INPUT [6×6×3] means the array of matrix of RGB; here 3 refers to RGB values. Whereas INPUT [4×4×1] means the array of matrix of grayscale image; here 1 refers to grayscale values.
- **Convolution Layer (CONV)**: It is the first layer and most of the computation i.e., convolutions between neurons and various patches in the input is done in this layer. We can consider it as a mathematical operation that takes an image matrix and a filter (kernel) as two inputs.
    - An image matrix (h × w × d)
    - A filter or kernel of dimension (k_h × k_w × d)
    - The output will be a volume of dimension (h-k_h +1) × (w- k_w +1) × 1

  For example, we can get the convolved feature map from 5 × 5 image matrix and 3 × 3 filter matrices by multiplying them as follows:

1	**1**	**1**	**0**	**0**
0	**1**	**1**	**1**	**0**
0	**0**	**1**	**1**	**1**
0	**0**	**1**	**1**	**0**
0	**1**	**1**	**0**	**0**

1	**0**	**1**
0	**1**	**0**
1	**0**	**1**

Table 10.2: *Image matrix: 5×5 and filter or kernel matrix: 3×3*

  The convolved feature map (3 × 3 output matrix) will be as follows:

4	**3**	**4**
2	**4**	**3**
2	**3**	**4**

Table 10.3: *Feature map*

  By applying various kinds of filters (kernels) in images, we can perform operations such as identity, Edge detection, Sharpen, Box Blur, Gaussian Blur, and so on.

  The following are the two configuration hyperparameters, namely, Stride and Padding, used in the Convolution layer:

    - **Stride**: The filter is moved across the input image from top to bottom and left to right. While doing a horizontal movement, it observes one-pixel column change. On the other hand, while doing a vertical movement, it observes the one-pixel row change. The amount of movement is called stride. The stride is almost always symmetrical in width and height dimensions. The default stride in 2-D is (1, 1), which can also be changed to (2, 2). The change of stride will change the size of the resulting feature map.

- **Padding**: It may be defined as the addition of the pixels to the edge of the input image. There are two options for padding:
    - **Zero padding**: As the name implies, zero padding is the technique to pad the input image with zeros.
    - **Valid padding**: As the name implies, valid padding is the technique to keep only the valid part of the input image. It will drop that part of the image where the filter (kernel) does not fit.

- **Non-Linearity (ReLU)**: It is called a rectified linear unit layer. Its function is to introduce non-linearity in the convolutional network. It basically applies an activation function to the output of the previous layer. The output of ReLU function is f(x) = max(0, x). We can also use two other non-linear functions, namely, tanh or sigmoid but the performance of ReLU is much better than these two.
- **Pooling Layer (POOL)**: The pooling layer, another building block of CNN, reduces the number of parameters in a case when images are too large. Its main task is down-sampling or subsampling by reducing the dimensionality of each map but retains important information. It is called spatial pooling because it operates independently on every slice of the input and resizes it spatially. Spatial pooling can be of the following three types:
    - **Max pooling**: It involves taking the largest element from the rectified feature map.
    - **Average pooling**: It involves taking the average for each part of the feature map.
    - **Sum pooling**: It involves taking the sum of all elements in the feature map.
- **Fully Connected Layer (FC Layer)**: We flatten our matrix into a vector and feed it into the fully connected layer (FC layer) or more specifically called the output layer. The output class score will also be computed in this layer.

The typical architecture of CNN is shown here:

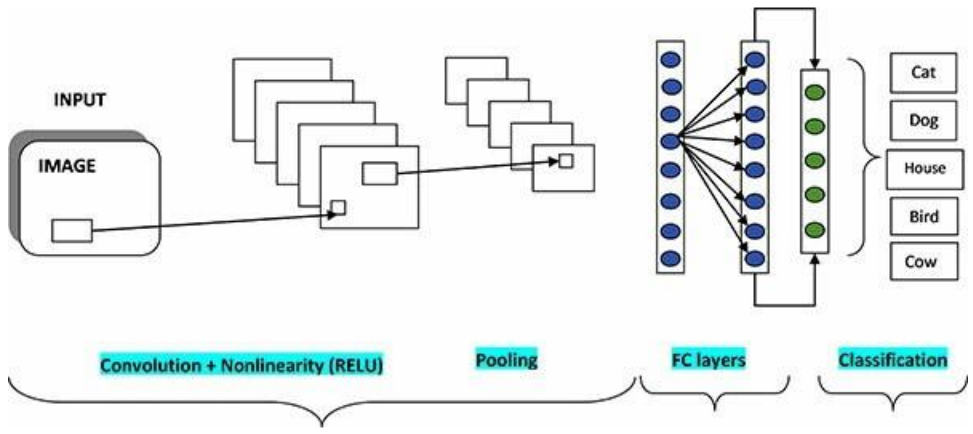


Figure 10.3: *Architecture of CNN*

Localization and object recognition with deep learning

Object recognition may be defined as the general term to describe a collection of computer-related vision tasks – image classification, object localization, and object detection – that involve identifying objects mainly in digital images.

First, let's understand these three computer vision tasks:

- **Image classification**: As the name implies, it involves predicting the class of an object in a digital image. The input for image classification would be an image with a single object and the output would be a class label.
- **Object localization**: It refers to locating the presence of one or more objects and drawing a bounding box to show their location. The input for object localization would be an image with one or more objects and the output would be one or more bounding boxes depending on the number of objects in an image.
- **Object detection**: You can say it is the combination of image classification and object localization because it refers to locating the presence of one or more objects and drawing a bounding box to show their location along with predicting the classes of the located objects in an image. The input for object detection would be an image with one or more objects and the output would be one or more bounding boxes along with a class label for each bounding box.
- **Object segmentation**: It is another extension to the breakdown of computer vision tasks. In this, the instances of recognized objects, instead of indicating by a coarse bounding box, are indicated by highlighting the specific pixels of the object.

The following figure gives an overview of the object recognition computer vision tasks:

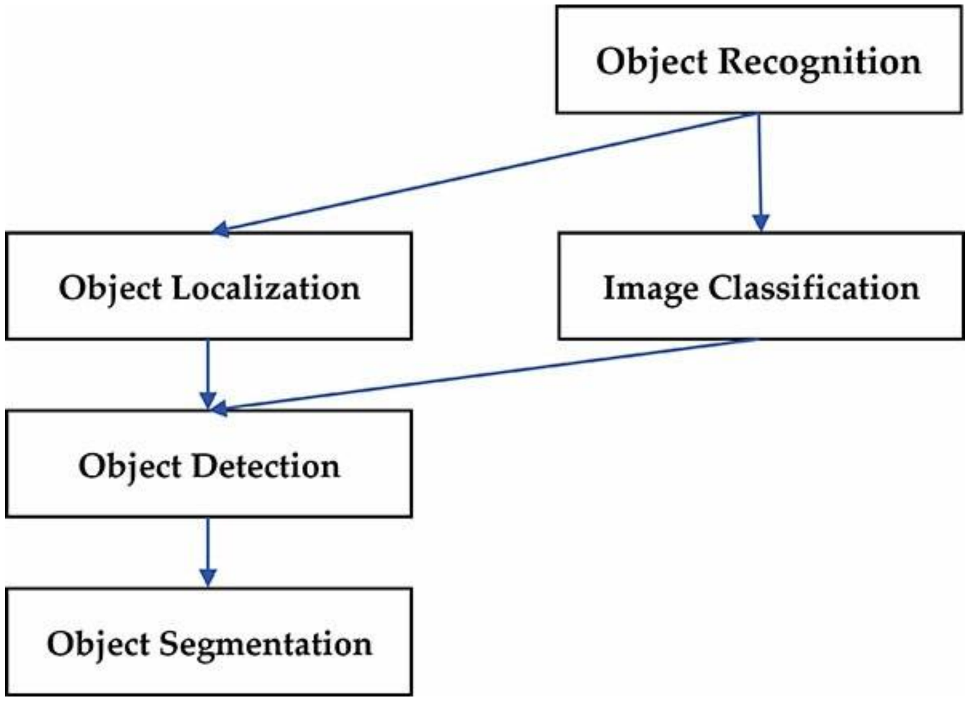


Figure 10.4: *Object recognition computer vision tasks*

We got most of the recent innovations in image recognition from **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**), which is an annual academic competition with a separate challenge for previously defined problems. Let's see an example of comparing single-object localization (a simpler version of the more broadly defined object localization) and object detection taken from the ILSVRC review paper[1] (refer *figure 10.5*).

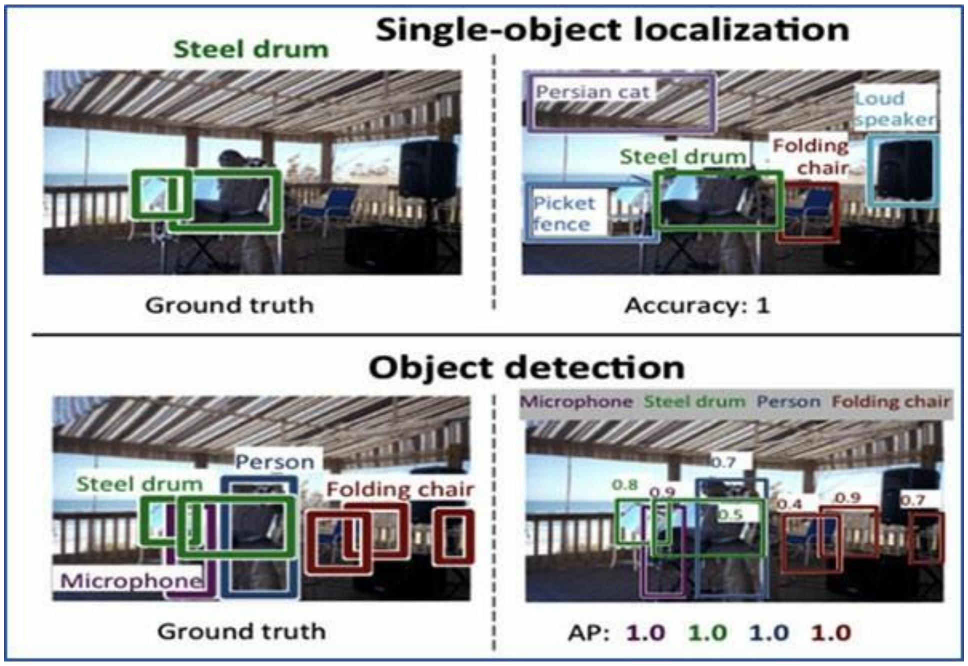


Figure 10.5: *Comparison between single-object localization and object detection*

Deep learning models

As we are now familiar with the problem of image classification, object localization, object detection, and object segmentation, next we will understand some top-performing deep learning models.

R-CNN Model family

R-CNN family of methods was developed by Ross Girshick, et al. It may stand for *Regions with CNN Features* or *Region-Based Convolutional Neural Network*. R-CNN model family includes three techniques, namely, R-CNN, Fast R-CNN, and Faster-RCNN. Let's have a look at all of them:

- **R-CNN**: The R-CNN, one of the first large and successful applications of CNN to the problem of object detection, object localization, and object segmentation, was described in the paper[2] by Ross Girshick, et al. Their proposed R-CNN model is having the following given modules:
  - **Region proposal**: This is the first module, and it generates and extracts category-independent region proposals. Example: candidate bounding boxes.
  - **Feature extractor**: This is the second module and as the name implies, it extracts

features from each candidate region. Example: Using a deep CNN.

- **Classifier**: This is the third module, and as the name implies, it classifies features as one of the known classes. Example: Linear Support Vector Machine Classifier model.

The architecture of the model, taken from the previously mentioned paper, is summarized as follows:

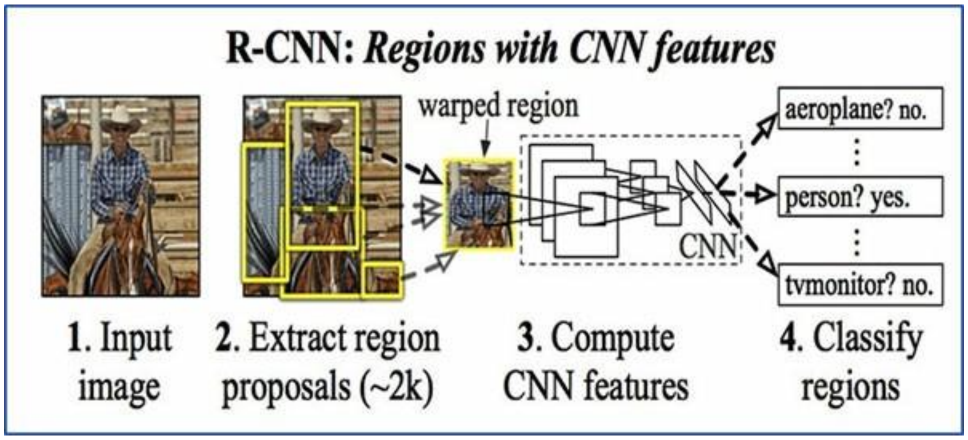


Figure 10.6: *R-CNN model architecture*

- **Fast R-CNN**: The Fast R-CNN is proposed by Ross Girshick in a 2015 paper[3] titled Fast R-CNN as an extension to address the following given limitations of R-CNN:

  - R-CNN involves the preparation and operation of three separate models, hence the training process in this technique is a multi-stage pipeline.
  - The training in R-CNN is expensive because training a deep convolutional neural network on so many region proposals per image is very slow.
  - Object detection in R-CNN is slow because making a prediction using a deep convolutional neural network on so many region proposals is very slow.

The architecture of the model, taken from the previously mentioned paper, is summarized as follows:

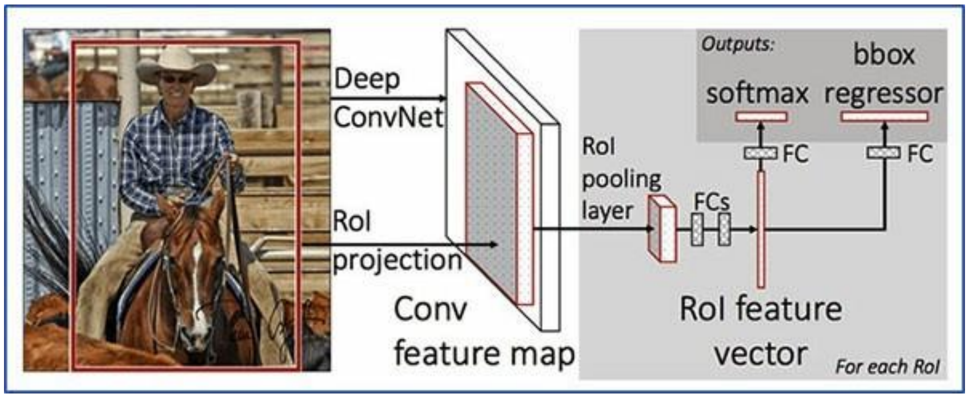


Figure 10.7: *Fast R-CNN model architecture*

As we can see in the architecture of the Fast R-CNN image, the model takes the photograph of a set of region proposals as input. This set of region proposals then passed through a deep CNN and here a pre-trained CNN (such as a VGG-16) is used for extracting features. The end of this deep CNN is a custom layer extracting feature specific for a given input candidate region. This custom layer is called **Region-of-Interest (RoI**) Pooling. After this, the fully connected layer interprets the output of the CNN and bifurcates it into two outputs. One output will be used for class prediction via a SoftMax layer and the other output will be used with a linear output for the bounding box. The whole process is repeated many times for each region of interest in a given photograph.

- **Faster R-CNN**: The Faster R-CNN is proposed by Shaoqing Ren et. al in a 2016 paper[4] titled "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks" with both improved speed of training and detection. Their proposed Faster R-CNN model has the following given modules:
    - **Region Proposal Network**: This is the first module, and it consists of CNN for proposing regions and the type of object to consider in the region as well.
    - **Fast R-CNN**: This is the second module and as the name implies, it uses CNN to extract features from the proposed regions and can output the bounding box along with class labels.

The first module, that is the region proposal network, acts as an attention mechanism for the second module, that is the fast R-CNN network. The architecture of the model, taken from the previously mentioned paper, is summarized as follows:

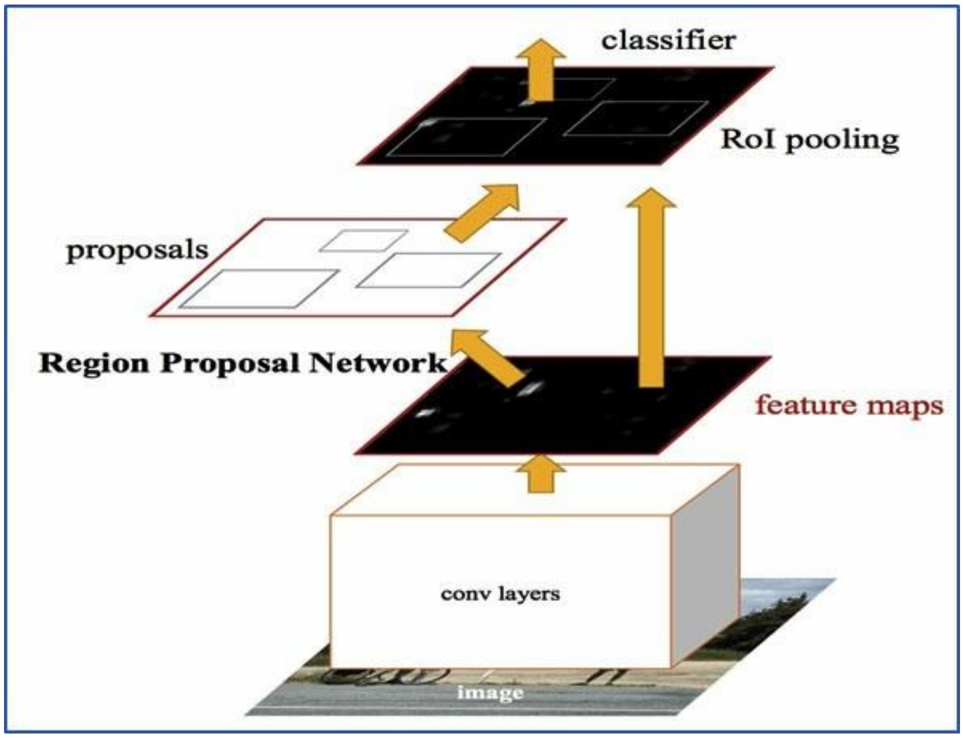


Figure 10.8: *Faster R-CNN model architecture*

YOLO model family

YOLO family of methods, another popular family of object recognition model, was developed by Joseph Redmon, et al. It stands for *You Only Look Once*. YOLO family of models achieve object detection in real time and hence are much faster than R-CNN models. YOLO model family includes techniques, namely, YOLO, YOLOv2, and YOLOv3. Let's have a look at all of them:

- **YOLO:** The YOLO model, one of the fastest and successful applications of CNN to the problem of object detection, was described in the paper[5] by Joseph Redmon, et al. The YOLO approach single NN trained end-to-end. The trained NN takes the photograph as input and predicts both bounding boxes and their class labels directly. This technique is relatively fast as it operates at 45 frames per second and up to 155 frames per second. One of the disadvantages of this technique, as compared with R-CNN, is that it offers lower predictive errors that is, having more localization errors.

  The working of YOLO model starts by splitting the input image into a grid of cells. Each cell is responsible for predicting a bounding box and its class labels. This prediction is done if the center of a bounding box falls within the cell. For example, an image may be divided into 7× 7 grids. If each cell in the grid may predict 2 bounding boxes, the output will be 94 proposed bounding box predictions. The summary of predictions made by YOLO model, taken from the previously mentioned paper, is summarized as follows:

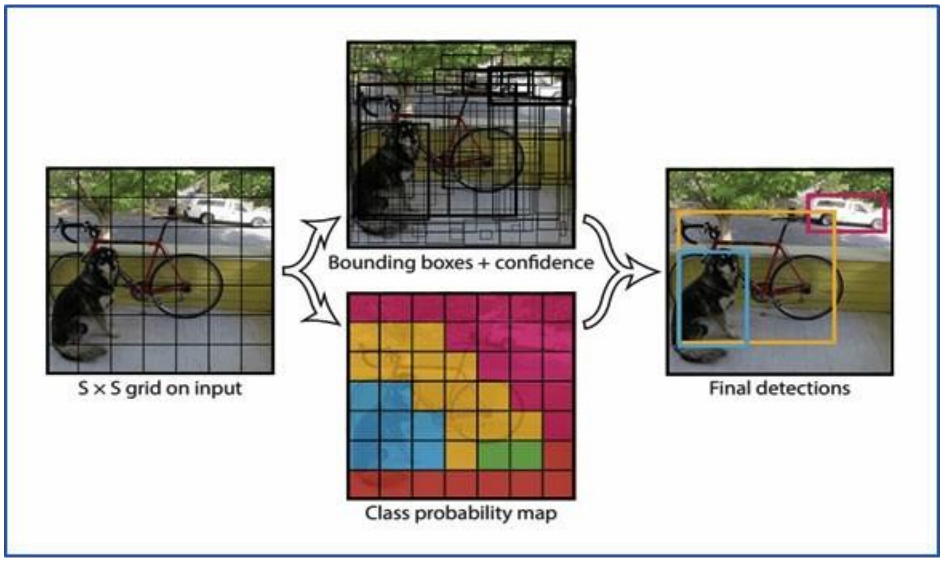


Figure 10.9: *Summary of predictions made by YOLO*

- **YOLOv2 and YOLOv3**: To further improve the YOLO model performance, the YOLOv2 model was purposed by Joseph Redmon and Ali Farhadi in their paper[6] titled *YOLO9000: Better, Faster, Stronger*. This model is referred to as YOLOv2 but as it was trained on two object recognition datasets in parallel and capable of predicting 9000 object classes, it is given the name YOLO9000. It uses batch normalization, high-resolution input images, anchor boxes (as used in faster R-CNN), and pre-defined bounding boxes with useful shapes and sizes.

The following is the example, taken from the previously mentioned paper, of the representation chosen while predicting the bounding box position and shape:

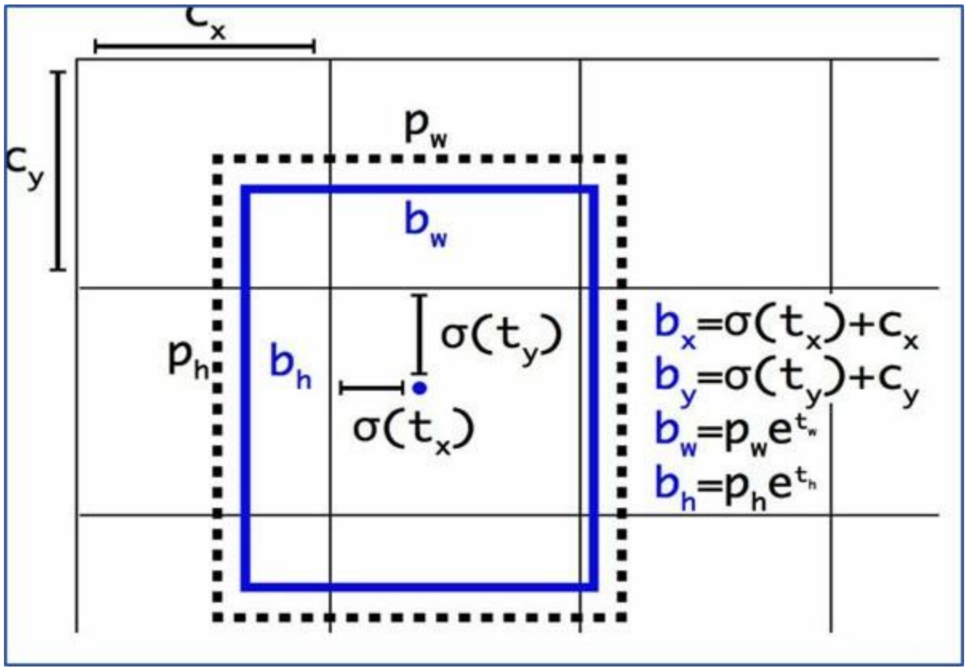


Figure 10.10: *Example of the representation chosen while predicting bounding box position and shape*

YOLOv3 model is having some reasonably minor improvements. It was proposed by Joseph Redmon and Ali Farhadi in their paper[7] titled *YOLOv3: An Incremental Improvement*.

Image classification using CNN in Python

Here, we are going to create an image classifier using CNN in the Python programming language. The image classifier distinguishes which category or class the input image belongs to. For this, we will use Keras deep learning library, a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theno. You can learn more about Keras at **https://keras.io/**. To install Keras on your computer system, use the following commands:

```
pip install keras
```

To install it using `conda`, the command would be as follows:

```
conda install -c conda-forge keras
```

For training and testing the dataset of images, we will use the dataset of cats and dogs, which you can download from the link **https://www.kaggle.com/c/dogs-vs-cats/data**.

```
# Importing the required libraries and packages.
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from  keras.layers  import  Flatten
from keras.layers import Dense
from keras.layers import Activation
```

```
# Initializing the CNN by using the Sequential Class from keras.
Image_Classifier = Sequential()

# Adding the first convolutional layer.
Image_Classifier.add(Conv2D(filters=32,kernel_size=(3, 3), input_shape = (64, 64,
3), activation = 'relu'))
```

Let's understand the working of arguments we passed previously in the Convolutional layer:

- **`filters`**: It denotes the number of feature detectors.
- **`kernel_size`**: It denotes the shape of feature detector. For example, (3, 3) represents a matrix.
- **`input_shape`**: It will standardize the size of the input image.
- **`activation`**: It represents the activation function to introduce non-linearity. Here we will use the ReLU activation function.

```
#Adding a pooling layer.
Image_classifier.add(MaxPooling2D(pool_size = (2, 2)))
```

The argument **`pool_size`** is representing the shape of the pooling window:

```
# Adding the flatten layer that will convert the data into a 1-Dimensional array.
Image_classifier.add(Flatten())

# Adding Full-Connection layers consisting of two layers, Hidden layer and Output
layer.
Img_classifier.add(Dense(units = 128, activation = 'relu'))
Img_classifier.add(Dense(units = 1, activation = 'sigmoid'))
The argument units are representing the number of nodes in the layer:

#Compiling our classifier.
Image_classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',
metrics = ['accuracy'])
```

The details of arguments are given as follows:

- **Optimizer**: It is used to reduce the cost calculated by cross-entropy, which is a measure of the difference between two probability distributions for random variables.
- **Loss**: It is used to calculate the error.
- **Metrics**: It is used to represent the efficiency of the model.

```
# Rescaling the images by using ImageDataGenerator.
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,shear_range = 0.2,
zoom_range = 0.2,horizontal_flip = True)
test_datagen = ImageDataGenerator(rescale = 1./255)
```

The details of arguments are described as follows:

- **`rescale`**: It represents a rescaling factor. The default value of rescale argument is none. If the value is none or **`0`**, there will be no rescaling else the data will be multiplied by the value provided in the argument.
- **`shear_range`**: It represents the sheer intensity.
- **`zoom_range`**: It represents the range for random zooming of the input image.

```
# Now fit the CNN to the images that lets the classifier directly identify
the labels form the name of the directories where images lie in.
training_set =
train_datagen.flow_from_directory('D:/Dataset/train',target_size = (64,
64),batch_size = 32,class_mode = 'binary')
test_set = test_datagen.flow_from_directory('D:/Dataset/test',target_size =
(64, 64),batch_size =32,class_mode = 'binary')
```

The details of arguments are described as follows:

- **`directory`**: It gives the location of the **`training_set`** and **`test_set`** both.
- **`target_size`**: It represents the dimension to which all input images will be resized.
- **`Batch_size`**: It represents the size of the batches of data. Its default value is **`32`**.
- **`Class_mode`**: This argument will determine the type of label arrays that are returned.

```
# Training and evaluating our classifier.
Image_classifier.fit_generator(training_set,steps_per_epoch = 4000,epochs =
15,validation_data = test_set,validation_steps = 10)
```

The details of arguments are given as follows:

- **`training_set`**: As the name implies, it represents the sequence used to train the neural network.
- **`Steps_per_epoch`**: This argument represents the total number of steps.
- **`epochs`**: It represents one complete cycle of predictions of the neural network.
- **`validation_data`**: As the name implies, this argument represents the sequence used to test the neural network.
- **`validation_steps`**: As the name implies, this argument represents the total number of steps to yield from **`validation_data`**.

```
#Making new predictions by providing test image to our classifier.
import numpy as np
from keras.preprocessing import image
test_image = image.load_img('{File file path to image}', target_size = (64,
64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = Image_classifier.predict(test_image)
training_set.class_indices
if result[0][0] == 1:
   prediction = 'dog'
else:
     prediction = 'cat'
print(prediction)
```

Conclusion

In this chapter, we learned about the basics of deep learning and convolutional neural network, which is one of the most popular deep neural networks. We also constructed an image classifier using CNN in the Python programming language.

We got to know that the difference between CNNs and ordinary NNs is in the types of layers along with how they both treat the input data. Ordinary NNs ignore the structure of the input data and convert all the input data into one-dimensional data before feeding it into the network for final processing. Whereas, while processing, CNN architecture considers the two-dimensional structure of images. The CNN architecture is designed in such a way that it extracts the properties specific to images. We also learned about the architecture of CNN and its layers. From the implementation perspective, we learned how to build an image classifier in the Python programming language by using CNN.