المملكة العربية السعودية

VISION رؤية 2030 المملكة العربية السعودية KINGDOM OF SAUDI ARABIA

وزارة التعليم
Ministry of Education

Artificial Intelligence

1444 - 2023 Edition

Third year
Secondary stage - Pathways system

binarylogic

Artificial Intelligence   Third year - Secondary stage - Pathways System   1444 - 2023 Edition

Dear student

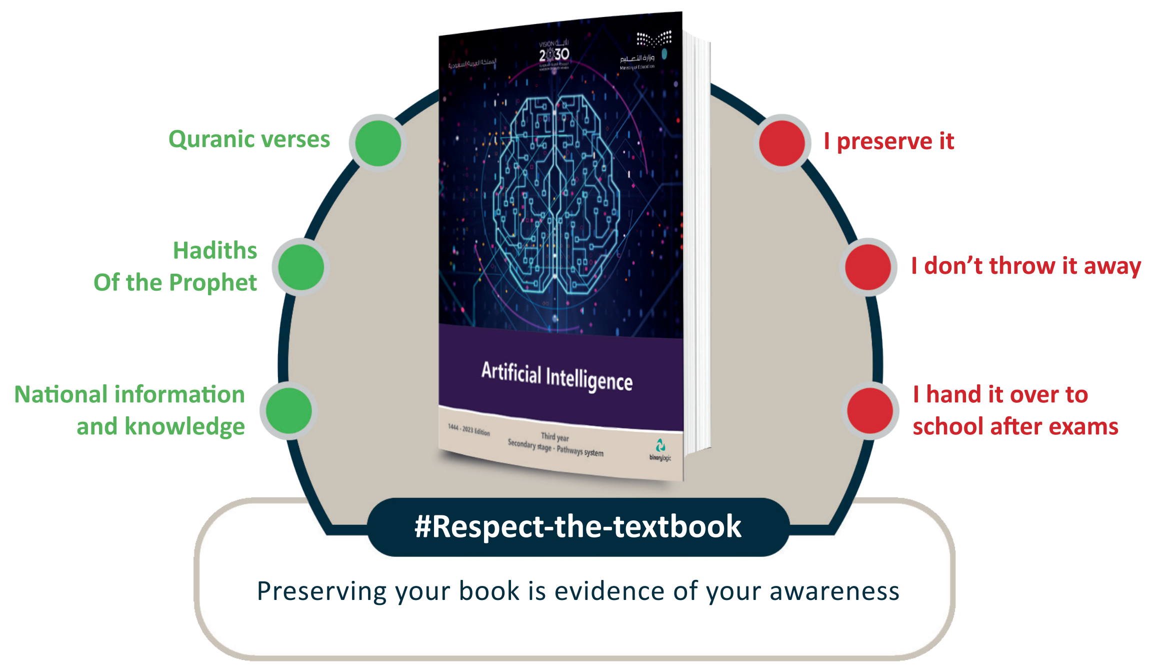
There are significant efforts behind the completion of this book, in the process of its preparation, review, and development, and there are funds spent to print it and deliver it to you, to support your learning and your scientific and moral advancement, so, try to be faithful to this effort, appreciating it by preserving your book.

Deposit number: **1444/10749**
ISBN: **978-603-511-481-3**

التعليمية TALEMIA

وزارة التعليم
Ministry of Education
2023 - 1445

Name: .... School: ....


Quranic verses
Hadiths Of the Prophet
National information and knowledge
I preserve it
I don’t throw it away
I hand it over to school after exams
Artificial Intelligence
#Respect-the-textbook
Preserving your book is evidence of your awareness

وزارة التعليم
Ministry of Education
2023 - 1445

قــــررت وزارة الــتـعلـيــم تــدريــس
هــذا الكـتـاب وطبـعه علـى نفقتها

وزارة التــعــليم
Ministry of Education

المملكة العربية السعودية

Artificial Intelligence

Secondary stage - Pathways system

Second year

وزارة التــعــليم
Ministry of Education
2023 - 1445

1444 - 2023 Edition

Publisher: Tatweer Company for Educational Services

Published under a special agreement between Binary Logic SA and Tatweer Education Services Company (Contract No. 0003/2022) for use only in the Kingdom of Saudi Arabia

binarylogic

وزارة التعليم
Ministry of Education
2023 - 1445

King Fahd National Library Cataloging-in-Publication Data

Ministry of Education

Artificial Intelligence / Ministry of Education

Riyadh, 2023

346p.; 210*25.5cm

ISBN:

1- Artificial Intelligence 2- Curriculum I-Title

006.3 dc 1444/10749

L.D. no.: 1444/10749

ISBN: 978-603-511-481-3

Educational Support Materials at "iEN Ethraia Platform"

ien.edu.sa

Dear students, parents and anyone interested in education, we welcome your communication to improve our textbooks. Your suggestions are our top priorities.

fb.ien.edu.sa

Dear teachers and educational supervisors, we appreciate your participation in developing the new textbooks. Your input will have a definite impact on supporting and improving the educational process for our students.

fb.ien.edu.sa/BE

Introduction:

The progress and development of countries is measured by the ability to invest in education, and the extent to which their educational system responds to the requirements and changes of the generations. In the interest of the Ministry of Education sustaining the development of its educational systems, and in response to the vision of the Kingdom of Saudi Arabia 2030, the Ministry of education has taken the initiative to adopt the “Secondary Education Pathways” system to bring about an effective and comprehensive change in high school.

The secondary education pathways system provides a distinguished and modern educational model for high school in the Kingdom of Saudi Arabia, which efficiently contributes to:

- Strengthening the values of belonging to our homeland “the Kingdom of Saudi Arabia” and loyalty to its wise leadership “may God protect him” based on a pure belief supported by the tolerant teachings of Islam.
- Strengthening the values of citizenship by focusing on them in school subjects and activities, in line with the demands of sustainable development, and the development plans in the Kingdom of Saudi Arabia that emphasize the consolidation of both values and identity, based on the teachings of Islam and its moderation.
- Qualifying students in line with future specializations in universities or the required jobs; ensuring the consistency of education outputs with the labor market requirements.
- Enabling students to pursue education in their preferred path at early stages, according to their interests and abilities.
- Enabling students to join specific scientific and administrative disciplines related to the labor market and future jobs.
- Participation of students in an enjoyable and encouraging learning environment in school based on a constructive philosophy and applied practices within an active learning environment.
- Delivering students through an integrated educational journey from the primary level to the end of the high school level and facilitating their transition process to post-general education.
- Providing students with technical and personal skills that help them deal with life and respond to the requirements of their level.
- Expanding opportunities for graduate students through various options in addition to universities, such as: obtaining professional certificates, joining applied faculties, and earning job diplomas.

The pathways system consists of nine semesters that are taught over three years, including a common first year in which students receive lessons in various scientific and humanities fields, followed by two specialized years, in which students study a general path and four specialized paths consistent with their interests and abilities, which are: the Rightful path, Business Administration path, Computer Science and Engineering path, Health and Life path, which makes this system the best for students in terms of:

- The existence of new study subjects that match the requirements of the Fourth Industrial Revolution and development plans, and the Kingdom’s Vision 2030, which aims to develop higher-order thinking, problem-solving, and research skills.
- Elective field programs that are consistent with the needs of the labor market and students› interests, as they enable students to join a specific elective field according to a specific job skill.
- Scale as it ensures the achievement of students› efficiency and effectiveness, and helps them identify their tendencies and interests, and reveal their strengths, which enhances their chances of success in the future.
- Volunteer work designed specifically for students in line with the philosophy of activities in schools, and is one of the graduation requirements; which helps to promote human values, and build society (its development and cohesion).

- Bridging which enables students to move from one path to another according to specific mechanisms.
- Proficiency classes through which skills are developed and the achievement level improved, by providing enrichment and remedial mastery classes.
- The options of integrated learning and distance learning, which are built in the paths system based on flexibility, convenience, interaction and effectiveness.
- The graduation project that helps students integrate theoretical experiences with applied practices.
- Professional and skill certificates granted to students after completing specific tasks, and certain tests compatible with specialized organizations.

Accordingly, the computer science and engineering path as one of the updated paths at the secondary level contributes to achieving best practices by investing in human capital, and transforming the student into a participating and productive individual for science and knowledge, while providing him with the skills and experience necessary to complete his studies in fields that meet his interests and abilities, or to join the labor market.

Artificial Intelligence is one of the main subjects in the Computer science and Engineering Pathway as it contributes to clarifying the concepts of Artificial Intelligence and the technologies associated with it, which are employed in several areas, such as smart cities, education, agriculture, medicine, and other economic fields. This course aims to introduce the student to the importance of Artificial Intelligence and its role in Industry 4.0. It focuses first on the basic building blocks of Artificial Intelligence technologies and after that, there is a deep dive into advanced applications for rule-based systems and Natural Language Processing systems. This course also includes projects and practical exercises for what the student learns. There are also realistic exercises for the student to solve that stimulate his cognitive levels under the guidance and supervision of the teacher.

The Artificial Intelligence book is characterized by modern engagement methods, which make students can learn and interact with it through the various exercises and activities it provides. This book also emphasizes important aspects of artificial intelligence education and learning, which are:

- The connection between the content and real-life problems.
- Diversity of ways to display engaging content.
- Highlights the role of the learner in the teaching and learning processes.
- Attention to the contents› structure and coherence.
- The skill of employing appropriate techniques in different situations.
- The ability to employ various methods in evaluating students in proportion to their individual differences.

To be on pace with global developments in this field, the Artificial Intelligence book will provide the teacher with an integrated set of diverse educational materials that take into account the individual differences between students, in addition to educational software and websites, which provide students with the opportunity to employ modern technologies and practice-based communication; This solidifies its role in the teaching and learning process.

As we present this book to our dear students, we hope it will capture their interest, meet their requirements, and make learning this material more enjoyable and useful.

God grants success

Contents

بسم الله الرحمن الرحيم

وزارة التعـــليم
Ministry of Education
2023 - 1445

Part 1

وزارة التعليم
Ministry of Education
2023 - 1445

وزارة التعليم
Ministry of Education
2023 - 1445

1. Basics of Artificial Intelligence

In this unit, you will learn about the history and the applications of Artificial Intelligence (AI). You will also learn more advanced data structures such as queues, stacks, linked lists, graphs and binary trees. These are the structures that you will use later to create AI projects.

Learning Objectives

In this unit, you will learn to:

> List the milestones of AI history.

> Cite examples of AI applications.

> Describe the operations of the stack data structure.

> Describe the operations of the queue data structure.

> Determine the differences between the stack and the queue data structure.

> Describe the main operations on the data of a linked list.

> Explain the use of tree data structure.

> Determine the differences between the tree and the graph data structure.

> Use Python programming language to explore complex data structures.

Tools

> Jupyter Notebook

Lesson 1

Introduction to Artificial Intelligence

Link to digital lesson

www.ien.edu.sa

What is Artificial Intelligence (AI)?

AI is the field of Computer Science that deals with the design and implementation of programs that are capable of imitating human cognitive abilities. These programs display characteristics that we usually attribute to human behavior, such as problem solving, learning, decision making, reasoning, planning, taking actions, etc.

AI agents

An AI agent is a software program that acts on a user's or system's behalf by perceiving its environment, making decisions, and taking actions based on those decisions. An agent can be simple or complex, autonomous or semiautonomous, and can operate in various environments, such as web-based, physical, or virtual.

Neural networks

Neural networks are a type of computer program that are designed to simulate the way the human brain works. They are made up of interconnected "neurons and layers" that can process and transmit information.

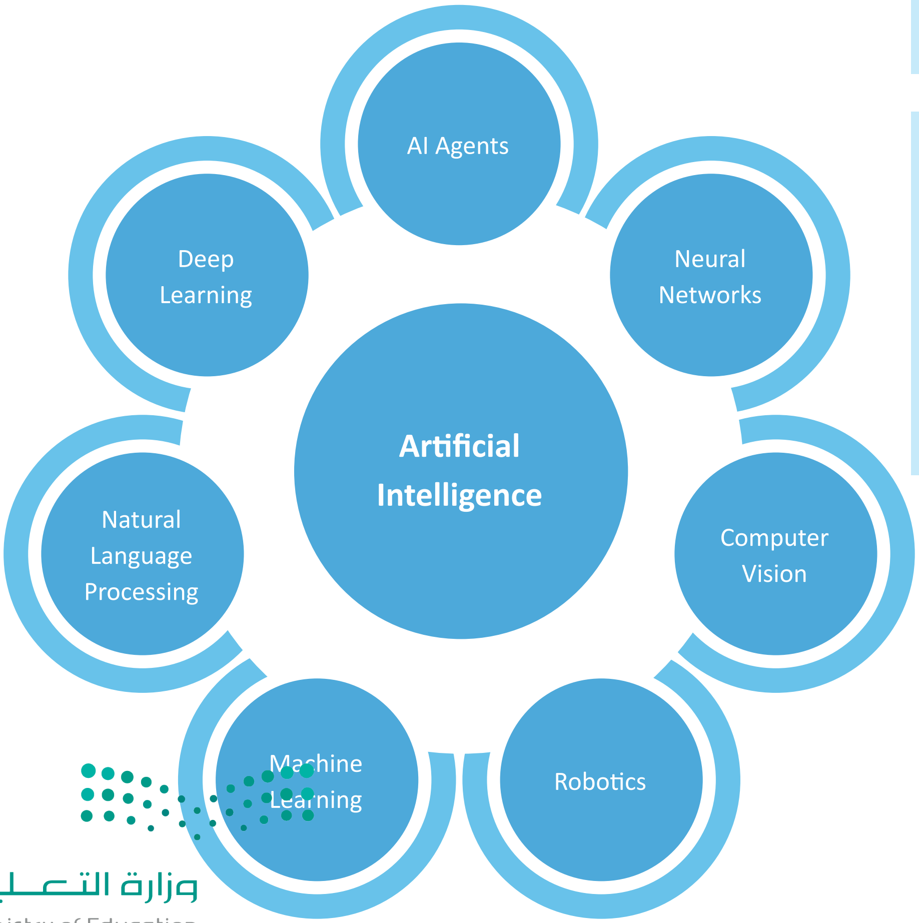


Figure 1.1: Some AI fields

AI and Other Fields

AI also has strong connections to multiple other fields including:

Philosophy: Philosophy is the ancestor of modern science. Philosophy studied fundamental problems that are central to AI, such as the origin and representation of knowledge, logical and rule-based reasoning, goal-based analysis, and the connection between knowledge and action.

Mathematics: Mathematics as a field serves as the core of AI and provides it with fundamental building blocks such as logic, computation, and probability theory.

Decision Theory: Decision theory studies logical and mathematical properties of the decision-making process. It analyzes how decisions are made in a system where the decision environment is uncertain. Theoretical frameworks and methods from this field have been consistently applied to AI problems.

Neuroscience: Neuroscience is defined as the scientific study of the human nervous system. The key neuroscience finding that a collection of simple cells can lead to complex outcomes such as thought, action, and consciousness has been a guiding principle for AI. In fact, artificial neural networks often emulate actual neural architectures found in the human brain.

Cognitive Psychology: Cognitive Psychology is a branch of psychology, which is dedicated to studying how people think. Advances in this field have consistently informed breakthroughs in AI, by providing insight that can help computers emulate human thinking.

Computer Science and Engineering: The field of Computer Science and Engineering has provided AI with the necessary software and hardware platform it requires to go from theoretical concepts to practical applications. Advances in AI have been consistently supported by breakthroughs in operating systems, programming languages, storage, memory, and processing power.

Cybernetics: Cybernetics is defined as the study of systems that achieve a desired state by receiving information from their environment and modifying their behavior accordingly. The key difference is that Cybernetics uses mathematics to model closed systems that can be fully described by specific variables, while AI uses logical inference and computation to overcome such limitations and study complex problems such as the comprehension and generation of language and visual information.

Linguistics: Linguistics is the scientific study of human language. The comprehension and generation of human language have been a key application area for AI, leading to the creation of subfields like Natural Language Processing (NLP) and Computational Linguistics.

Vision Science: Vision Science is defined as the scientific study of visual perception. Teaching computers how to understand and generate images, animations and videos are one of the most exciting applications of AI and specifically in the Deep Learning and Computer Vision subfields.

INFORMATION

The term "Artificial Intelligence" was formally introduced in 1956, making AI one of the youngest disciplines.

Turing Test

Perhaps the most famous method for defining AI, which was proposed in 1950 is the Turing Test: an experiment for determining whether a computer is intelligent or not.

During the test, the computer has to answer some written questions provided by a human interrogator. The test is considered successful if the interrogator cannot tell whether the written response came from a person or from a computer.

To successfully pass the test, the computer needs to have the capabilities shown in the following table:

Turing Test

Turing Test tests the ability of a machine to exhibit intelligent behavior equivalent to or indistinguishable from that of a human.

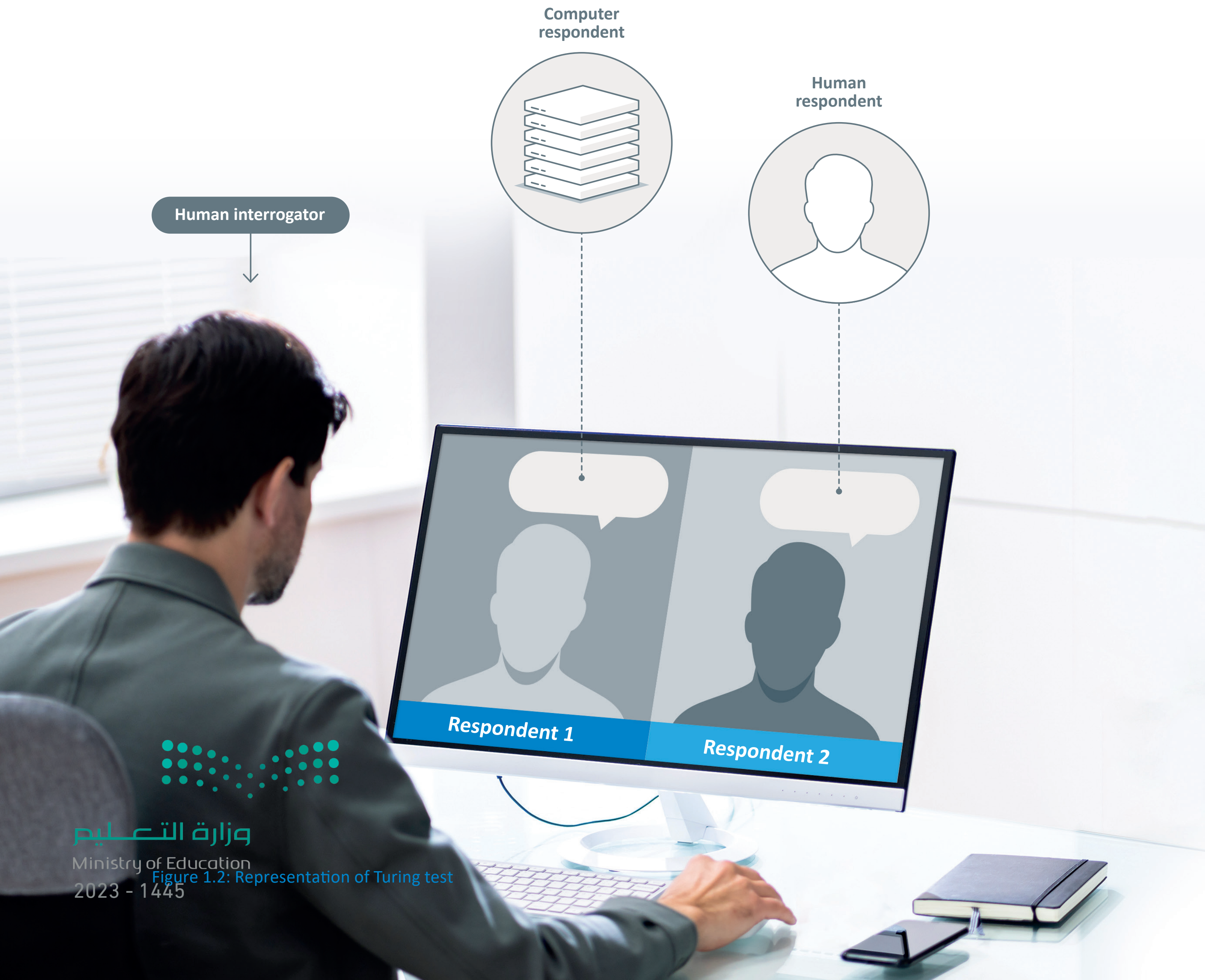


Figure 1.2: Representation of Turing test

Table 1.1: Computer capabilities to pass the Turing test

1	Natural Language Processing to enable it to understand and answer questions.
2	Knowledge Representation to organize, store and retrieve information during test performance.
3	Automated reasoning to use the stored information to answer the questions.
4	Machine Learning to adapt to new language constructs (e.g. different syntax or vocabulary) that it has not seen before and is not in its stored information.
5	Computer Vision, so that the computer can respond to visual signals provided by the interrogator via image and video feed.
6	Robotics, so that the computer can receive and process objects passed by the interrogator via a hatch.

The above capabilities cover a large part of the broad field of Intelligence.

Let's define some of these capabilities.

Natural Language Processing (NLP), is a branch of AI which gives computers the ability to understand human and natural language.

Knowledge representation in AI refers to the process of encoding human knowledge into a machine-readable form that can be processed and used by AI based systems. This knowledge can take many forms, including facts, rules, concepts, relationships, and processes.

Automated reasoning refers to the ability of an AI-based system to automatically deduce new knowledge and make logical inferences based on a set of given rules and premises.

Computer vision is a field of AI that enables computers to interpret and understand visual information from the world, such as images and videos.

Robotics is a branch of AI that deals with robot design, construction, and use. It involves the integration of various technologies, such as machine learning, computer vision, and control systems, to create intelligent machines that can perform tasks autonomously or with minimal human supervision.

Artificial Intelligence: 9 Decades of History

Despite being less than 100 years old, AI has had a rich history spanning from the 1940s until today. Let's look at a timeline of the main AI milestones in each decade.

1940s Early days and the first artificial neurons

1943: The first model based on artificial neurons is proposed. Each neuron could be in an active ("on") or inactive ("off") state, depending on the stimulation that it received from other neighboring neurons that it was connected to.

1948: Elmer and Elsie, two autonomous robots, can navigate their way around obstacles using light and touch.

1950s The founding of Artificial Intelligence

1950: The Turing Test is introduced: a test of a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human. A plethora of key AI concepts is also introduced such as machine learning, genetic algorithms, and reinforcement learning.

1951: Stochastic Neural Analog Reinforcement Computer (SNARC), the first neural network computer is built.

1958: Lisp is developed, a programming language designed specifically for AI. In the same year, a paper is published for a hypothetical Advice Taker, an AI system capable of learning from experience just like a human.

1960s & 1970s The First AI Winter

1964: ELIZA is the first NLP program and the ancestor of today's chatbots.

1974-1980: This period is referred to as the "First AI Winter''. Funding for AI projects was reduced during this time, due to the lack of progress and impact in real-world applications. One major criticism was the inability of AI techniques to address the combinatorial explosion problem, which limited their applicability to only very small problems and datasets.

1980s & 1990s Expert Systems and the Second AI Winter

1980: The first successful commercial expert system designed to emulate the decision-making ability of a human expert is released.

1987-1993: This period is referred to as the "Second AI Winter". The rule-based nature of early AI systems limits their applicability and makes them unable to solve key real-life problems.

1997: The Deep Blue supercomputer beats world chess champion Gary Kasparov. The first win of an AI program over a world chess champion.

2000s Mainstream popularity, supported by Hardware and Software breakthroughs

2005: Stanford University creates STANLEY, a self-driving car that wins an autonomous vehicle challenge. The U.S. military begins investing in autonomous robots.

2009: Deep-learning neural networks were trained with graphics processing units (GPUs) for the first time. The use of this specialized hardware rapidly accelerated the training of complex networks on very large datasets, ushering in a new age for deep learning and Artificial Intelligence.

2010s & 2020s Golden Age

2011: The question-answering system Watson defeats the world's two greatest Jeopardy! players. Watson was able to understand and successfully answer the questions, marking a breakthrough in using artificial intelligence to understand natural language.

2012: An AI system instantaneously translates spoken English to spoken Chinese.

2021: A full self-driving system uses a neural network trained on the behavior of hundreds of thousands of drivers.

2022: ChatGPT (Generative Pre-trained Transformer) is a chatbot built on top of large language models. The models are fine-tuned with both supervised and reinforcement learning techniques to mimic a human conversation.

Applications of AI

AI is a rapidly evolving technology that has the potential to transform a wide range of fields and industries. In this unit, you will explore the various applications of AI and how it is being used to lead to improvements and innovations in a wide range of domains and industries.

Virtual Assistants

One of the most popular applications of AI has been in the area of virtual assistants, that can communicate with users through voice or text-based interactions. They are often accessed through devices such as smartphones, tablets, or smart speakers, and can be used for a wide range of tasks such as setting reminders, answering questions, playing music, and placing orders for products and services. One of the most well-known examples of an AI-powered virtual assistant is Apple's Siri. Other companies have also developed their own virtual assistants, including Amazon's Alexa, Google's Assistant, and Microsoft's Cortana. These assistants have become increasingly sophisticated over time, with the ability to understand and respond to a growing number of commands and queries. For example they can be used to control a wide range of smart home devices, such as thermostats, lights, and appliances. Virtual assistants also come in the form of specialized chatbots, typically designed to provide information and answer questions in a particular domain.

An example of such a domain is customer service, where AI-powered chatbots are used to answer questions about products or services, troubleshoot issues, and provide information about orders and accounts. Chatbots can be accessed through a variety of channels, such as websites, messaging apps, and social media, and can provide assistance 24/7. You can see an example of a chatbot application in figure 1.3.

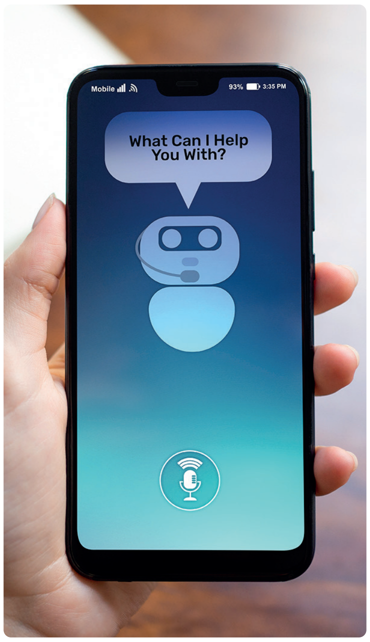


Figure 1.3: Conversation with chatbot

Robotics

AI has historically been linked to robotics. While a robot can be seen as the physical manifestation of an artificial being, AI represents the robot’s software brain, providing it with the ability to sense its environment, make decisions, and adapt to changing conditions. Intelligent robots can then apply these abilities to perform a wide range of tasks without human intervention. These tasks can include manufacturing, exploration, search and rescue, and many others. In figure 1.4, you can see a robot assembly line in a car factory

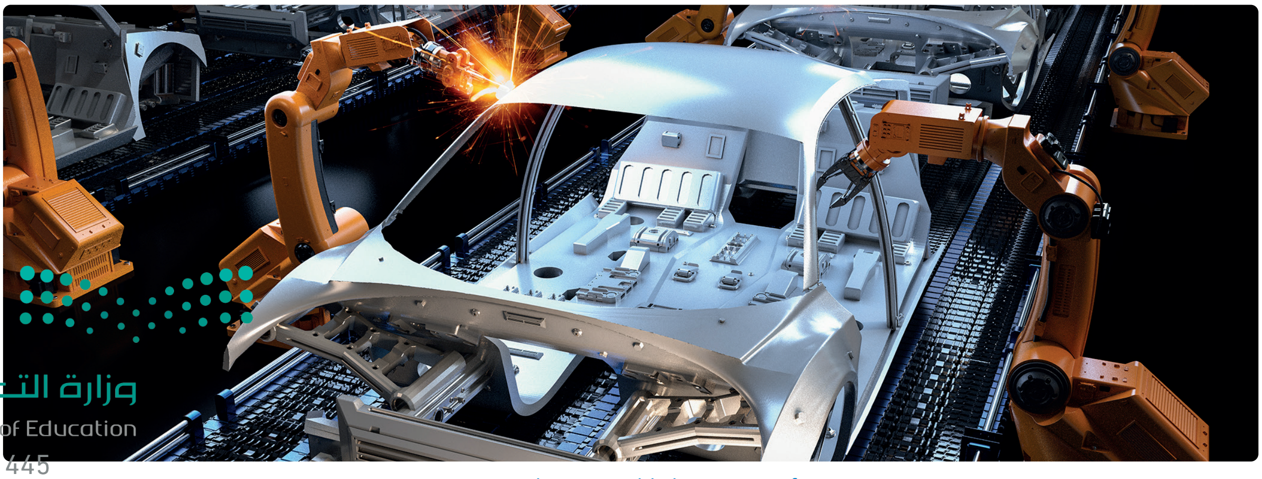


Figure 1.4: Robot assembly line in a car factory

One of the earliest examples of AI in robotics was the development of factory robots which were used to perform tasks like welding, painting, and assembly. Since then, the use of AI in robotics has become increasingly sophisticated, with the development of more advanced algorithms and the use of machine learning to improve robot performance. One milestone in the use of AI in robotics was the development of humanoid robots, like Honda's Advanced Step in Innovative Mobility (ASIMO), which was introduced in 2000 and was capable of walking and performing basic tasks.

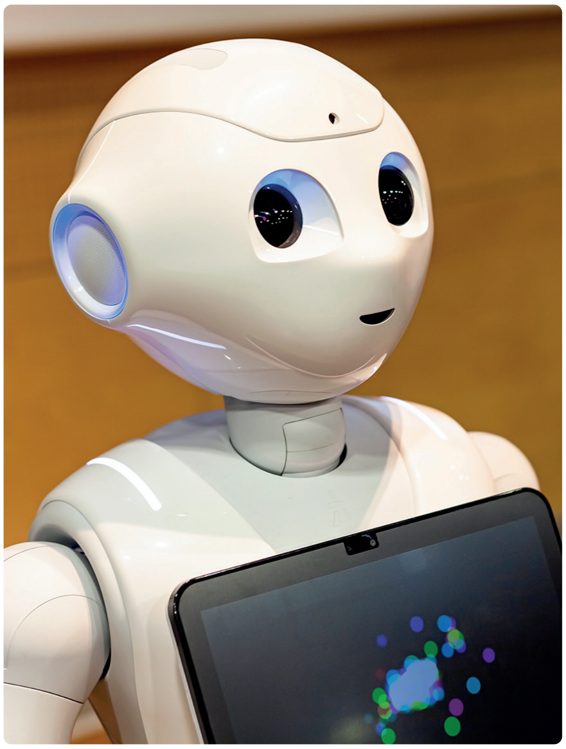


Figure 1.5: Pepper robot

Humanlike Robots

Pepper and Nao are humanoid robots developed by Aldebaran Robotics. Both robots are designed for human-robot interaction and are widely used in research, education, and entertainment. Pepper is a social robot designed to interact with people naturally, using its cameras, microphones, and touch sensors to perceive its environment and respond to people's actions and emotions. Pepper has many features that allow it to recognize faces, understand speech, and respond to gestures. You can see the Pepper robot in figure 1.5.

Nao is a smaller, more compact robot designed for human interaction. Like Pepper, Nao has a range of sensors that allow it to perceive its environment, as well as cameras and microphones for speech and facial recognition. Nao is highly customizable and programmable, making it an attractive choice for researchers and educators who want to study and develop new applications for humanoid robots.

In 2017 the robot Sophia was the first robot to receive Saudi citizenship and in 2023 Saudi Arabia's first interactive robot Sarah was introduced.

Self-Driving Cars

Another milestone was the development of self-driving cars (figure 1.6), which use AI to navigate roads and make decisions about how to safely interact with other vehicles and pedestrians. One of the key requirements of such applications is the ability to process and understand visual data, such as photos and videos, commonly referred to as "Computer Vision". Computer Vision Algorithms can be used to identify objects, people, and other features in images and videos, as well as to understand the context and meaning of the content. This has a wide range of applications beyond robotics, including facial recognition, content moderation, and media analysis. A key milestone in the use of AI in image and video analysis was the development of deep learning algorithms, which can analyze large amounts of data and identify complex patterns in images and videos.

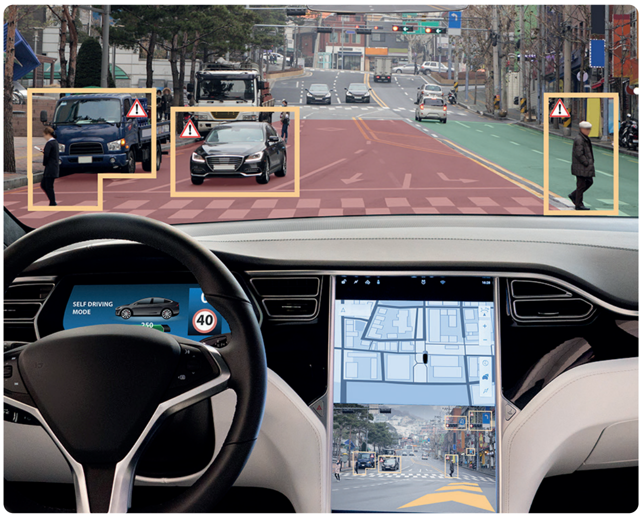


Figure 1.6: Self-driving car

Industries Affected by AI

Education

Over the past few decades, there have been several key milestones in the use of AI in education. Early examples include the development of AI-powered tutoring systems, which used NLP to interact with students and provide feedback on their work. Then, adaptive learning platforms emerged, using machine learning algorithms to personalize learning for each student based on their strengths and weaknesses. Next, AI-powered grading systems were developed, which used NLP and machine learning algorithms to grade written assignments and provide feedback.

More recently, virtual assistants and chatbots have been integrated into education to provide personalized support to students and answer their questions in real-time. AI can be used to analyze data about student performance, learning preferences, and other factors to create personalized learning plans and recommend materials or activities that are most likely to be effective for each student.

AI benefits in education

- Time-saving for teachers/ professors.
- AI tutors can assist students.
- Help teachers to become learning motivators
- AI-driven functionality can give feedback to students and educators

Healthcare

Healthcare is another field that has consistently enjoyed innovation thanks to advances in AI. The first innovations came in the form of AI-powered diagnostic systems and the use of AI in drug discovery. Next, AI was integrated into electronic health records to extract relevant information, and in the 2010s, AI-powered telemedicine systems were developed. Today, modern AI is used to create personalized treatment plans and power wearable devices that track a person's health. AI has played a significant role in the healthcare industry, enabling doctors and other healthcare professionals to analyze large amounts of data and make more informed decisions about patient care. Such data can come from diverse sources including medical records, lab tests, and even images such as X-rays and CT scans. Modern computer vision algorithms are nowadays routinely used to detect abnormalities and assist with diagnosis.

Figure 1.7: Analyzing health data

Agriculture and Climate Modeling

In agriculture, AI is used to optimize crop yields and improve the efficiency of farming practices. This is achieved by continuously analyzing data about soil conditions, weather patterns, and other factors to predict the best time to plant, irrigate, and harvest crops. AI can also be used to monitor crops in real time and identify problems, such as pests or diseases, allowing farmers to take corrective action before yields are significantly impacted. One of the earliest examples of AI in agriculture was the use of simple decision-making algorithms to optimize irrigation schedules. Another key milestone was the use of sensor networks to monitor crops and automatically calibrate the application of key treatments such as fertilizers and pesticides. More recently, the use of drones and satellite imagery has been used to analyze crops at a larger scale.In figure 1.8, you can see an autonomous drone to fertilize a field.

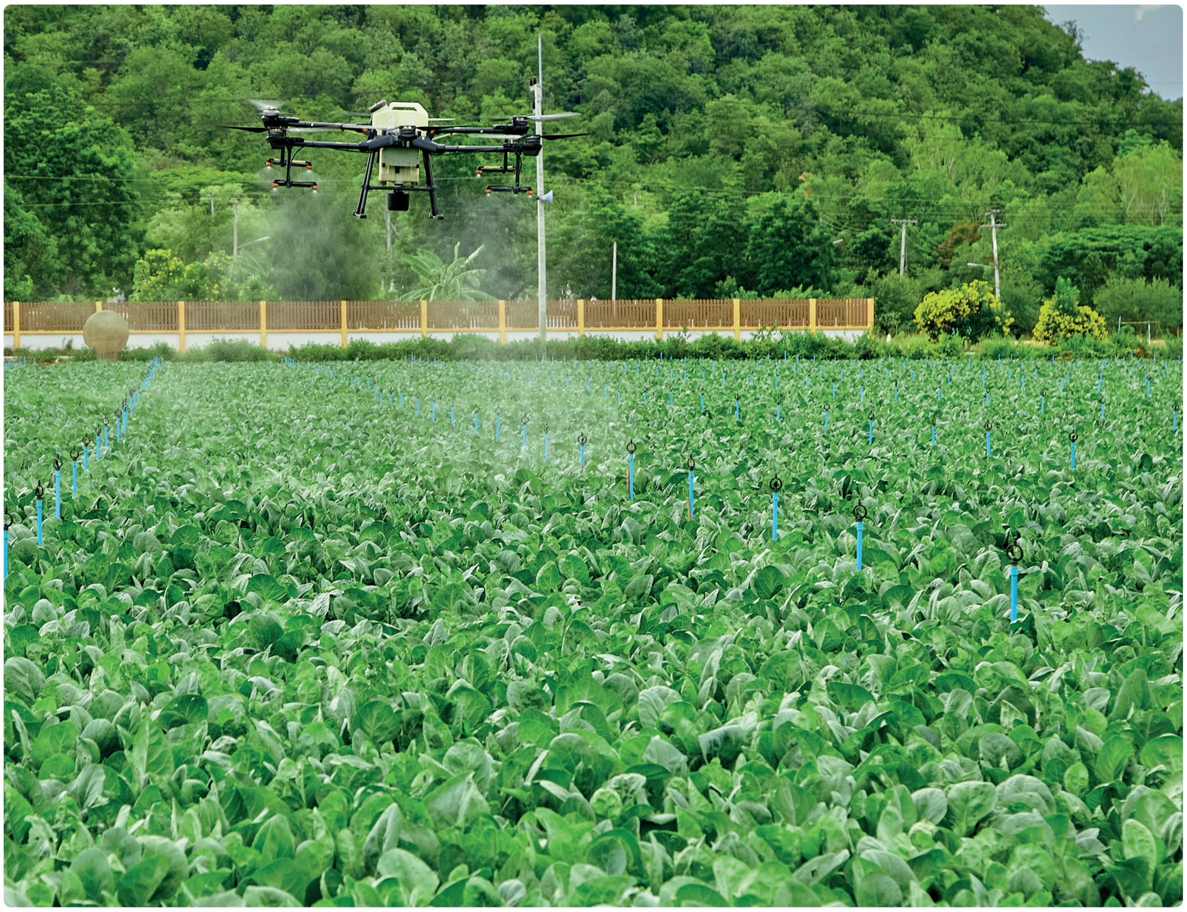


Figure 1.8: Fertilizing with autonomous drone

Another area that is closely related to agriculture and has also been significantly influenced by AI is climate modeling. Applications in this area started early, with the development of AI-powered weather forecasting systems. Later, AI was used to analyze large amounts of data on climate change and make predictions about future trends. Such data can come from various sources, including satellite imagery, weather station observations, and computer simulations. Today, AI is being used in a wide range of climate modeling applications, including predicting the impacts of climate change on specific regions, understanding the causes of extreme weather events, and identifying the most effective strategies for mitigating or adapting to climate change.

Energy

AI has had a significant impact on the energy industry, enabling companies to optimize energy use, reduce waste, and improve efficiency. One of the earliest examples was the use of machine learning algorithms to analyze data on energy use and identify ways to reduce waste and optimize consumption. In the 1990s, AI was used to predict the potential output of renewable energy sources and optimize their use. This was an important development as it allowed energy companies to better plan for the integration of renewable energy sources into their operations.

Figure 1.9: Clean electrical energy from solar photovoltaic panels

The 2000s saw the integration of AI into smart grids, which used machine learning algorithms to analyze data on energy use and adjust supply and demand in real-time. This helped to improve the efficiency of energy distribution and reduce waste. In the 2010s, AI was used to develop energy storage systems that could store excess energy and release it when needed. This was an important development as it allowed energy companies to better manage the intermittent nature of renewable energy sources, such as solar and wind. Figure 1.9 shows solar photovoltaic panels. In recent years, AI has been used to increase energy efficiency by analyzing data on energy use and identifying ways to reduce waste. This has included the development of AI-powered systems that can optimize the energy use of buildings, factories, and other large energy consumers. AI has also been used in the oil and gas industry to analyze data on drilling and production and optimize operations.

Law Enforcement

In law enforcement, AI is actively used to help predict and prevent crimes. Specifically, AI can be used to analyze data from sources such as crime records, social media, and surveillance cameras to identify and predict patterns and trends in criminal activity. Early examples include the development and the use of AI in facial recognition (figure 1.10). Later, AI was integrated into police dispatch systems and used to monitor social media platforms for potential threats. More recently, AI has been used to develop drones for surveillance and to analyze footage from body-worn cameras worn by law enforcement officers. AI has played a significant role in law enforcement, enabling agencies to analyze large amounts of data, identify patterns and trends, and make more informed decisions about how to prevent and respond to crime.

Figure 1.10: Face recognition and personal identification technologies

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. Mathematicians set the groundwork for understanding computation and reasoning about algorithms.	○	○
2. The Turing Test determines whether a computer has humanlike behavior.	○	○
3. Elmer and Elsie were the first robots to navigate obstacles using light and touch.	○	○
4. AI has only been used in the manufacturing industry for robots.	○	○
5. AI has not had any impact on the energy industry.	○	○

2 What is Artificial Intelligence (AI)?

____

____

____

____

____

3 Briefly explain some applications that AI is used for in real life.

____

____

____

____

____

4 Provide the key historical events that influenced the evolution of AI during the 1940s and 1950s.

5 Outline how, in the 2010s, commercial applications of AI technologies were introduced.

6 Summarize how AI applications can combat climate change through climate modelling and enhancements in the energy industry.

Lesson 2

Data Structures in AI

Link to digital lesson

www.ien.edu.sa

The Importance of Data Structures in AI

Data is critical in AI as it is the foundation for training machine learning models. The quality and quantity of data available determine the accuracy and effectiveness of AI models. Without sufficient relevant data, AI algorithms cannot learn patterns, make predictions or perform tasks effectively. Hence, data plays a crucial role in shaping AI systems' decision-making abilities and capabilities.

Data structures are important in AI because they provide an efficient way to organize and store data that allows for efficient retrieval and manipulation. They determine the complexity and efficiency of algorithms used to process data and thus directly impact the performance of AI systems. For instance, using an appropriate data structure can improve the speed and scalability of AI algorithms, making them more suitable for real-world applications. Additionally, well-designed data structures can help reduce memory usage and make algorithms more memory-efficient, enabling the processing of larger datasets.

Data Structure

A Data Structure is a technique to store and organize data in the memory so that it can be used efficiently.

Computers store and process data with extraordinary speed and accuracy. So, it is highly essential that the data is stored efficiently and can be accessed in a fast way.

Data Structures can be classified as follows:

- **Primitive Data Structures**.
- **Non-Primitive Data Structures**.

Simple data is also called primitive, raw, or basic data.

The diagram in figure 1.11 visualizes the classification of data structures.

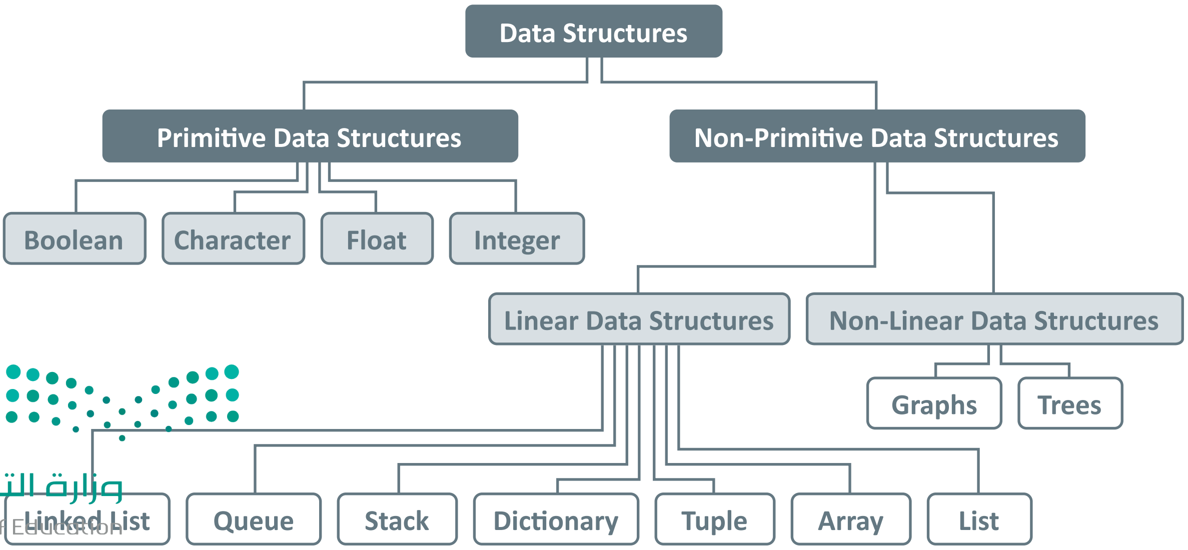


Figure 1.11: Data structures diagram

Primitive Data Structures

Primitive Data Structures are also referred to as basic data structures in Python. This type of structure contains simple values of data. Simple data types tell the compiler which type of data to store in it.

The basic data structures in Python are:

- **Numbers** (Numbers are used to represent numeric data)
  - Integers
  - Floating point number
- **Strings** (Strings are collections of characters and words)
- **Boolean** (A Boolean data type takes one of two values True or False)

Different types of data structures are used for different computer applications and tasks, based on the requirements of the project and the restrictions on memory.

Non-Primitive Data Structures

Non-Primitive Data Structures are specialized structures which store a group of values. They are created by the programmer and they are not defined by Python like the primitives.

Non-primitive data structures can also be divided into two categories:

- **Linear or sequential data structures.**
  The linear data structures store the data elements in a sequence
- **Non-linear data structures.**
  Non-linear data structures do not have a sequential linking between data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence

Linear Data Structures

Linear data structures store the data elements in a sequence. In this lesson, you will learn about some linear data structures such as **stack** and **queue**. These are two of the most common structures you will come across in your daily life.

Stack

A stack can actually be represented by a group of books stacked on top of each other, as shown in figure 1.12. To group a stack, you have to put the books one on top of another. When you want to use a book, you have to pick up the book at the top of the stack. To access the other books in the stack, you will have to remove the books from the top of the stack.

A stack can either have a fixed size or it can have a sense of dynamic resizing. Python implements stacks using lists.

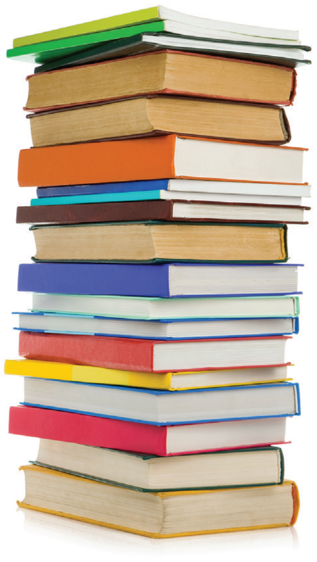


Figure 1.12: A stack of books as a real-life example

Last In First Out (LIFO) rule

The element which is added last, is accessed first.

Operations on the stack

There are two main operations on the stack:

- **Push**: This operation is used to add an element to the top of the stack.
- **Pop**: This operation is used to remove an element from the top of the stack.

Push Operation

The operation of adding a new element on the stack is called a push.

The stack uses a pointer called Top. The pointer points to the element on the top of the stack. When a new element is added to the stack:

- The value of the top pointer is increased by one to show the new position the element will be placed in.
- The new element is added to the top of the stack.

Stack Overflow

The stack has a specific capacity that depends on the computer's memory. If that capacity is full, adding a new element will cause the stack overflow. The stack should be checked for fullness before adding any element.

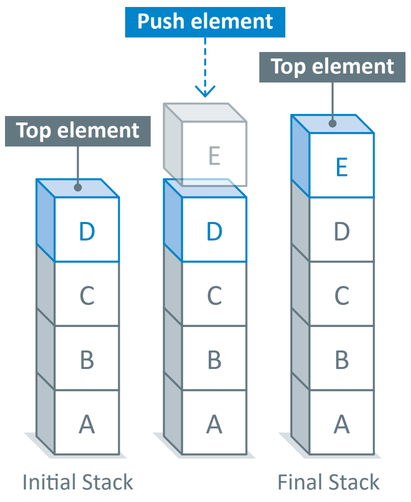


Figure 1.13: Push operation

Pop Operation

The operation of removing an element from the stack is called a pop.

When removing an element from the stack:

- The element at the top of the stack is removed.
- The value of the top pointer is decreased by one to show the element on the top of the stack.

Stack Underflow

If you want to remove an element from the stack, you must check first that the stack contains at least one element; If the stack is empty, you will cause a stack underflow.

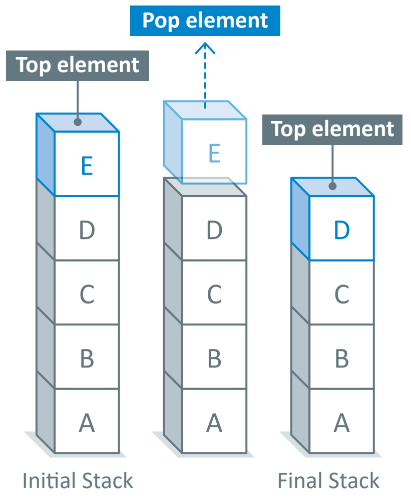


Figure 1.14: Pop operation

Stack in Python

Stacks are represented in Python using Lists which in turn provide some ready-to-use operations with stacks.

Table 1.2: Stack operations

Operation	Description
listName.append(x)	Adds the x element to the end of the list.
listName.pop()	Removes the last element from the list.

Let's see an example of the implementation of a stack in Python.

1. Create a stack to store a set of numbers (1, 21, 32, 45).
2. Use the pop operation twice to remove the last two elements (45, 32) from the stack.
3. Use the push operation to add a new element (78) to the stack.

The push operation of the stack is implemented in Python by using the append function.

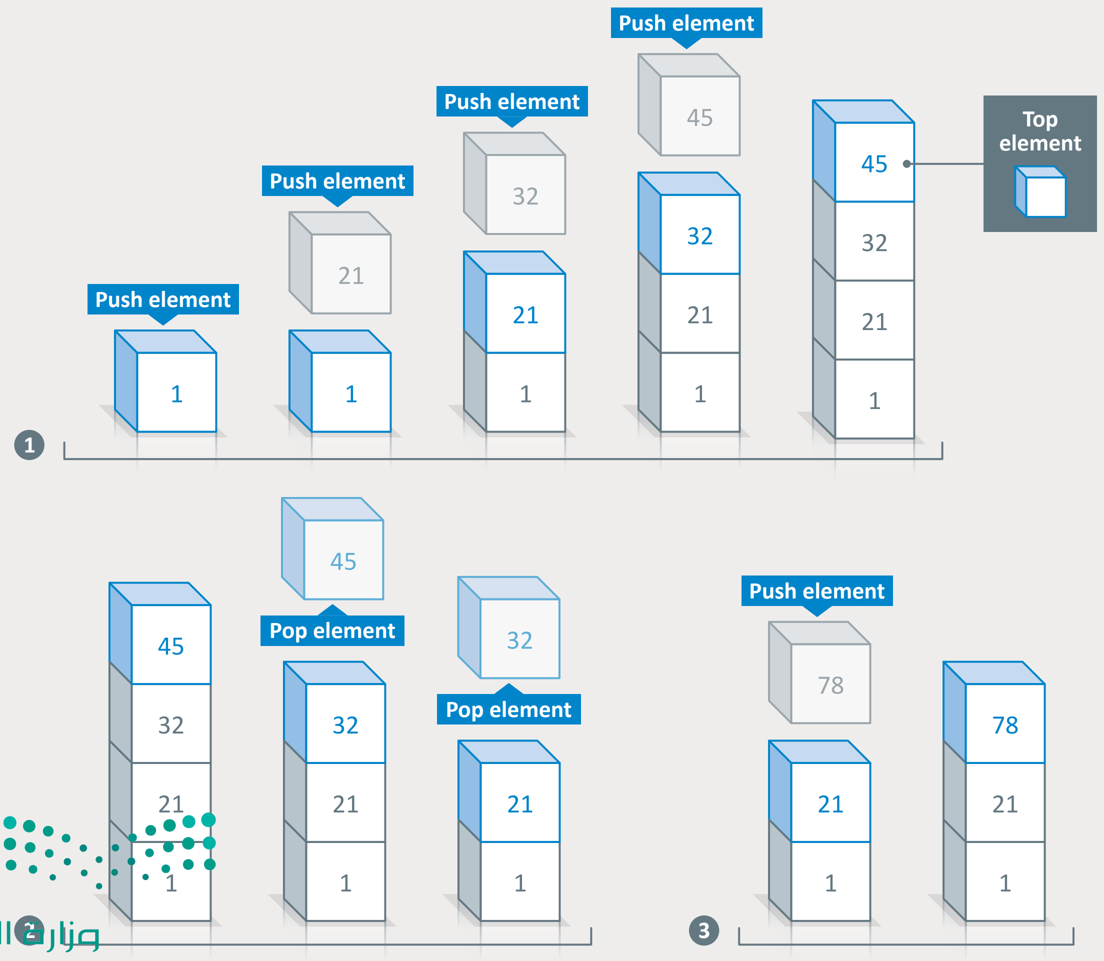


Figure 1.15: Stack Example

Jupyter Notebook

In this unit, you will write a Python code using Jupyter Notebook. Jupyter Notebook is an online web application to create and share computational documents. Each document, called a notebook, includes your code, comments, raw and processed data, and data visualizations. You will use the offline version of Jupyter Notebook.

The easiest way to install it locally is through Anaconda, an open-source distribution platform, which is free for students and hobbyists. Download and install Anaconda from here: **https://www.anaconda.com/products/distribution**.

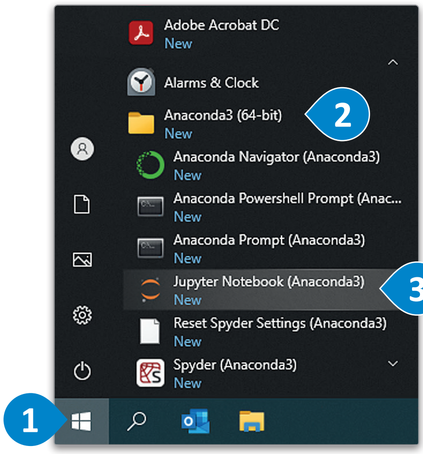
Python and Jupyter Notebook will be installed automatically.

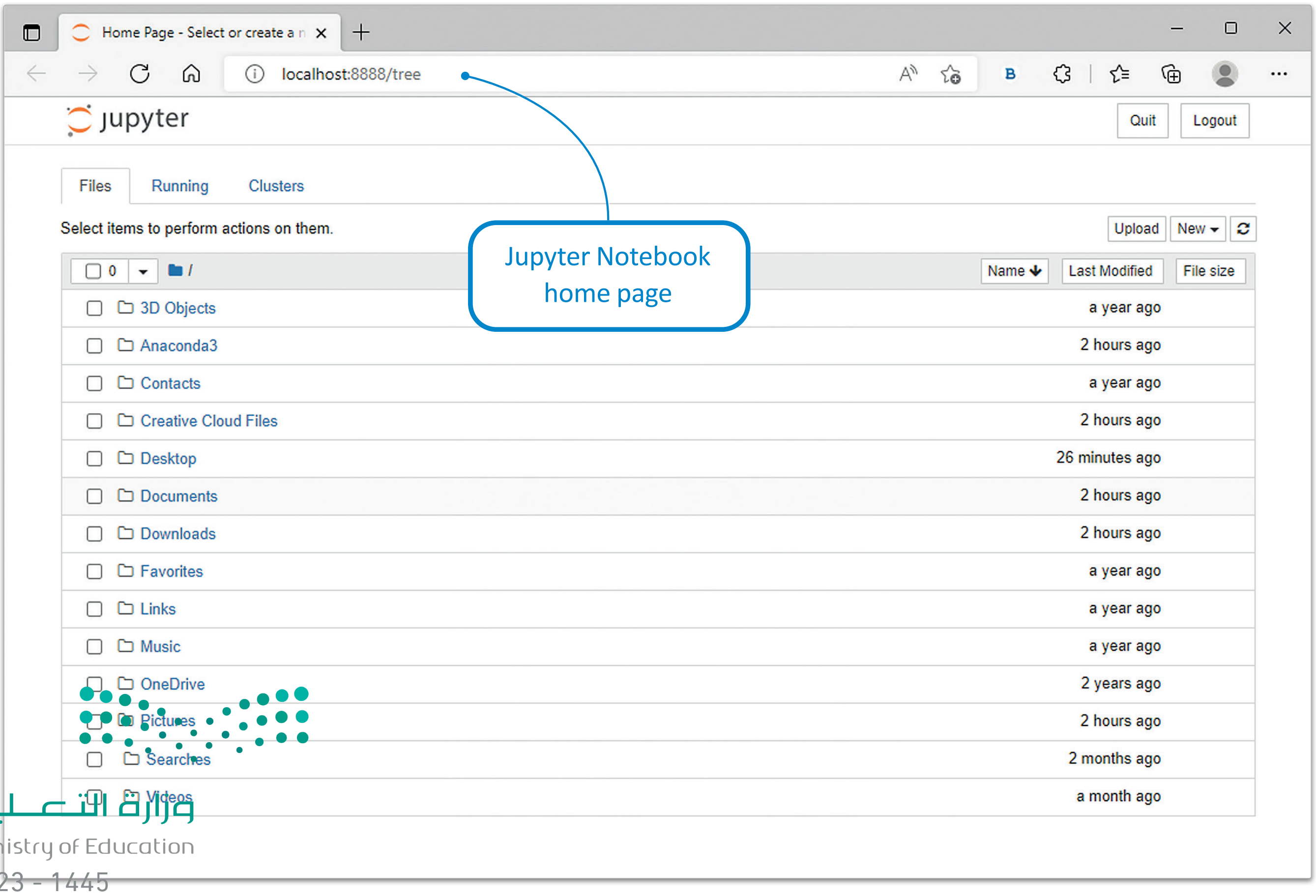
To open Jupyter Notebook:

> Click **Start** 1, click **Anaconda3**. 2

> Select **Jupyter Notebook**. 3

> The Jupyter Notebook home page opens in the browser.


Adobe Acrobat DC
New
Alarms & Clock
Anaconda3 (64-bit)
New
Anaconda Navigator (Anaconda3)
New
Anaconda Powershell Prompt (Anac...
New
Anaconda Prompt (Anaconda3)
New
Jupyter Notebook (Anaconda3)
New
Reset Spyder Settings (Anaconda3)
New
Spyder (Anaconda3)
New
1
2
3



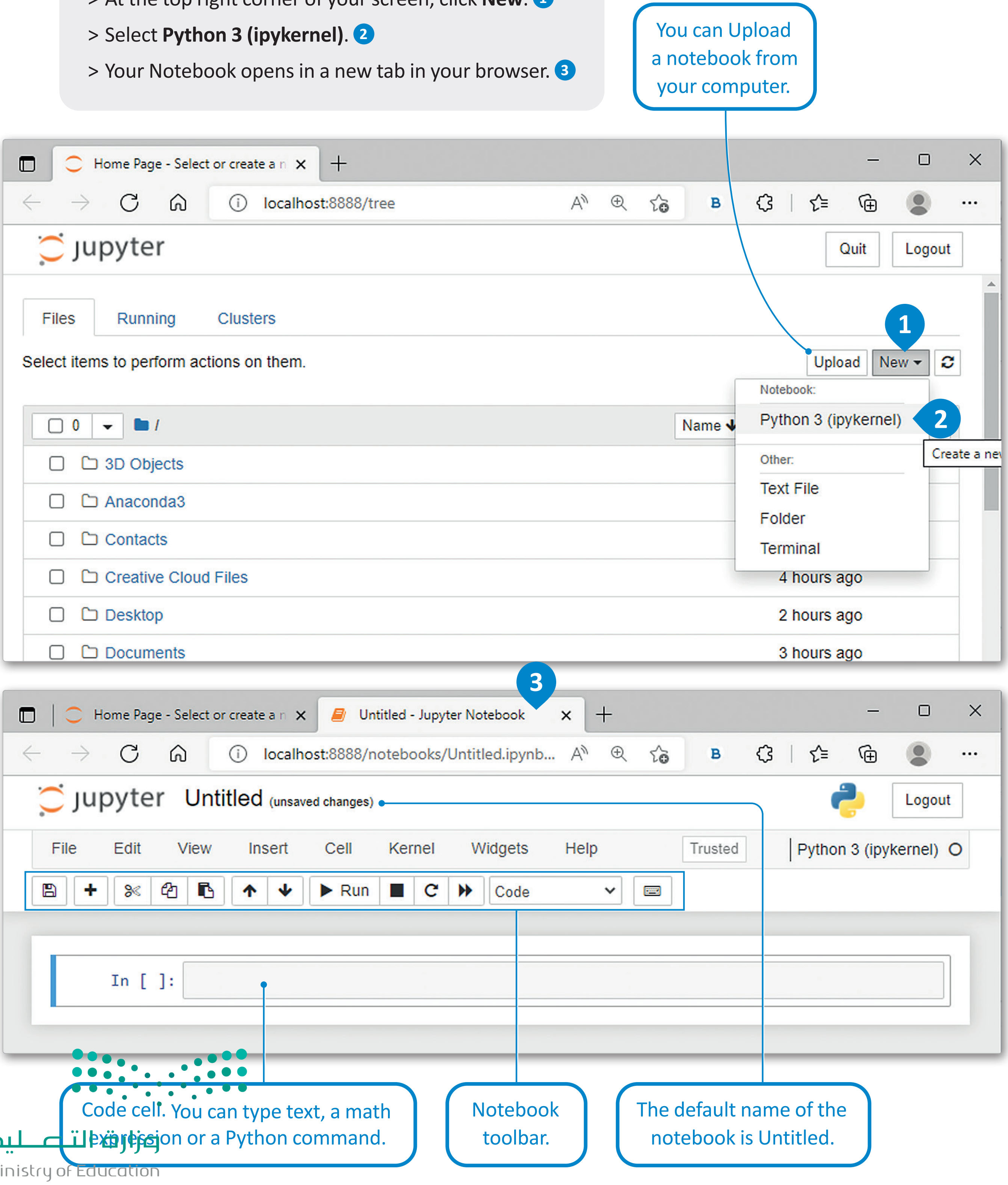
Select items to perform actions on them.

Name	Last Modified
3D Objects	a year ago
Anaconda3	2 hours ago
Contacts	a year ago
Creative Cloud Files	2 hours ago
Desktop	26 minutes ago
Documents	2 hours ago
Downloads	2 hours ago
Favorites	a year ago
Links	a year ago
Music	a year ago
OneDrive	2 years ago
Pictures	2 hours ago
Searches	2 months ago
Videos	a month ago

Figure 1.16: Jupyter Notebook's home page

To create a new Jupyter Notebook:

> At the top right corner of your screen, click **New**. 1
> Select **Python 3 (ipykernel)**. 2
> Your Notebook opens in a new tab in your browser. 3



You can Upload a notebook from your computer.
Home Page - Select or create a n
localhost:8888/tree
jupyter
Quit
Logout
Files
Running
Clusters
Select items to perform actions on them.
Upload
New
1
Notebook:
Python 3 (ipykernel)
2
Create a ne
Other:
Text File
Folder
Terminal
Name
3D Objects
Anaconda3
Contacts
Creative Cloud Files 4 hours ago
Desktop 2 hours ago
Documents 3 hours ago
3
Untitled - Jupyter Notebook
localhost:8888/notebooks/Untitled.ipynb...
Untitled (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [ ]:
Code cell. You can type text, a math expression or a Python command.
Notebook toolbar.
The default name of the notebook is Untitled.

Figure 1.17: Create a new Jupyter Notebook

Now that your notebook is ready, it's time to write and run your first program in Jupyter Notebook.

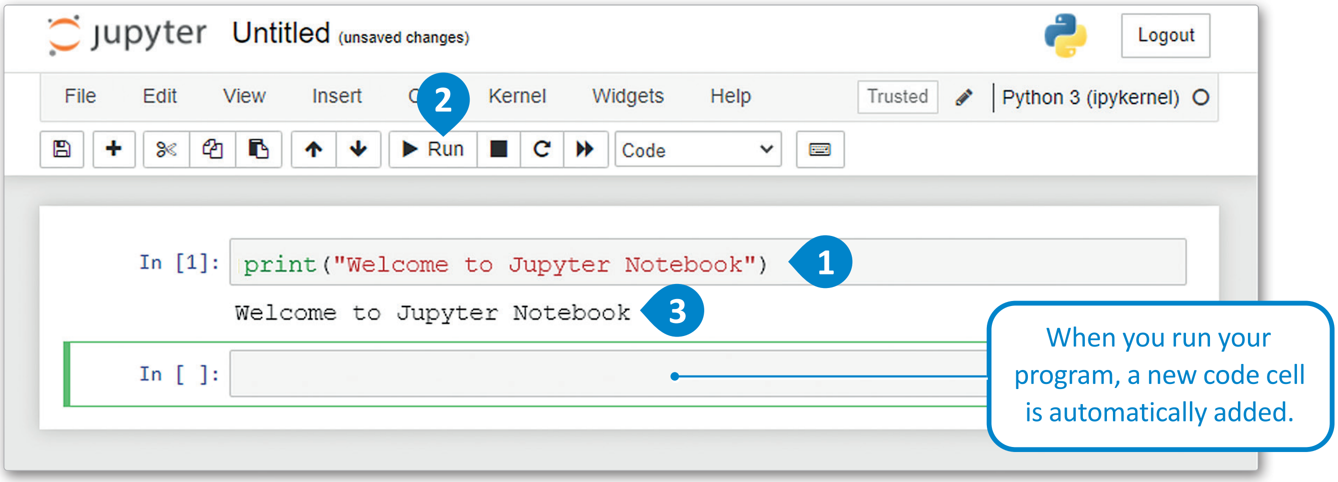
You can have as many different cells as you need in the same Notebook. Each cell contains its own code.

To create a program in Jupyter Notebook:

> Type the commands inside the code cell. 1

> Click the **Run** button. 2

> The result is displayed under the commands. 3


jupyter Untitled (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
1
2
3
When you run your program, a new code cell is automatically added.

```
In [1]: print("Welcome to Jupyter Notebook")

        Welcome to Jupyter Notebook

In [ ]:
```

Figure 1.18: Create a program in Jupyter Notebook

You can run your program by pressing Shift + Enter ↵.

It's time to save your Notebook.

To save your Notebook:

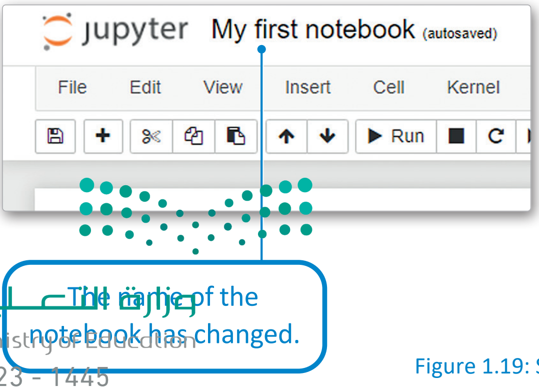
> Click **File**. 1

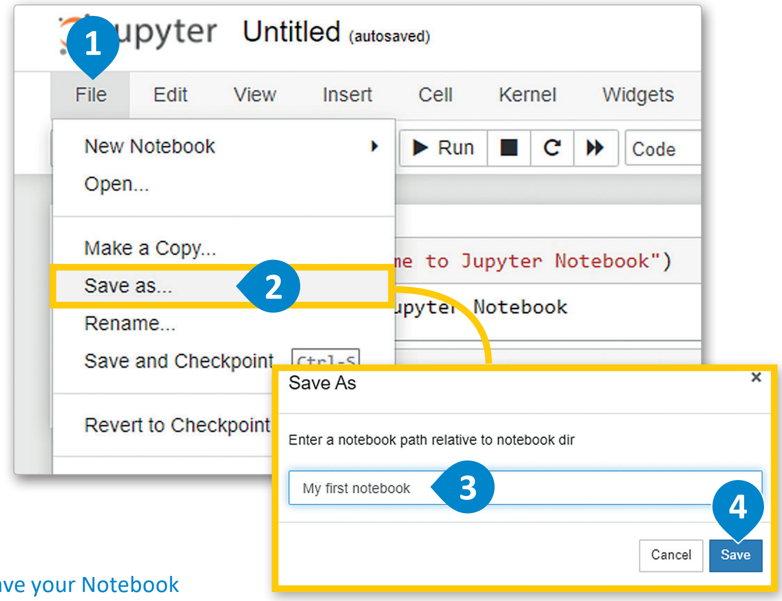
> Select **Save as**. 2

> Type a name for your Notebook. 3

> Press **Save**. 4

When you are working, the Notebook is autosaved.


jupyter My first notebook (autosaved)
File Edit View Insert Cell Kernel
Run
The name of the notebook has changed.


jupyter Untitled (autosaved)
File Edit View Insert Cell Kernel Widgets
New Notebook
Open...
Make a Copy...
Save as...
Rename...
Save and Checkpoint Ctrl-S
Revert to Checkpoint
Run
Code
Save As
Enter a notebook path relative to notebook dir
My first notebook
Cancel
Save
1
2
3
4

```
me to Jupyter Notebook")
upyter Notebook
```

Figure 1.19: Save your Notebook

Let's see the example of figure 1.15 in Jupyter.

1. Create a stack to store a set of numbers (1, 21, 32, 45).
2. Use the pop operation twice to remove the last two elements from the stack.
3. Use the push operation to add a new element to the stack.

```
myStack=[1,21,32,45]
print("Initial stack: ", myStack)
print(myStack.pop())
print(myStack.pop())
print("The new stack after pop: ", myStack)
myStack.append(78)
print("The new stack after push: ", myStack)
```

The print(myStack.pop()) function is used to display the value returned by the myStack.Pop() function.

```
Initial stack: [1, 21, 32, 45]
45
32
The new stack after pop: [1, 21]
The new stack after push: [1, 21, 78]
```

```
myStack=[1,21,32,45]
print("Initial stack:", myStack)
a=len(myStack)
print("size of stack",a)
# empty the stack
for i in range(a):
   myStack.pop()
print(myStack)
myStack.pop()
```

The len function returns the length of the stack.

This statement is used to delete all elements of the stack.

```
Initial stack: [1, 21, 32, 45]
size of stack 4
[]
--------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [3], in <cell line: 9>()
      7     myStack.pop()
      8 print(myStack)
----> 9 myStack.pop()

IndexError: pop from empty list
```

The error appeared because the stack is empty and you typed a command to delete an element from the empty stack

IndexError

You will notice that an error appears. You typed a command to delete an element from the empty stack and this caused underflow to the stack. You should always check that there are elements in the stack before trying to delete an element from it.

In the following program, you will create a stack and you will add or remove elements from it. The program displays a menu which asks you about the action you want to do each time.

- To add an element to the stack, you have to press the number 1 in the program menu.
- To remove an element from the stack, you have to press the number 2 in the program menu.
- To exit the program, you have to press the number 3 in the program menu.

```
def push(stack,element):
    stack.append(element)
def pop(stack):
    return stack.pop()
def isEmpty(stack):
    return len(stack)==0
def createStack():
    return []

newStack=createStack()
while True:
    print("The stack so far is:",newStack)
    print("-----------------------------")
    print("Choose 1 for push")
    print("Choose 2 for pop")
    print("Choose 3 for end")
    print("-----------------------------")
    choice=int(input("Enter your choice: "))
    while choice!=1 and choice!=2 and choice!=3:
        print ("Error")
        choice=int(input("Enter your choice: "))
    if choice==1:
        x=int(input("Enter element for push: "))
        push(newStack,x)
    elif choice==2:
        if not isEmpty(newStack):
            print("The pop element is:",pop(newStack))
        else:
            print("The stack is empty")
    else:
        print("End of program")
        break;
```

Execute the previous program as follows:

- Create a stack of three numbers, and
- Add elements to the stack.

```
The stack so far is: []
-----------------------------
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 1
Enter element for push: 26
The stack so far is: [26]
-----------------------------
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 1
Enter element for push: 18
The stack so far is: [26, 18]
-----------------------------
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 1
Enter element for push: 23
The stack so far is: [26, 18, 23]
-----------------------------
```

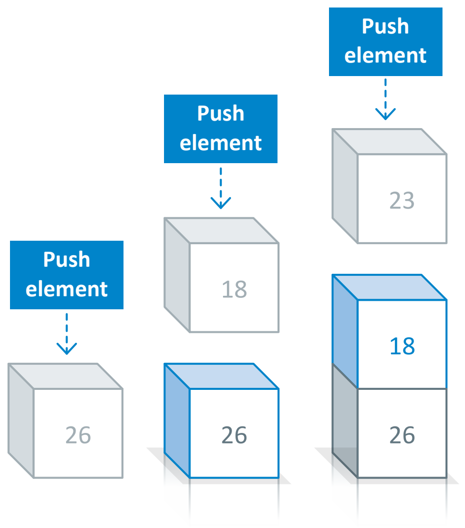


Figure 1.20: Pushing elements

Now, you will remove two elements from the stack and then exit the program.

```
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 2
The pop element is: 23
The stack so far is: [26, 18]
-----------------------------
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 2
The pop element is: 18
The stack so far is: [26]
-----------------------------
Choose 1 for push
Choose 2 for pop
Choose 3 for end
-----------------------------
Enter your choice: 3
End of program
```

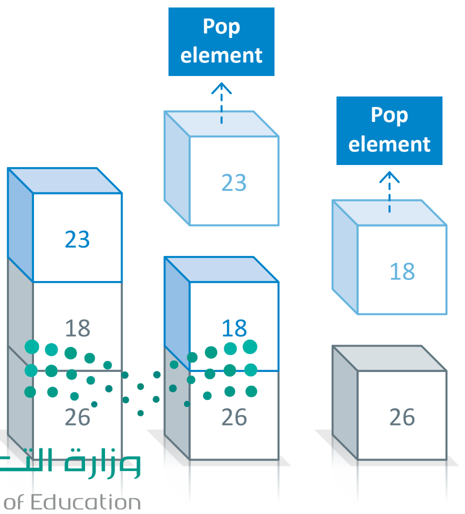


Figure 1.21: Popping elements

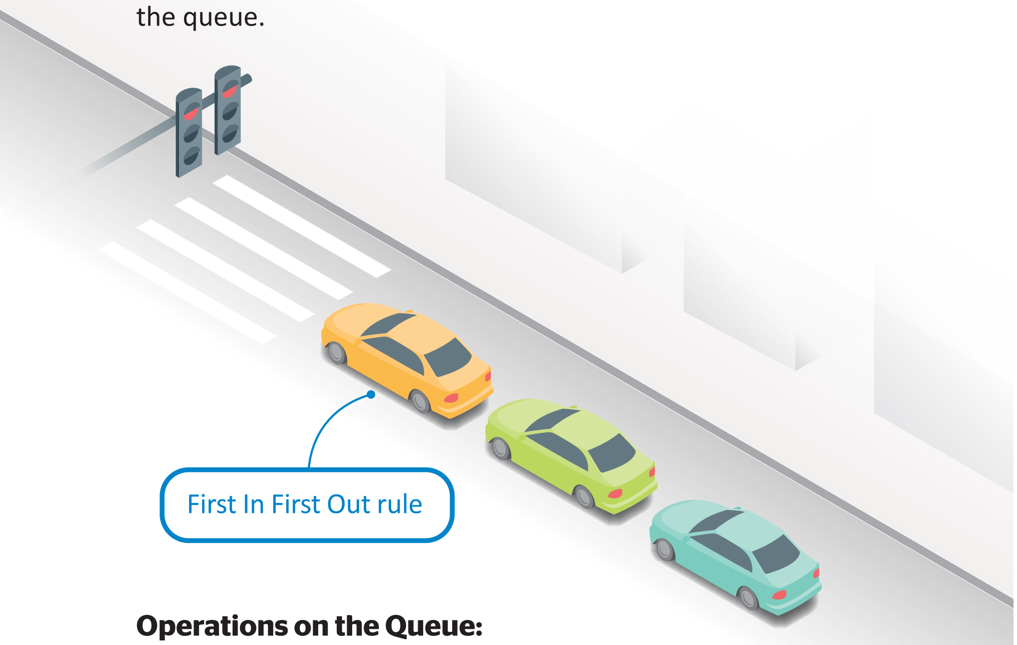
Queue

The next data structure you are going to explore is the queue. We often come across queues in our everyday life. The most common queue is the queue of cars waiting at a traffic light. When the traffic light turns green, the car that entered first in the queue will be the one which exits first. A queue is a data structure that follows the First In First Out (FIFO) rule, meaning that each element in the queue is served in the order it reaches the queue.

First In First Out (FIFO) rule

The first element added in the list is processed first and the newest element is processed last.

The difference between the stack and the queue is that in the stack, the addition and the deletion of an element are done from the same side. In the queue, the addition is done on one side and the deletion is done on the other side. So, in the stack, when deleting, the last added element is deleted, while in the queue, the first added element is deleted.



Operations on the Queue:

There are two main operations on the queue:

- **Enqueue**: This operation is used to add an element to the rear of the queue.
- **Dequeue**: This operation is used to remove an element from the front of the queue.

Pointer

The pointer is a variable which stores or points to the address of another variable. The pointer is like a page number in the index of a book that drives the reader to the required content.

Queue Pointers

The queue has two pointers:

- Front pointer: Points to the first element of the queue.
- Rear: points to the last element of the queue.

Index

Index is a number that describes the position of an element in a data structure.

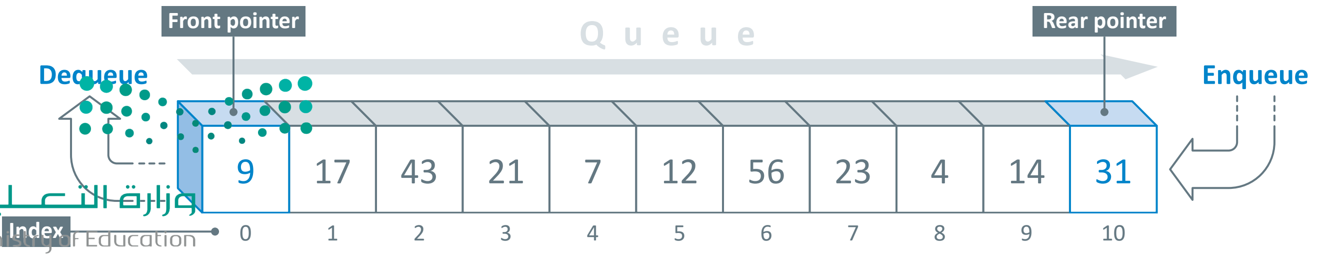


Figure 1.22: Operations on the Queue

Enqueue Operation

The operation of adding a new element in the queue is called Enqueue. To insert a new element into the queue:

- the value of the rear pointer is increased by one and points to the position of the new element to be entered.
- the element is inserted.

You cannot add or remove an element from the middle of the queue.

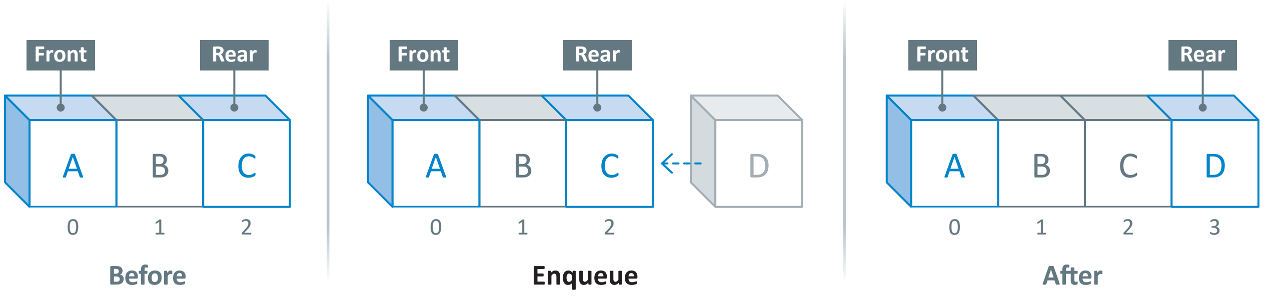


Figure 1.23: Enqueue operation

Dequeue Operation

The operation of removing an element of a queue is called dequeue. To remove an element from the queue:

- the element indicated by the front pointer is removed.
- the value of the front pointer is increased by one to point to the next available element of the queue.

Before any action you must check if there is free space in the queue to add a new element and if there is at least one element for export.

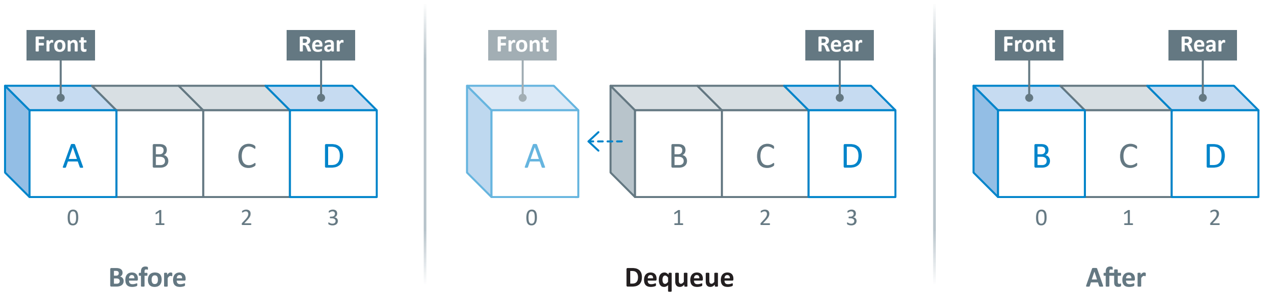


Figure 1.24: Dequeue operation

Queue in Python

In Python, the queue can be represented in several ways, including lists. This is due to the fact that a list represents a group of linear elements and also to the possibility of adding an element at the end of the list and the possibility of deleting an element from its beginning.

Below you will learn the general formulas for some of the operations that can be performed on a queue:

Table 1.3: Queue methods

Method	Description
listName.append(x)	Enqueue the element x to the list representing the queue.
listName.pop(0)	Dequeue the first element from the list.

The listName.pop() method can be used for both stack and queue data structures. When it is used with a stack, the method has no arguments. When it is used with a queue, the method needs a zero to be added in the arguments: listName.pop(0). The difference between the two functions is presented in table 1.4 below.

Table 1.4: listName.pop() vs listName.pop(0) method

Method	Description
listName.pop()	If the function argument is empty, the last element is removed from the end of the list that represents the stack.
listName.pop(0)	If the function argument is zero, the first element of the list representing the queue is removed.

Let's see an example of the implementation of a queue in Python.

- Create a queue to store the set of numbers (1, 21, 32, 45).
- Use the dequeue operation twice to remove the first two elements from the queue.
- Use the enqueue operation to add a new element to the queue.

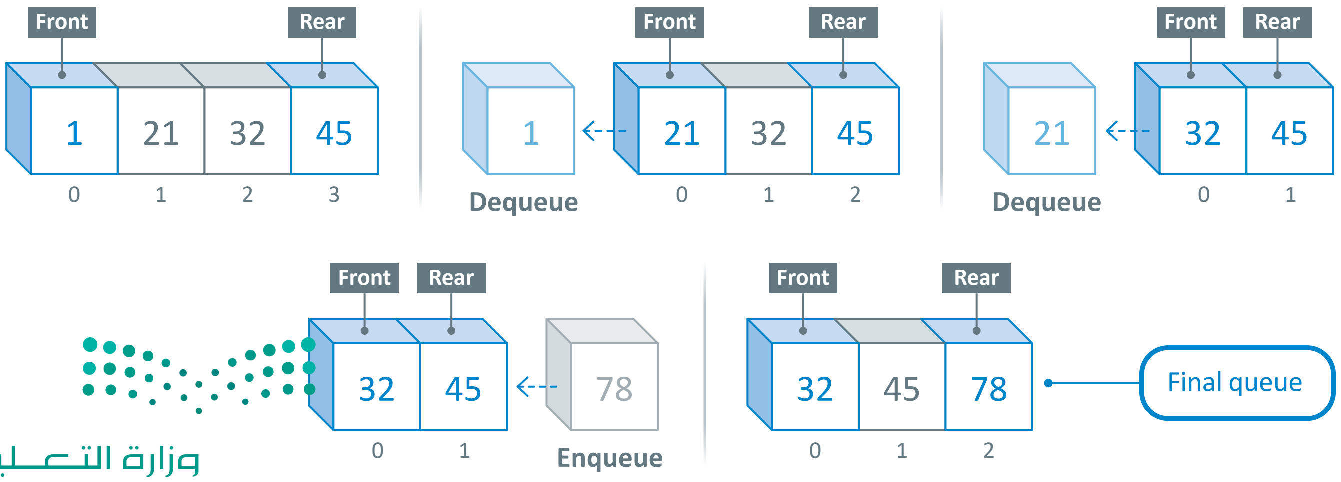


Figure 1.25: Queue graphical example

To program the above steps in Python, you will use a Python list to implement the queue structure, as you did with the stack.

```
myQueue=[1,21,32,45]
print("Initial queue: ", myQueue)
myQueue.pop(0)
myQueue.pop(0)
print("The new queue after pop: ", myQueue)
myQueue.append(78)
print("The new queue after push: ", myQueue)
```

```
Initial queue: [1, 21, 32, 45]
The new queue after pop: [32, 45]
The new queue after push: [32, 45, 78]
```

Let's see what happens if you try to remove an element from an empty queue. First you have to empty the queue.

```
myQueue=[1,21,32,45]
print("Initial queue: ", myQueue)
a=len(myQueue)
print("size of queue ",a)
# empty the queue
for i in range(a):
   myQueue.pop(0)
print(myQueue)
myQueue.pop(0)
```

```
Initial queue: [1, 21, 32, 45]
size of queue 4
[]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [6], in <cell line: 9>()
      7     myQueue.pop()
      8 print(myQueue)
----> 9 myQueue.pop()

IndexError: pop from empty list
```

You should always check that there are elements in the queue before trying to delete an element.

The error appeared because you tried to delete an element from an empty queue.

Queue Applications

One example of a queue in Computer Science is the printing queue. For example, you have a computer lab with 30 computers connected to one printer. When students want to print, their print jobs create a queue. The tasks are queued to be processed using a First In First Out (FIFO) method. Tasks will be printed in the chronological order they were submitted. The task that was submitted first will be printed before the one that was submitted after. The task at the end of the queue will not be printed until all tasks before it have been printed. When the printer completes a job it will look in the queue to see if there are any jobs left to process.

Stack and Queue Using Queue Module

A list in Python can act as a queue and a stack as well. Python offers the **Queue module** which is another way to implement these two data structures. The Queue module includes some ready-to-use functions that can be used with both stack and queue.

Table 1.5: Queue module methods

Methods	Description
queueName=queue.Queue()	Creates a new queue named queueName.
queueName.put(x)	Adds the element x to the queue.
queueName.qsize()	Returns the size of the queue.
queueName.get()	Gets and removes the first element from the queue and the last element from the stack.
queueName.full()	Returns True if the queue is full and False if the queue is empty. Can be applied to the stack as well.
queueName.empty()	Returns True if the queue is empty and False if the queue is full. Can be applied to the stack as well.

The methods of the Queue library can be used with both the stack and the queue.

You will use the Queue module to create a queue. In this example you should:

- Import the queue library to use the queue's methods.
- Create an empty queue named myQueue.
- Add the elements a, b, c, d, e to myQueue queue.
- Print queue elements.

```
from queue import *

myQueue = Queue()
# add the elements in the queue
myQueue.put("a")
myQueue.put("b")
myQueue.put("c")
myQueue.put("d")
myQueue.put("e")

# print the elements of the queue
for element in list(myQueue.queue):
   print(element)
```

You import the Queue module at the beginning of your code.

```
a
b
c
d
e
```

وزارة التعليم
Ministry of Education
2023 - 1445

Create a queue in which five values are entered by the user during program execution, and then print these values and finally print the size of the queue.

```python
from queue import *

myQueue = Queue()

# the user enters the elements of the queue for i in range(5):
for i in range(5):
   element=input("enter queue element: ")
   myQueue.put(element)

# print the elements of the queue
for element in list(myQueue.queue):
   print(element)

print ("Queue size is: ",myQueue.qsize())
```

```
enter queue element: 5
enter queue element: f
enter queue element: 12
enter queue element: b
enter queue element: 23
5
f
12
b
23
Queue size is: 5
```

Create a program to check if the queue is empty or full.

```python
from queue import *

myQueue = Queue()

myQueue.put("a")
myQueue.put("b")
myQueue.put("c")
myQueue.put("d")
myQueue.put("e")

checkFull=myQueue.full()
print("Is the queue full? ", checkFull)
checkEmpty= myQueue.empty()
print("Is the queue empty? ", checkEmpty)
```

```
Is the queue full? False
Is the queue empty? False
```

As mentioned before, the Queue module includes some ready-to-use methods that can be used with a stack or a queue. The table 1.6 shows the module methods that can be used with the stack data structure.

Table 1.6: Queue module methods used for the stack

Method	Description
stackName=queue.LifoQueue()	Creates a new stack named stackName.
stackName.get()	Pops the last element from the stack.

Let's use the Queue module to create an empty stack.

```
from queue import *

myStack = LifoQueue()

myStack.put("a")
myStack.put("b")
myStack.put("c")
myStack.put("d")
myStack.put("e")

for i in range(5):
   k=myStack.get()
   print(k)

# empty the stack
checkEmpty= myStack.empty()
print("Is the stack empty?", checkEmpty)
```

Remember that operations in the stack operate according to the LIFO rule.

When using the get function with a queue, the fetching and printing operations will be based on the FIFO rule.

```
e
d
c
b
a
Is the stack empty? True
```

Example: Print

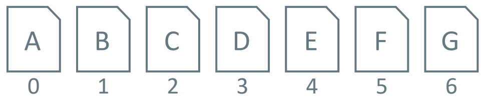
In the following example, you will see a simulation of the printer's print queue. When users send print jobs, they are added to the print queue. The printer uses this queue to find out what file to print next.

- Suppose the capacity of the printer is only 7 files, but at the same time you need to print 10 files from file A to file J.
- Write a program that captures the print queue from the start of the first print job A until all print jobs are completed.
- Add the block that confirms that the print job queue is empty.

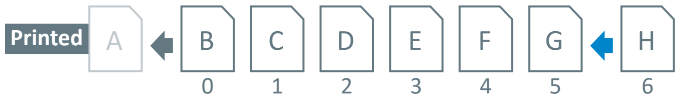
You can use the following algorithm:

1 Create a print job queue

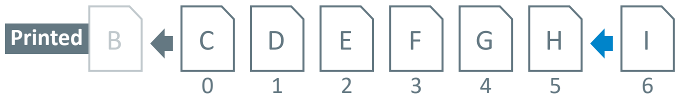
2 Insert files from A to G into the print job queue



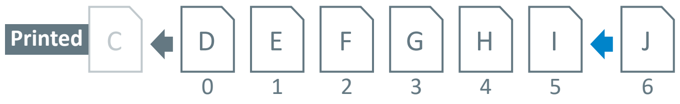
3 Output file A and insert file H



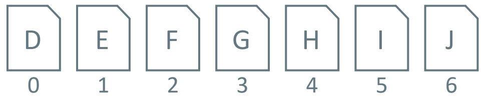
4 Output file B and insert file I



5 Output file C and insert file J



6 Output files that have been printed (D-E-F-G-H-I-J) one by one.



```python
# import the queue library
from queue import *
# import the time library to use the sleep function
import time
# initialize the variables and the queue
printDocument = " "
printQueueSize = 0
printQueueMaxSize = 7
printQueue = Queue(printQueueMaxSize)
# add a document to print the queue
def addDocument(document):
    printQueueSize = printQueue.qsize()
    if printQueueSize == printQueueMaxSize:
        print("!! ", document, " was not sent to print queue.")
        print("The print queue is full.")
        print()
        return
    printQueue.put(document)
    time.sleep(0.5) #Wait 5.0 seconds
    print(document, " sent to print queue.")
    printQueueSizeMessage()
# print a document from the print queue
def printDocument():
    printQueueSize = printQueue.qsize()
    if printQueueSize == 0:
        print("!! The print queue is empty.")
        print()
        return
    printDocument = printQueue.get()
    time.sleep(1) # wait one second
    print ("OK - ", printDocument, " is printed.")
    printQueueSizeMessage()
# print a message with the size of the queue
def printQueueSizeMessage():
    printQueueSize = printQueue.qsize()
    if printQueueSize == 0:
        print ("There are no documents waiting for printing.")
    elif printQueueSize == 1:
        print ("There is 1 document waiting for printing.")
    else:
        print ("There are ", printQueueSize, " documents waiting for printing.")
    print()
# the main program
# send documents to the print queue for printing
addDocument("Document A")
addDocument("Document B")
addDocument("Document C")
addDocument("Document D")
addDocument("Document E")
addDocument("Document F")
addDocument("Document G")
printDocument()
addDocument("Document H")
printDocument()
addDocument("Document I")
printDocument()
addDocument("Document J")
addDocument("Document K")
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
```

```
Document A sent to print queue.
There is 1 document waiting for printing.

Document B sent to print queue.
There are 2 documents waiting for printing.

Document C sent to print queue.
There are 3 documents waiting for printing.
```

```
Document D sent to print queue.
There are 4 documents waiting for printing.

Document E sent to print queue.
There are 5 documents waiting for printing.

Document F sent to print queue.
There are 6 documents waiting for printing.

Document G sent to print queue.
There are 7 documents waiting for printing.

OK - Document A is printed.
There are 6 documents waiting for printing.

Document H sent to print queue.
There are 7 documents waiting for printing.

OK - Document B is printed.
There are 6 documents waiting for printing.

Document I sent to print queue.
There are 7 documents waiting for printing.

OK - Document C is printed.
There are 6 documents waiting for printing.

Document J sent to print queue.
There are 7 documents waiting for printing.

!! Document K was not sent to print queue.
The print queue is full.

OK - Document D is printed.
There are 6 documents waiting for printing.

OK - Document E is printed.
There are 5 documents waiting for printing.

OK - Document F is printed.
There are 4 documents waiting for printing.

OK - Document G is printed.
There are 3 documents waiting for printing.

OK - Document H is printed.
There are 2 documents waiting for printing.

OK - Document I is printed.
There is 1 document waiting for printing.

OK - Document J is printed.
There are no documents waiting for printing.

!! The print queue is empty.
```

Static and Dynamic Data Structures

As mentioned before, a data structure is a way to efficiently store and organize data. You also learned about the classification of data structures into primitive and non-primitive.

Data structures can also be classified into **Static** and **Dynamic**.

Static Data Structure

In a static data structure, the size of the structure is fixed. The elements of the data are allocated to contiguous memory location. The most representative example of a static data structure is the Array.

Dynamic Data Structure

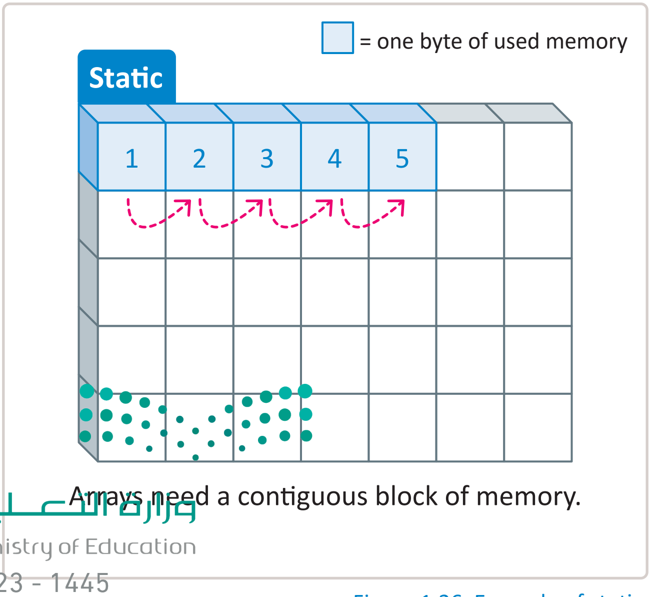
In a dynamic data structure, the size is not fixed and it can be modified during the execution of the program, depending on the operations performed on it. The dynamic data structures are designed to facilitate the change in size of the data structures during run time. The most representative example of a dynamic data structure is the Linked List.

Table 1.7: Static Data Structures vs Dynamic Data Structures

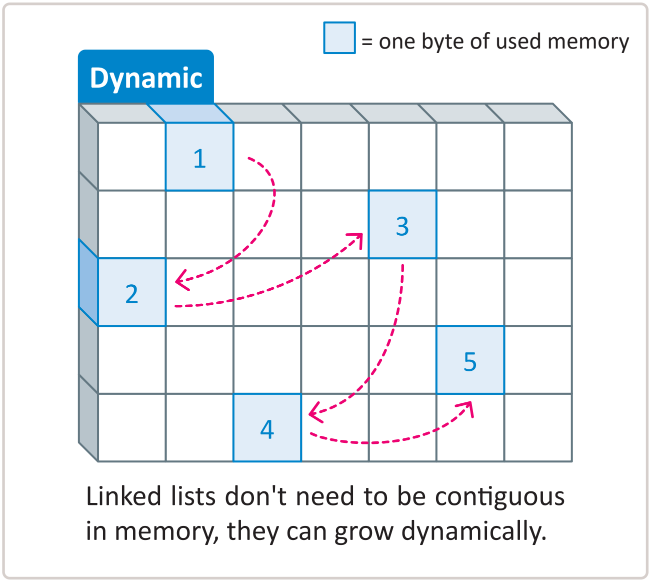
	Static	Dynamic
Memory size	Fixed memory size.	The size can be changed during run time.
Types of memory storage	The elements are stored in contiguous locations in the main memory.	The elements are stored in random places in the main memory.
Data access speed	Faster to access.	Slower to access.

Memory Allocation

Linked lists belong to dynamic data structures. This means that the nodes of the linked list are not stored in contiguous memory locations like the data of arrays. This is the reason you need to store the pointer from one node to another.



Arrays need a contiguous block of memory.



Linked lists don't need to be contiguous in memory, they can grow dynamically.

Figure 1.26: Example of static and dynamic memory allocations.

Linked List

A Linked List is a linear data structure, and it is one of the most popular data structures in programming. A linked list is like a chain of nodes. Each node contains two fields: the data field where the data are stored and a field containing the pointer to the next node. This excludes the last node in which the address field does not carry any data.

One of the advantages of the linked list is that it can dynamically increase or decrease its size by adding or deleting nodes.

Linked List

A Linked List is a linear data structure, which is like a chain of nodes.

Node

Node is an individual block of a data structure which contains data and one or more links to other nodes

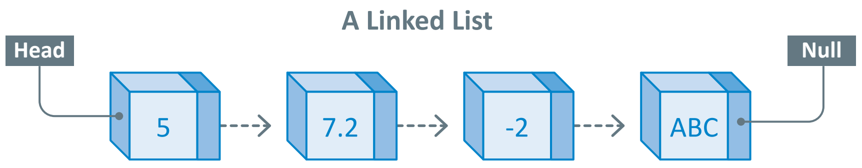


Figure 1.27: Graphical representation of a linked list

Node

Each node in the linked list consists of two parts.

- The first section contains the data.
- The second part contains a pointer pointing to the next node.

To read the content of a specific node, you must pass through all previous nodes.

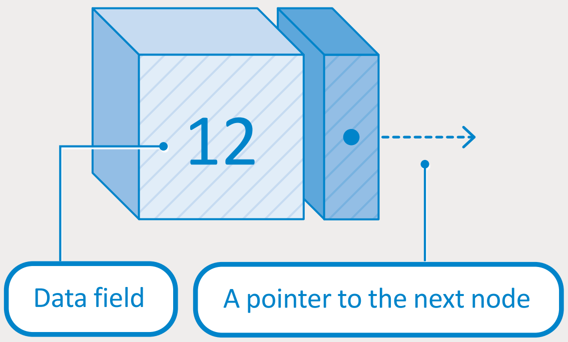


Figure 1.28: Graphical representation of node

Here you can see an example of an integer linked list.
The linked list consists of five nodes.

Null means having no value, not defined or empty. Although sometimes we use the number 0 to symbolize null, 0 is a specific number and can be a real value.

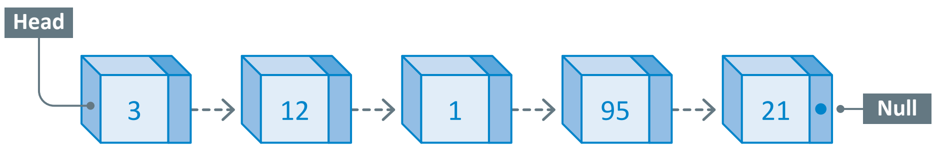


Figure 1.29: Graphical representation of an integer linked list

The nodes in a list do not have names. What you know about the node is the address (memory location) at which it is stored. To access any node of the list, you only need to know the address of the first node. Then you follow the chain of nodes to reach the node you need.

For example, if you want to access the third node of the list, to process the data it contains, you have to start from the first node of the list. From the first node to access the second, and from the second to reach the third.

- The address of the first node is stored in a special (independent) variable that is usually called the Head.
- The pointer of the last node of the list is null and is represented with the symbol ●.
- When the list is empty, the head pointer points to a null value.

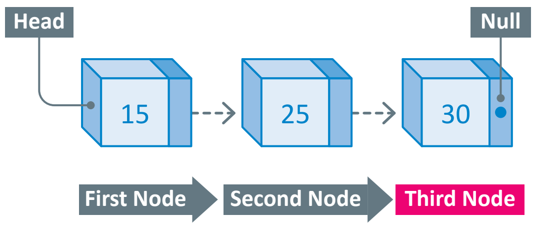


Figure 1.30: Access the third node of the linked list

Let's see a graphic example of a linked list in figure 1.31. As mentioned earlier, each node consists of data and a pointer pointing to the next node. Each node is also stored in memory at a specific address.

Specific node example:

- The node data is number 15.
- The address of the node in memory is 10.
- The next node will be at address 20.

Let's connect the previous node with the next node with data value 42, which in turn points to the third and last node at address 30 with data value 37.

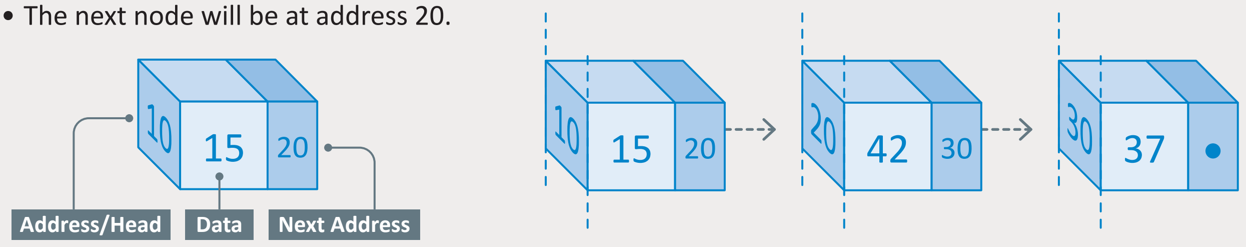


Figure 1.31: Pointers in a linked list

Table 1.8: Differences between list and linked list

Differences	List	Linked List
Memory storage method	Contiguous locations in the memory.	Random locations in the memory.
The structure	Each element can be accessed by the index number.	Its elements can be accessed through the pointer.
Size	Each element is stored one after the other.	Objects are stored as nodes containing the data and the address of the next element.
Memory usage	Only data is stored in memory.	Data and pointers are stored in memory.
Data access type	Random access to any list element.	Sequential access to elements.
The speed of addition and removal	Adding and removing elements is slower.	Faster addition and removal.

Linked List in Python

Python does not provide a predefined data type for linked lists. You have to create your own data type or use additional python libraries that provide a representation of this data type. To create a linked list, you can use Python classes. In the following example, figure 1.32, you will create a linked list with three nodes each containing a day of the week.

Class

A class is a user defined data structure that holds its own data members (properties) and methods (behavior). Classes are used as a template for creating objects.

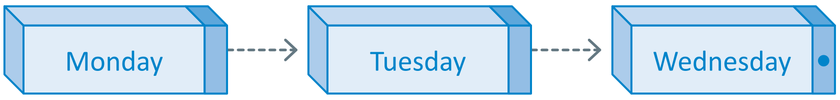


Figure 1.32: Linked list example

You will first create a node using Class.

```
# single node
class Node:
    def __init__(self, data, next=None):
        self.data = data # node data
        self.next = next # Pointer to the next node

# Create a single node
first = Node("Monday")
print(first.data)
```

```
Monday
```

The next step is to create a single-node linked list, this time you will use the head pointer to point to the first node.

```
# single node
class Node:
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next

# linked list with one head node
class LinkedList:
  def __init__(self):
    self.head = None

# list linked with a single node
Linkedlist1 = LinkedList()
Linkedlist1.head = Node("Monday")
print(Linkedlist1.head.data)
```

```
Monday
```

Now add more nodes to your linked list.

```
# single node
class Node:
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next

# an empty linked list with a head node.
class LinkedList:
  def __init__(self):
    self.head = None

# the main program
linked_list = LinkedList()
# the first node
linked_list.head = Node("Monday")
# the second node
linked_list.head.next = Node("Tuesday")
# the third node
linked_list.head.next.next = Node("Wednesday")

# print the linked list
node = linked_list.head
while node:
    print (node.data)
    node = node.next
```

The while statement is used to move from one node to another.

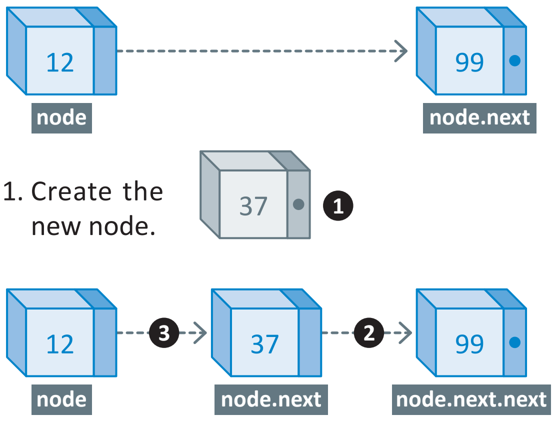
```
Monday
Tuesday
Wednesday
```

Add a Node to a Linked List

The actions required to add the new node are:

- The pointer of the first node must point to the address of the new node, so that the new node becomes the second node.
- The pointer of the new (second) node must point to the address of the third node.

In this way, you do not have to shift the elements if a new element is added in the middle. The process is limited to changing the address values in the node, that makes the addition faster in the case of linked lists compared to the case of sequential lists.



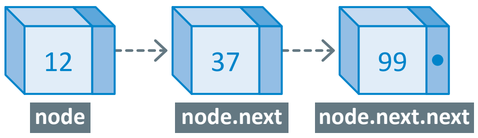
2. Link the **37** node to the **99** node.
3. Link the **12** node to the **37** (new node created).

Example:

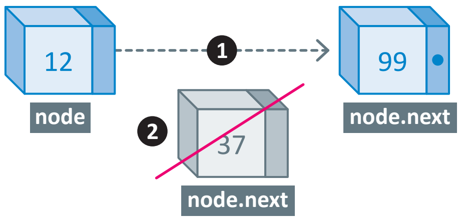
You have a linked list of two elements: 12, 99. You want to insert the element 37 as a second element. In the end, you will have a list of three elements: 12, 37, 99.

```
# single node
class Node:
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next

# linked list with one head node
class LinkedList:
  def __init__(self):
    self.head = None

def insertAfter(new, prev):
    # create the new node
    new_node = Node(new)
    # make the next of the new node the same as the next of the previous node
    new_node.next = prev.next
    # make the next of the previous node the new node
    prev.next = new_node

# create the linked list
L_list = LinkedList()

# add the first two nodes
L_list.head = Node(12)
second = Node(99)
L_list.head.next = second

# insert the new node after node 12 (the head of the list)
insertAfter(37, L_list.head)

# print the linked list
node = L_list.head
while node:
    print (node.data)
    node = node.next
```

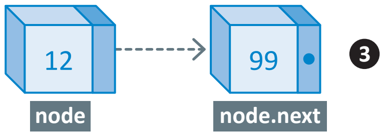
```
12
37
99
```



1. Link the **12** node pointer to **99** node.
2. Delete node **37**



3. Final result



Delete a Node from a Linked List

To delete a node, you must change the pointer of its predecessor to point to the node following the deleted node. The deleted (second) node is "useless data" and the memory space it occupies is allocated for other uses.

Example:

You have a linked list of three elements: 12, 37, 99. You want to delete the element 37. In the end, you will have a list of two elements: 12, 99.

```python
# single node
class Node:
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next

# linked list with one head node
class LinkedList:
  def __init__(self):
    self.head = None

def deleteNode(key, follow):

    # store the head node
    temp = follow.head

    # find the key to be deleted,
    # the trace of the previous node to be changed
    while(temp is not None):
        if temp.data == key:
            break
        prev = temp
        temp = temp.next

    # unlink the node from the linked list
    prev.next = temp.next
    temp = None

# create the linked list
L_list = LinkedList()

# add the first three nodes
L_list.head = Node(12)
second = Node(37)
third = Node(99)
L_list.head.next = second
second.next = third

# delete node 37
deleteNode(37,L_list)

# print the linked list
node = L_list.head
while node:
    print (node.data)
    node = node.next
```

If you want to delete the first node of a linked list, you must move the head to the second node of the list.

```
12
99
```

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. Python defines non-Primitive Data Structures.	○	○
2. Linear Data Structures store data items exclusively in random order.	○	○
3. Adding and removing items in a linked list is slower than a list.	○	○
4. The items in a list can only be accessed through their index number.	○	○
5. The size of a static data structure can be modified during the execution of a program.	○	○

2 State the differences between static and dynamic data structures.

Static data structures	Dynamic data structures

3 Write two examples of uses for linked lists.

4 You have a stack with six empty spaces.

- You will add the following letters C, E, B, A and D in positions 0 to 4.
- Fill the stack indicating the position of the top cursor.
- Execute the following operations:

pop → **pop** → **push X** → **push K** → **pop** →

Show the final output after performing the above operations, indicating the position of the top cursor.

Write a program that creates the stack shown above, and then perform the above operations using the standard queue library.

Stack	
5	
4	
3	
2	
1	
0	

Final Output	
5	
4	
3	
2	
1	
0	

5 You have the following number sequence: 4, 8, 2, 5, 9, 13.

- What is the process used to add the above elements into the queue?

- Complete the queue after adding the elements.

0	1	2	3	4	5

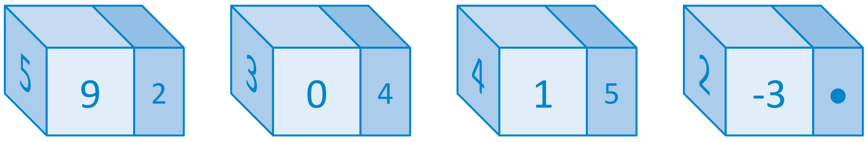
- What is the process used to remove elements from the queue?

- How many times should the above operation be performed to remove the element with value 5?

- Write Python code to create the previous queue.

6 Given the following nodes, draw the linked list and then write the values in the list in the correct order.

Head = 3



7 Create a list with the following numbers: 5, 20, 45, 8, 1.

- Draw the nodes of the linked list.
- Describe the process of adding the number 7 after the number 45.
- Draw the new list.
- Describe the process required to delete the second node of the list.
- Draw the final linked list.

وزارة التعليم
Ministry of Education
2023 - 1445

Lesson 3

Non-Linear Data Structures

Link to digital lesson

www.ien.edu.sa

In the previous lessons, you learned about some linear data structures. In linear data structures, each element follows the other element in a linear manner. Can you think of any case in which things do not proceed linearly? For example, can an element be followed by more than one element?

Non-Linear Data Structures

A data structure can be characterized by the possibility of linking an element to more than one other element at the same time. An element of a non-linear data structure could be connected to more than one element. Representive examples of non-linear data structures are trees and graphs. Figure 1.33 illustrates the linear and the non-linear data structures.

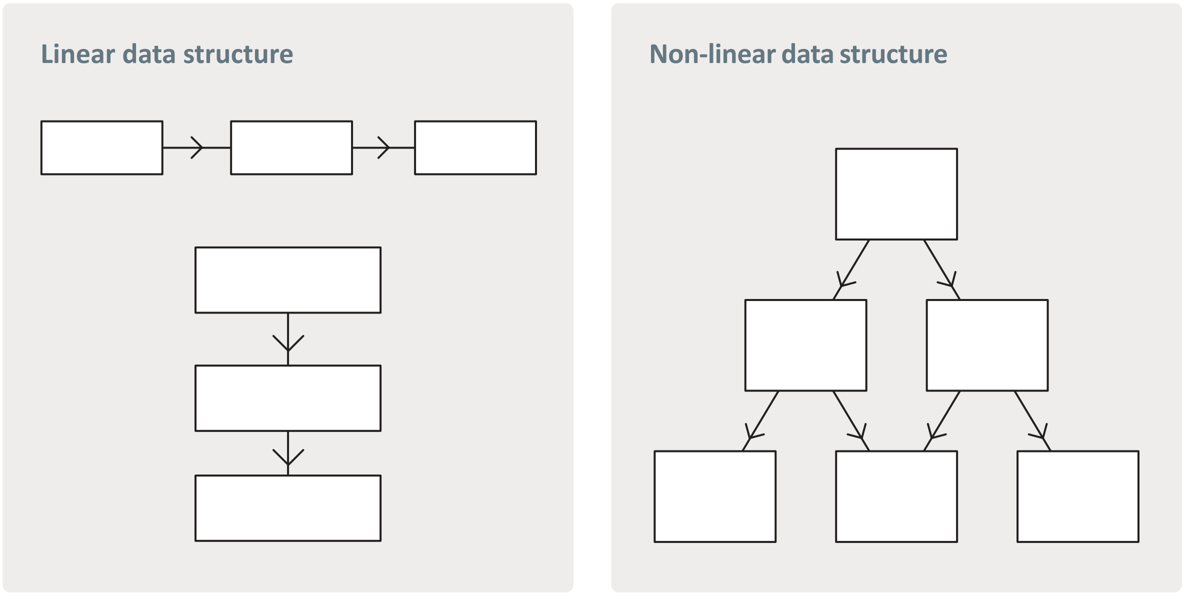


Figure 1.33: Graphical representation of linear and non-linear data structures

Table 1.9: Differences between Linear and Non-linear Data Structures

Linear	Non-linear
Data elements are arranged in a linear order where each element is attached to its previous and its next elements.	Data elements can be attached to many other elements.
Data elements can be traversed in a single pass.	Data elements cannot be traversed in a single pass.
The implementation is easier.	The implementation is more complicated.

Trees

Trees are non-linear data structures. A tree consists of a collection of nodes that are arranged in a hierarchical order. Each node can associate with one or more nodes. Nodes are connected with edges in a form of parent-child relationship. Trees are used in many areas of Computer Science, including operating systems, graphics, database systems, games, AI, and computer networks.

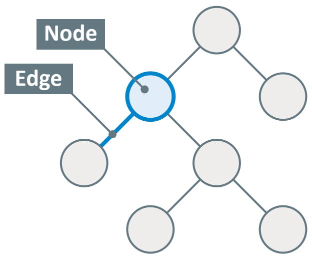


Figure 1.34: Relationships of a tree

Tree Terminology Used in the Tree Data Structure

Root: The first and only node in the tree that does not have a parent and is at the first level of the tree. (A node in figure 1.35)

Child: A node directly connected to a node at a higher level. (Node H is the child of node D and nodes B and C are the children of node A)

Parent: A node that has one or more children at a lower level. (Node B is the parent of nodes D and E)

Leaf: A node that does not have any child nodes. (Node F is a leaf node)

Siblings: All child nodes that have the same parent node. (Nodes D and E are siblings)

Edges: The links that connect tree nodes.

Sub-Tree: Smaller trees that can be found within a larger tree. (A tree with node D as a parent and node H and I as children)

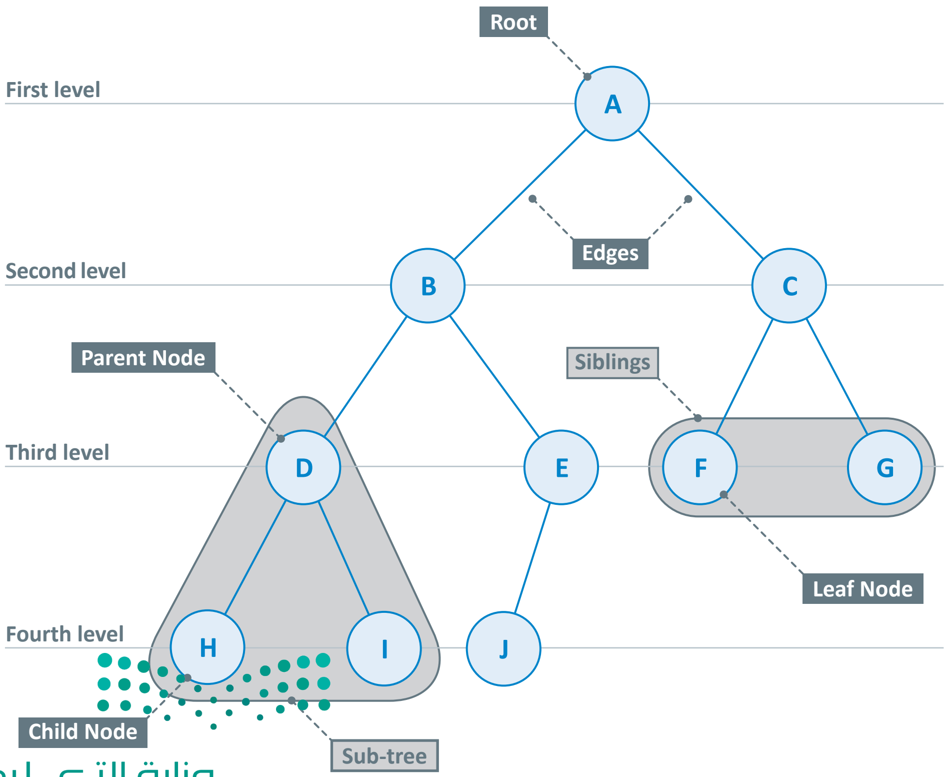


Figure 1.35: Tree data structure

Tree

A tree is a non-linear data structure. It is a collection of nodes arranged hierarchically.

Edge

An Edge connects nodes in a tree data structure.

You can have a simple tree, which consists of a single node. This node is also the root of this simple tree, because it has no parent.

Here is an example of a tree data structure.

A node can be both a child and a parent: a child of the previous node and a parent of the next node.

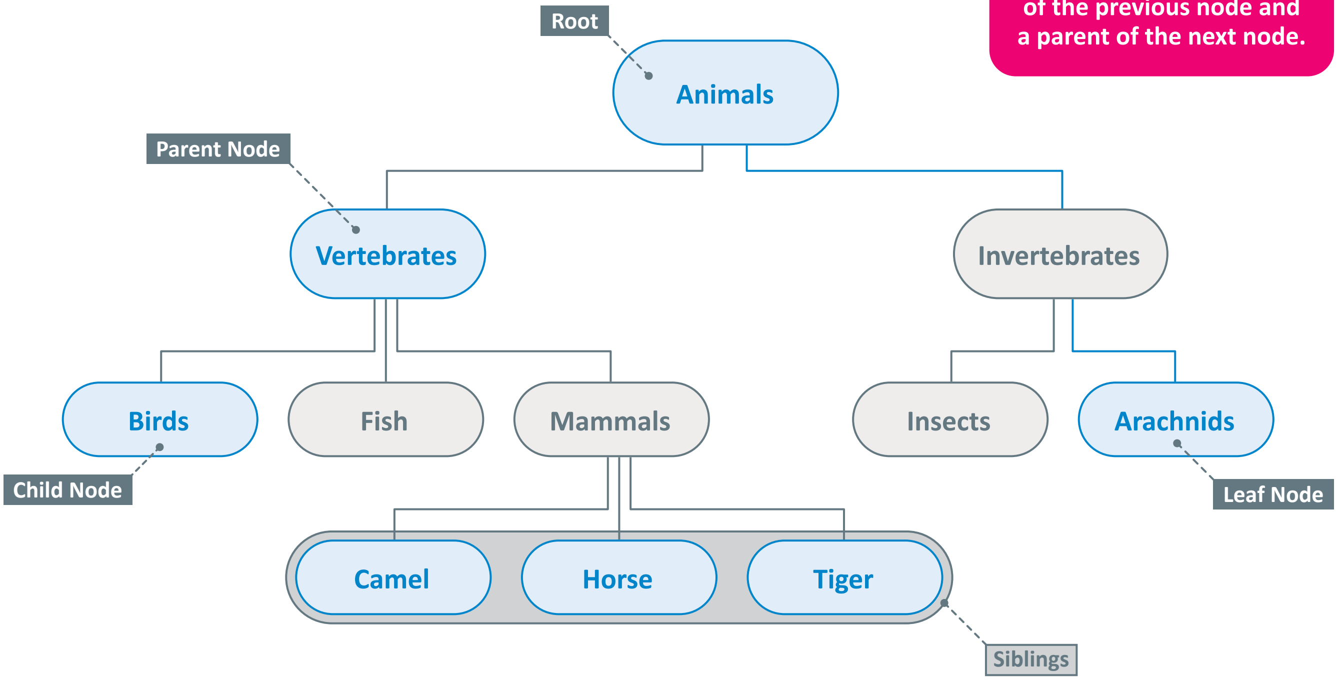


Figure 1.36: Example of a tree data structure

Tree Data Structure Features

- They are used to represent a hierarchy.
- They are flexible, it is very easy to add or remove an element from a tree.
- It is easy to search for an element in a tree.
- They reflect structural relationships between the data.

Example

The organization of files in the operating system is a practical example of a tree. As you can see in figure 1.37, inside the "Documents" folder is another folder called "Python Projects" which contains two other files.

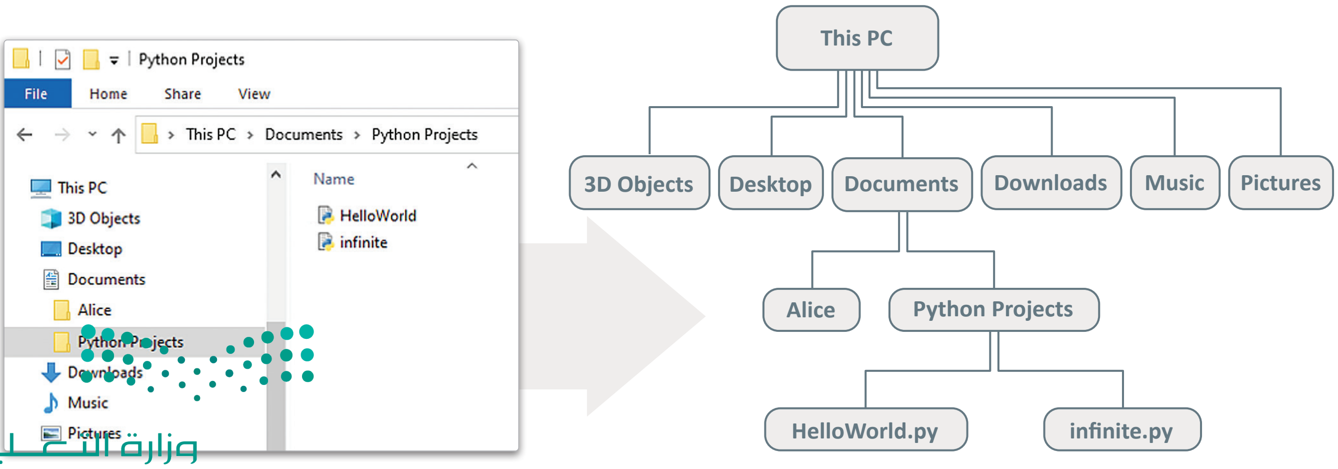


Figure 1.37: Organization of files in the operating system

Tree Data Structure in Python

Python does not provide a predefined data type for the tree data structure. However, trees are easily built out of lists and dictionaries. A very simple implementation of a tree using a dictionary is shown in figure 1.38.

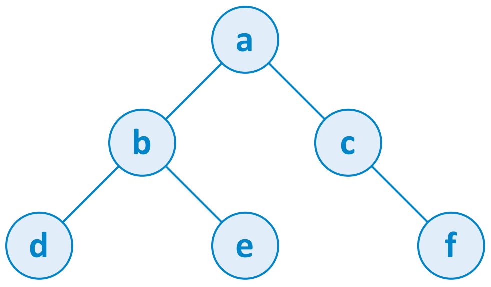


Figure 1.38: Python dictionary tree

In this example, you will create a tree using a Python dictionary. The keys of the dictionary are the nodes of the tree. For each key, the corresponding value is a list containing the nodes that are connected by a direct edge from this node.

```
myTree = {
    "a": ["b", "c"], # node
    "b": ["d", "e"],
    "c": [None, "f"],
    "d": [None, None],
    "e": [None, None],
    "f": [None, None],
}
print(myTree)
```

```
{'a': ['b', 'c'], 'b': ['d', 'e'], 'c': [None, 'f'],
'd': [None, None], 'e': [None, None], 'f': [None, None]}
```

In the following example, you will create a tree like the one in figure 1.39.

Parent

Children

```
myTree = {"Data Structures":["Linear","Non-linear"],
          "Linear":["Stack","Queue","Linked List"],
          "Non-linear":["Tree", "Graph"]}

for parent in myTree:
    print(parent, "has",len(myTree[parent]),"nodes" )
    for children in myTree[parent]:
        print(" ",children)
```

```
Data structures has 2 nodes
  Linear
  Non-linear
Linear has 3 nodes
  Stack
  Queue
  Linked List
Non-linear has 2 nodes
  Tree
  Graph
```

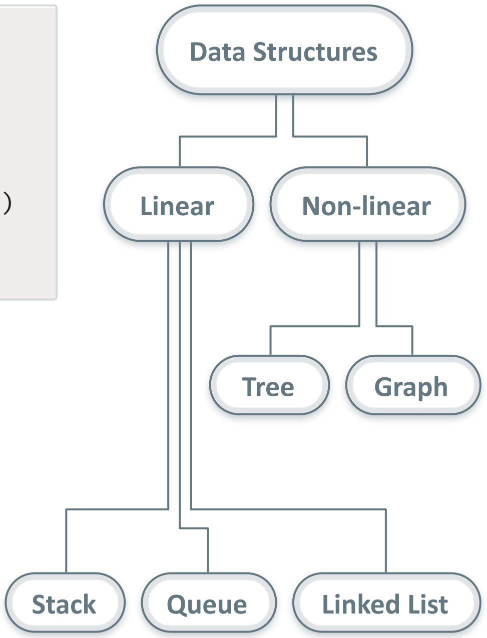
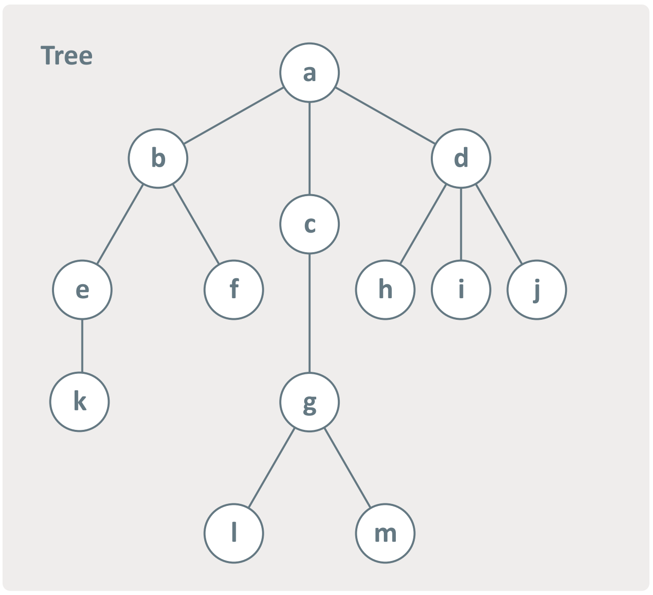


Figure 1.39: Data structures tree

Binary Tree

There is a special category of trees called binary trees. A binary tree is a tree where each node has two children at most, called Right Child and Left Child. In figure 1.40 you can see an example of a tree and a binary tree.



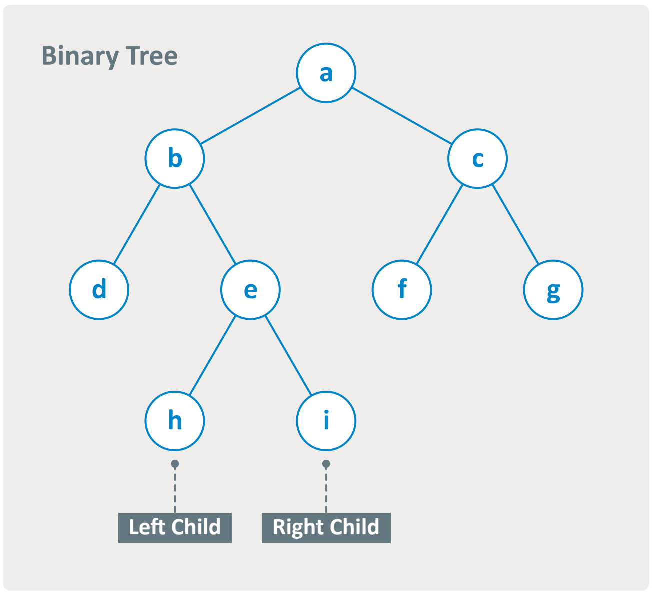
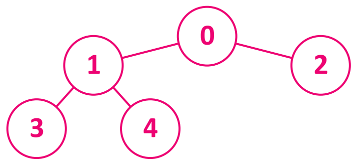
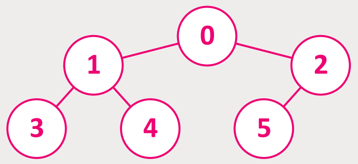
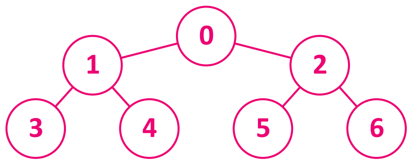


Figure 1.40: Tree and Binary tree

Table 1.10: Types of binary tree data structures

Type	Description	Structure drawing
Full binary tree	Each node, other than "leaves", has either 0 or 2 "children".	
Complete Binary Tree	Every level of the tree is fully filled, except for possibly the last level. All the nodes in the last level are filled from left to right.	
Perfect Binary Tree	All internal nodes have two children and all leaves are at the same level.	

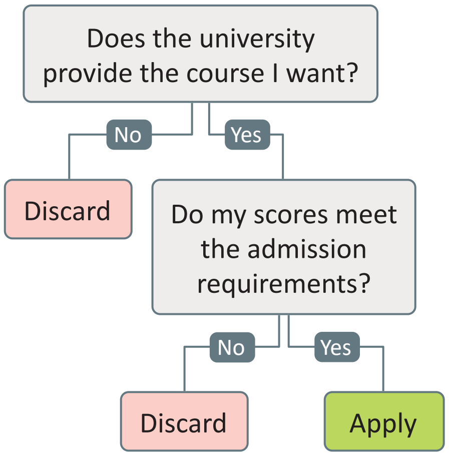
Examples of Applications of Tree Data Structures:

- Store hierarchical data, such as folder structures.
- Define data in HTML.
- Implementation of indexing in databases.

Decision Tree

The decision statement (if a: else b) is one of the most frequently used statements in Python. By nesting and combining these code statements, you can build a decision tree.

Decision trees are used in AI through a machine learning technique, called decision tree learning. The end nodes of the trees in this technique, also known as leaves, contain possible solutions to a problem. Each node, with the exception of the leaves, is associated with a logical condition from which the possibilities of answering yes or no branch out. Decision trees are easy to understand, use, visualize, and verify. For example, figure 1.41 shows a decision tree that determines whether to apply for a certain university or not based on two criteria: courses provided by the university and meeting admission requirements.



Does the university provide the course I want?
No
Yes
Discard
Do my scores meet the admission requirements?
No
Yes
Discard
Apply

Figure 1.41: Decision tree example

Graphs

The most important feature of non-linear data structures is that like arrays and linked lists, their data does not follow a sequence, and its elements can each be associated with more than one element. A rooted tree starts with a root node, which can be connected to other nodes. Trees follow certain rules: the tree nodes must be connected, and the tree must be free of loops and self loops.

But what will happen if you don't follow the rules of trees? Then, you are not talking about trees but about a new dynamic data structure called Graphs. In fact, trees are actually a type of graph. The graph is the most general data structure, in the sense that all previous structures presented can be considered special cases of graphs. figure 1.42 illustrates a graph with six nodes and ten edges.

Graph

A graph is a data structure that consists of a set of nodes and a set of lines, that connect all or some of the nodes.

A tree is a graph but the reverse is not true as not all graphs are trees.

Table 1.11: Differences between Trees and Graphs

Trees	Graphs
Nodes attached in hierarchical model.	Nodes attached in network model.
There is a unique node called the root (in rooted trees).	There is not a unique or root node.
Nodes are connected with a parent-child relationship.	There is no parent-child relationship between the nodes.
Trees are simpler structures compared to graphs.	Graphs are more complex structures.
Cycles are not allowed.	Can contain cycles.

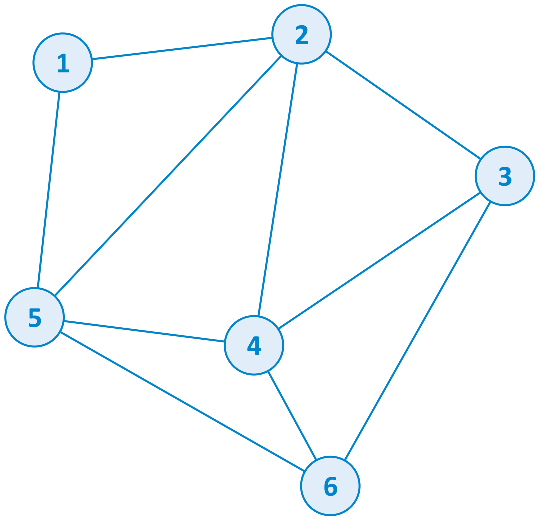


Figure 1.42: An example graph with six nodes and ten edges

Types of Graphs

- **Directed Graph**: In a directed graph, nodes are connected by directed edges – they only go in one direction.
- **Undirected graphs**: In undirected graphs, the connections have no direction. This means that the edges indicate a two-way relationship where each edge can be traversed in both directions.

Figure 1.43 shows a simple directed and undirected graph with six nodes and six edges.

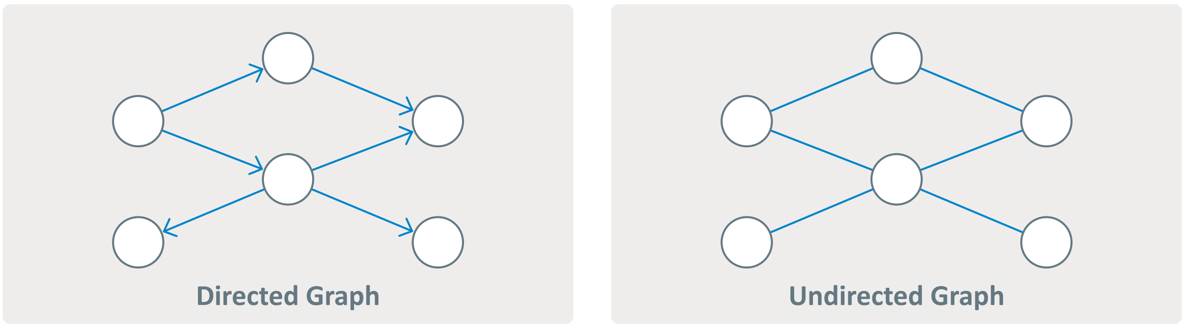


Figure 1.43: Directed and Undirected Graph

Graphs in Everyday Life

World Wide Web

The most representative example of graphs is the World Wide Web. The World Wide Web can be considered as a directed graph, in which the vertices represent web pages, and the directed edges the hyperlinks. Web structure mining is the discovery of useful knowledge from the structure of the World Wide Web represented though hyperlinks. A person can form a graph structure of such hyperlinks and the relationships they create between different web pages. You can see a graphical representation of the World Wide Web in figure 1.44. By accessing such graphs, someone can calculate the relative importance of a web page.

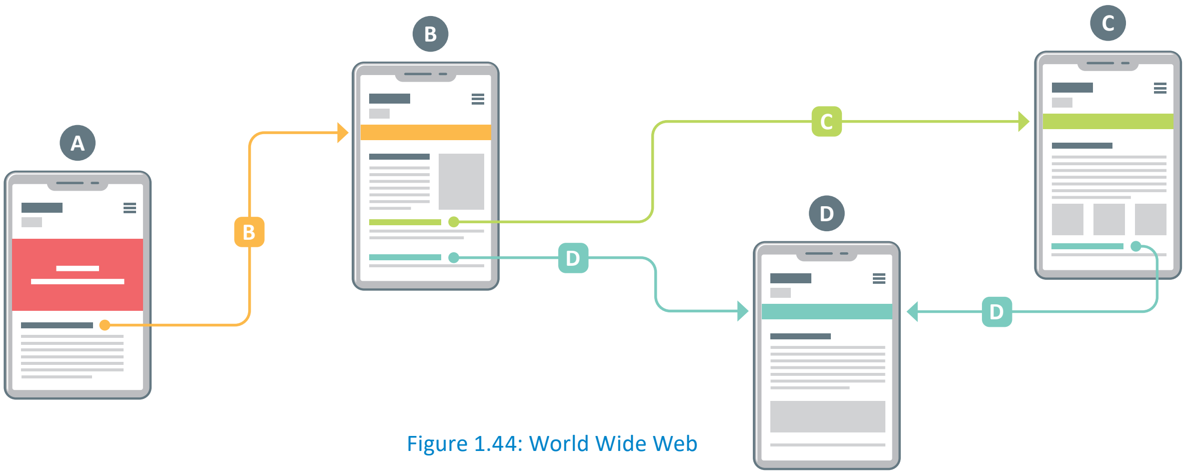


Figure 1.44: World Wide Web

The Google search engine uses a similar approach to find the relative importance of a web page and orders the search result according to this importance. This algorithm used by Google is known as the PageRank algorithm, and was invented by the Google founders.

Facebook

Facebook is another example of an undirected graph. As you can see in figure 1.46, the nodes represent Facebook users, while the edges represent friendship relationships. When you want to add a friend, he has to accept your request; that person cannot be your friend on the network without accepting your friend request. The relationship here between two users (two nodes) is a two-way relationship. The Facebook's friend suggestion algorithm uses graph theory. Social network analysis studies social relationships using graphs or network theory from Computer Science.

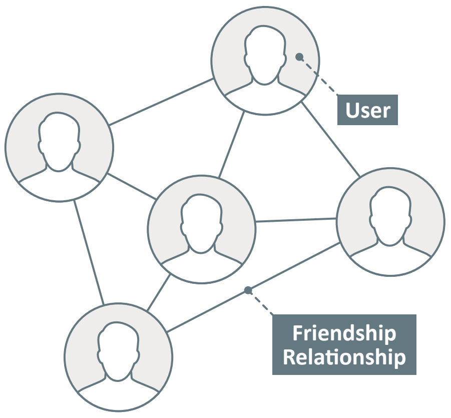


Figure 1.46: Facebook's undirected graph

Google Maps

Google maps and all other similar applications use graphs for showing transportation systems to calculate the shortest path between two locations. These applications use graphs that contain a very large number of nodes and edges that cannot be distinguished by the human eye.

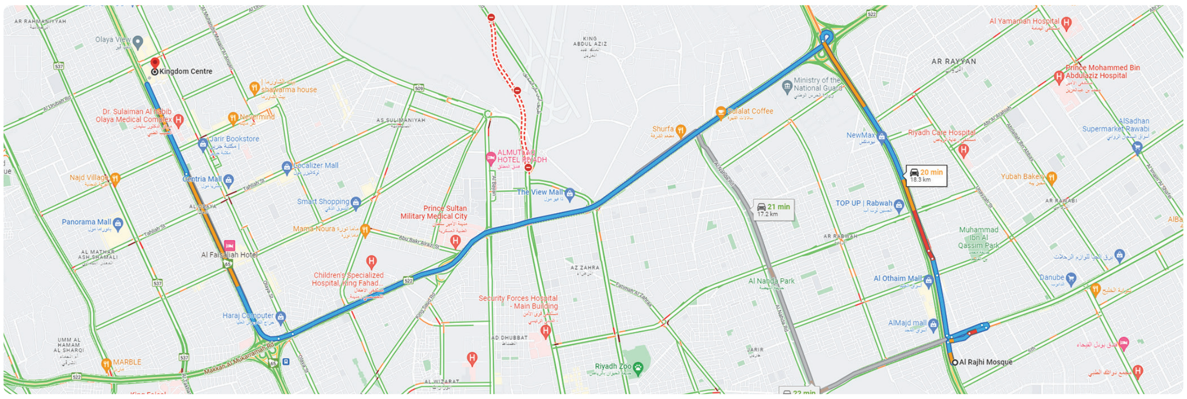


Figure 1.45: Google maps

Neural Network

A neural network is a machine learning graph that imitates the human brain. A neural network can be directed and undirected based on the learning objective. It consists of neurons and weights, distributed in different layers. The neurons are represented by nodes and the weights are represented by edges. Signal flows are computed and optimized throughout the neural network structure to minimize the error. It is used in many intelligent applications such as machine translation, image classification, object detection, and object recognition. Figure 1.47 shows an example of a neural network structure.

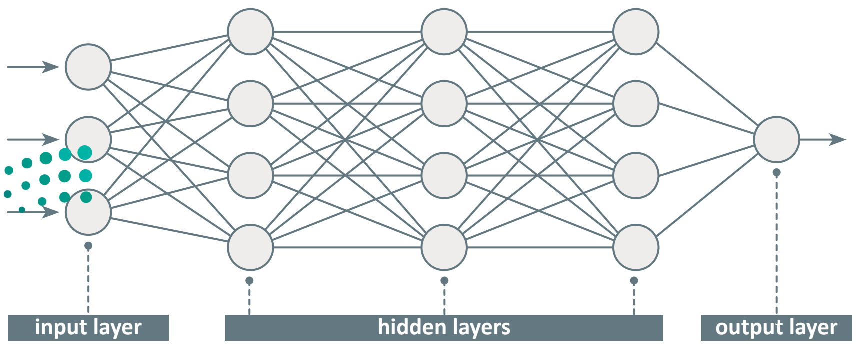


Figure 1.47: Neural Network Structure

Graphs in Python

Since Python does not provide a predefined data type for trees, it does not provide a predefined data type for graphs (remember that trees are a special case of graphs). However, graphs can also be built out of lists and dictionaries.

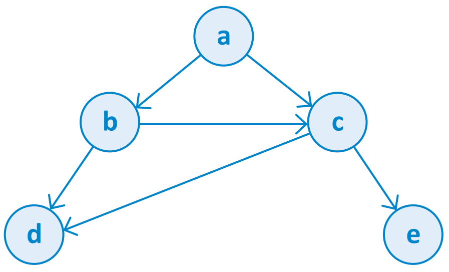


Figure 1.48: Graph example

In the following example, you will do the following:

1. Create a directed graph like the one shown in figure 1.48.
2. Create a function to add a node to the graph.
3. Create a function containing all paths of the graph.

```
myGraph = { "a" : ["b","c"],
            "b" : ["c", "d"],
            "c" : ["d", "e"],
            "d" : [],
            "e" : [],
            }
print(myGraph)
```

```
{'a': ['b', 'c'], 'b': ['c', 'd'], 'c': ['d', 'e'],
 'd': [], 'e': []}
```

Then the main program will:

1. Create the graph.
2. Print the graph.
3. Call the add function.
4. Print all the graph's paths.

You will use a dictionary whose keys are the nodes of the graph. For each key, the corresponding value will be a list containing nodes connected by a direct edge of this node.

```
# function for adding an edge to a graph
def addEdge(graph,u,v):
    graph[u].append(v)

# function for generating the edges of a graph
def generate_edges(graph):
    edges = []

    # for each node in graph
    for node in graph:

        # for each neighbouring node of a single node
        for neighbour in graph[node]:
            # if edge exists then append to the list
            edges.append((node, neighbour))
    return edges

# main program
# initialisation of graph as dictionary
myGraph = {"a" : ["b","c"],
           "b" : ["c", "d"],
           "c" : ["d", "e"],
           "d" : [],
           "e" : [],
          }

# print the graph contents
print("The graph contents")
print(generate_edges(myGraph))

# add more edges to the graph
addEdge(myGraph,'a','e')
addEdge(myGraph,'c','f')

# print the graph after adding new edges
print("The new graph after adding new edges")
print(generate_edges(myGraph))
```

```
The graph contents
[('a', 'b'), ('a', 'c'), ('b', 'c'), ('b', 'd'), ('c', 'd'), ('c', 'e')]
The new graph after adding new edges
[('a', 'b'), ('a', 'c'), ('a', 'e'), ('b', 'c'), ('b', 'd'), ('c', 'd'),
('c', 'e'), ('c', 'f')]
```

Exercises

1

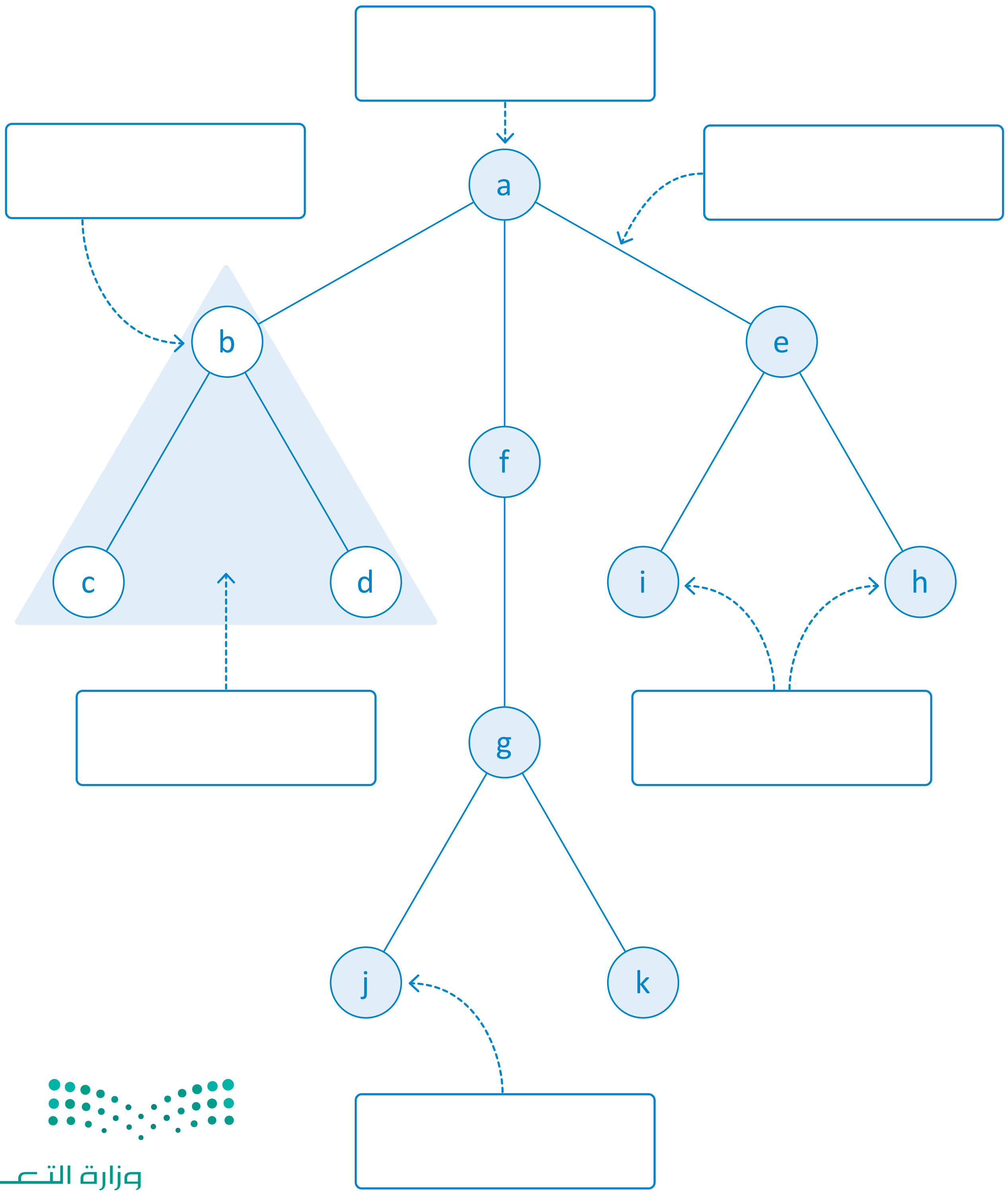
Read the sentences and tick ✓ True or False.	True	False
1. An item of a nonlinear data structure could be connected to more than one item.	○	○
2. The implementation of linear data structures is more complicated than the implementation of nonlinear data structures.	○	○
3. The leaves in a decision tree learning contain the answers to a problem.	○	○
4. The Google PageRank algorithm calculates the relative importance of a web page on the World Wide Web.	○	○
5. Neural networks are a graph used to visualize other problems.	○	○

2 State the difference between trees and graphs.

Trees	Graphs

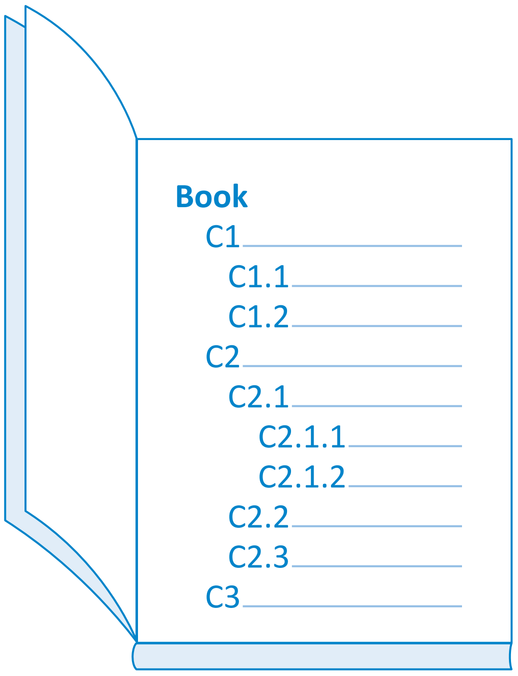
3 Describe how graph algorithms are utilized in commercial applications.

4 Fill in the blanks with the correct names of the parts of the tree.

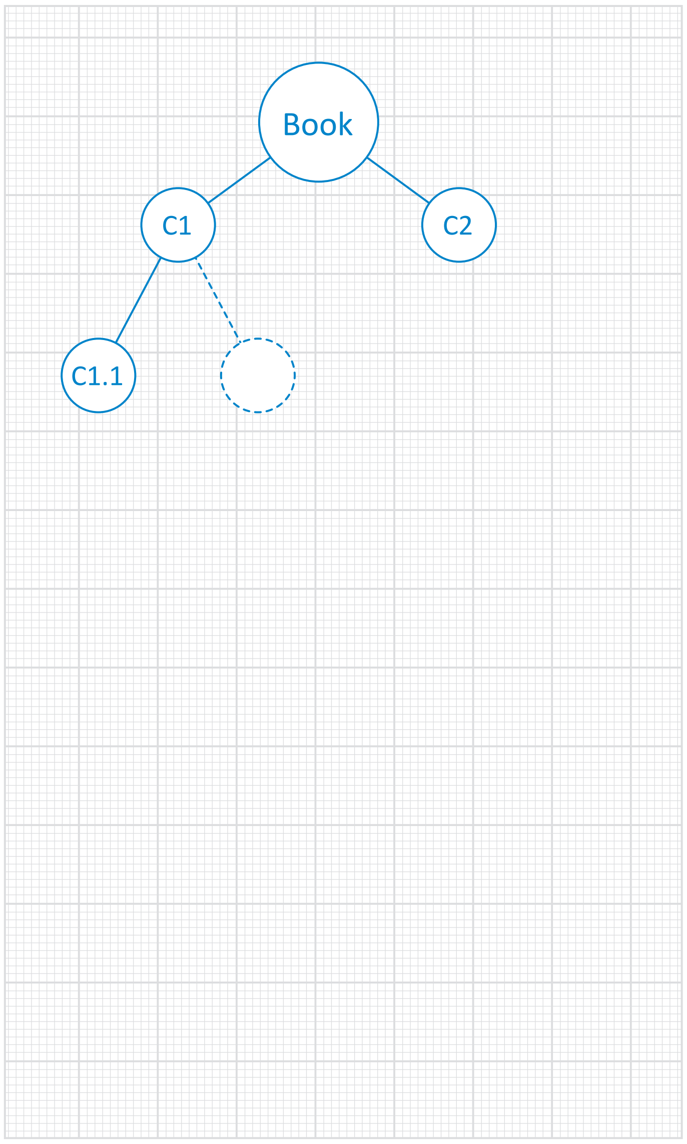


5 In the following image, you can see the book's contents page.

- Complete its tree representation.



Book
C1
C1.1
C1.2
C2
C2.1
C2.1.1
C2.1.2
C2.2
C2.3
C3



- Is it a binary tree? Justify your answer.

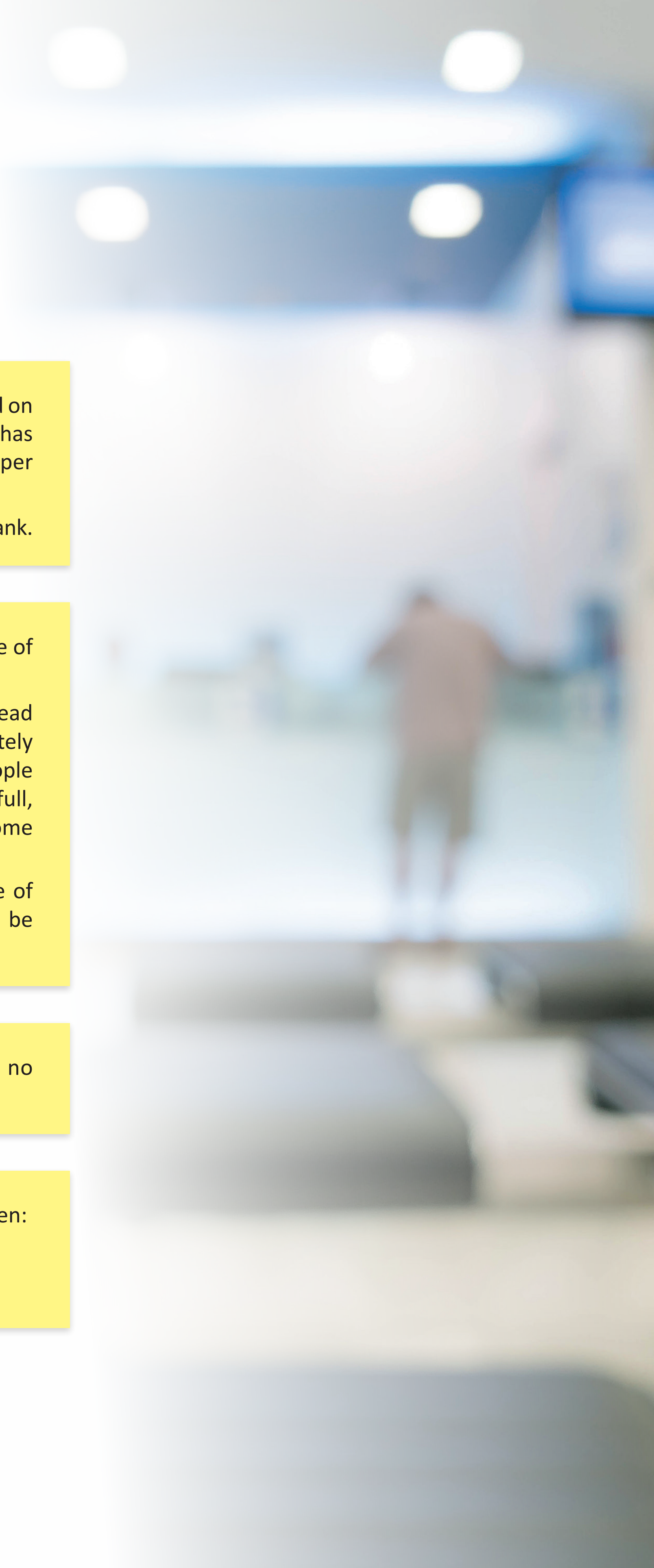
6 Draw the tree that will result from the following information:

- Node A has children B and C.
- Node D and E have the same parent which is node B.
- Node F has a sibling which is node G, and they have the same parent which is node C.
- Node H has two child nodes, I and J, and has a parent, node F.

What type of tree is described above?

Using the dictionary in Python, write the appropriate program to represent this tree and print the parents and children.

Project



Service is provided to bank customers based on their time of arrival at the branch. The bank has one cashier and the average service time per customer is 2 minutes.

The queue cannot exceed 40 people in the bank.

1

Create a Python program which will get one of the import values: "ENTRY" or "NEXT".

- If you enter the value "ENTRY", it will read the name of the customer and immediately after that will show the number of people waiting in front of him. If the queue is full, then the message "The branch is full. Come another day." will be displayed.
- If you enter the value "NEXT", the name of the next customer to be served should be displayed.

2

Repeat the above process until there are no customers waiting for service.

3

Finally the program will display on the screen:

- The number of people served.
- The average customer waiting time.

Wrap up

Now you have learned to:

> Define what AI is.

> Categorize the applications of AI.

> Classify data structures.

> Identify the difference between stack and queue data structures.

> Identify the difference between list and linked list data structures.

> Identify the difference between tree and graph data structures.

> Implement complex data structures using Python programming language.

KEY TERMS

Binary Tree	Index	Push
Child	Leaf	Rear
Data Structure	Linear	Root
Decision Tree	Linked List	Siblings
Dequeue	Non-Linear	Stack
Directed Graph	Non-Primitive	Sub-Tree
Dynamic	Null	Top
Front	Pointer	Underflow
Graph	Pop	Undirected Graph
Head	Primitive	

2. Artificial Intelligence Algorithms

In this unit, you will learn about some fundamental algorithms used in AI. You will also create a simple rule-based medical diagnosis system with multiple programming approaches and compare their results. Finally, you will learn about search algorithms and how to solve maze puzzles when taking multiple parameters into account.

Learning Objectives

In this unit, you will learn to:

> Create recursive code.

> Differentiate between the Breadth-first search and Depth-first search algorithms.

> Describe search algorithms and their application.

> Compare search algorithms.

> Describe what a rule-based system is.

> Train artificial intelligence models so they can learn to solve complex problems.

> Evaluate the results of your code and the efficiency of your program.

> Develop programs to solve simulations of real-life problems.

> Compare search algorithms.

Tools

> Jupyter Notebook

Lesson 1

Recursion

Link to digital lesson

www.ien.edu.sa

Dividing the Problem

In this lesson, you will use recursive functions to make your program more intuitive and efficient.

If your parents brought you a gift and you were eager to know what it was, but when you opened the box you found a new box inside that box, and thereafter there were boxes inside boxes, you would not know in which of those boxes the gift was.

Recursion

Recursion is one of the ways to solve problems in computer science, by dividing a problem into a group of small problems similar to the original problem so that you can use the same algorithm to solve those problems. Recursion is used by the operating system and by other applications, and most programming languages support it as well.

Recursion occurs when the same instructions are repeated but with different data and less complexity.

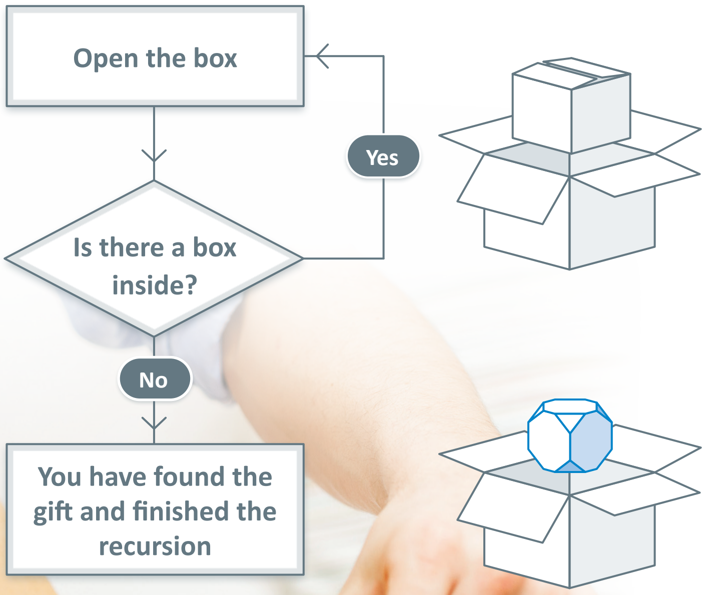


Figure 2.1: An example of recursion.

Let's look at an example of a function that calls another.

```
def mySumGrade (gradesList):
    sumGrade=0
    l=len(gradesList)
    for i in range(l):
        sumGrade=sumGrade+gradesList[i]
    return sumGrade

def avgFunc (gradesList):
    s=mySumGrade(gradesList)
    l=len(gradesList)
    avg=s/l
    return avg

# program section
grades=[89,88,98,95]
averageGrade=avgFunc(grades)
print ("The average grade is: ",averageGrade)
```

The mySumGrade function is called.

The len() function takes a list as an input argument, which counts and returns the number of items in the list.

```
The average grade is: 92.5
```

Recursive Function

In some cases, it is possible for a function to call itself, and this property is called recursive calls.

The general syntax of the recursive function:

```
# recursive function
def recurseFunction():
    if (condition): # base case
        statement
    else:
        #recursive call
        recurseFunction()

# main program
.......

# normal function call
recurseFunction()
.........
```

A recursive call is the process by which a function calls itself.

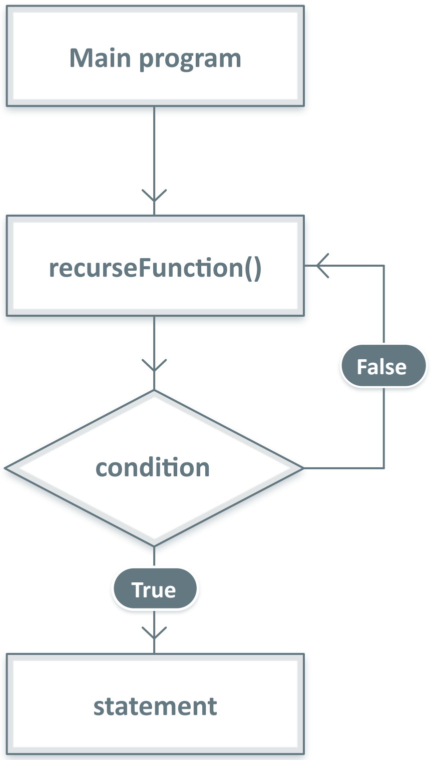


Figure 2.2: Representation of a recursive call

The recursive function consists of two states:

Base case

It is the state in which the function stops calling itself, and access to the base case is verified through a conditional statement. Without the base case, the self-recall process will be infinite.

Recursive case

It is the state in which the function calls itself when the stop condition is not met, and the function remains in this self-recalling state until it reaches the base state.

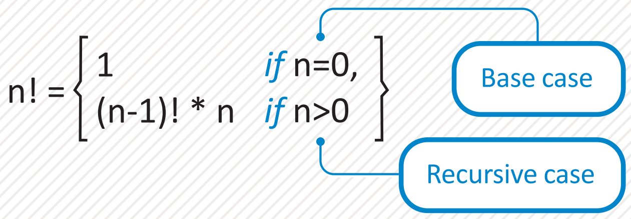
Recursion Common Examples

One of the most common examples of using recursion is the process of calculating the factorial of a specific number. The factorial of a number is defined as the product of all natural numbers smaller than or equal to that number. The factorial is expressed by the number followed by the symbol "!". For example, the factorial of 5 is 5! and 5!=5*4*3*2*1.

Table 2.1: Factorials from 0! to 5!

0!	0!=1		
1!	1!=1*1=1	or	1!= 0! *1
2!	2!=2*1=2	or	2!= 1! *2
3!	3!=3*2*1=6	or	3!= 2! *3
4!	4!=4*3*2*1=24	or	4!=3! * 4
5!	5!=5*4*3*2*1=120	or	5!=4! * 5

You notice that the process of calculating the factorial is based on the rule below



$$n! = \begin{cases} 1 & \textit{if } n=0, \\ (n-1)! * n & \textit{if } n>0 \end{cases}$$

Figure 2.3: The rule of calculating the factorial

Let's create a factorial loop that calculates the factorial of the number using the for iteration.

```
# calculate the factorial of an integer using iteration

def factorialLoop(n):
    result = 1
    for i in range(2,n+1):
          result = result * i

    return result

# main program
num = int(input("Type a number: "))
f=factorialLoop(num)
print("The factorial of ", num, " is:", f)
```

```
Type a number: 3
The factorial of 3 is:6
```

Now let's calculate the factorial of a number using a factorial function.

```python
# calculate the factorial of an integer using a
# recursive function
def factorial(x):
    if x == 0:
        return 1
    else:
        return (x * factorial(x-1))

# main program
num = int(input("Type a number: "))
f=factorial(num)
print("The factorial of ", num, " is: ", f)
```

Base case

Recursive case

```
Type a number: 3
The factorial of 3 is: 6
```

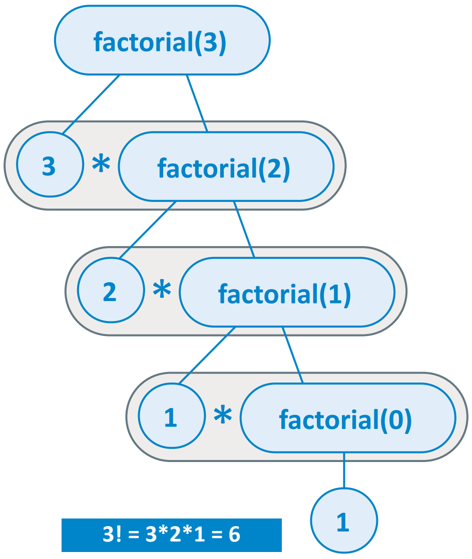


Figure 2.4: Recursion tree

Table 2.2: Advantages and disadvantages of recursion

Advantages	Disadvantages
• Recursive functions reduce the code segment to a smaller number of instructions. • A task can be broken down into a set of sub-problems using recursion. • Sometimes it's easy to use recursion to replace nested duplicates.	• Sometimes it is difficult to follow the logic of recursive functions. • Recursion requires more memory and time. • It is not easy to define cases in which recursive functions can be used.

Recursion and Iteration

Recursion and iteration are both involved in executing a set of instructions a number of times, and the main difference between recursion and iteration is the way in which a recursive function is terminated. A recursive function calls itself and ends its execution when the base state is reached. An iteration is executed repeatedly until a specific condition is met or a specific number of iterations has elapsed.

Here some of the differences are reviewed between recursion and iteration in the following table.

Table 2.3: Recursion and Iteration

Iteration	Recursion
Fast execution.	Slower execution compared to iteration.
It needs less memory.	It needs more memory.
The size of the code is larger.	The size of the code is smaller.
Ends by completing the specified number of iterations or satisfying a condition.	It expires upon reaching the base state.

When do you use recursion?

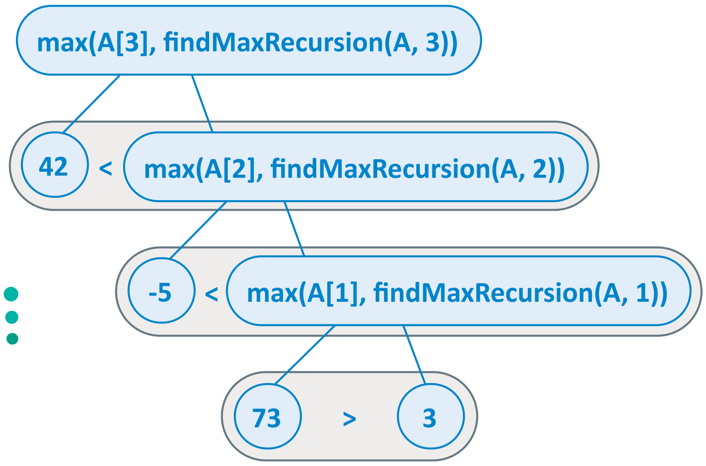
- In many cases it is considered as a more intuitive way of dealing with a problem.
- Some data structures are easy to explore using recursion.
- Some sort algorithms, such as quick sort, use recursion.

In the following example, you will extract the largest number in a list of numbers using a recursive function. Also shown in the last line of the example is another function using iteration for the purpose of comparison.

```python
def findMaxRecursion(A,n):

    if n==1:
        m = A[n-1]
    else:
        m = max(A[n-1],findMaxRecursion(A,n-1))
    return m

def findMaxIteration(A,n):

    m = A[0]
    for i in range(1,n):
        m = max(m,A[i])
    return m

# main program
myList = [3,73,-5,42]
l = len(myList)
myMaxRecursion = findMaxRecursion(myList,l)
print("Max with recursion is: ", myMaxRecursion)
myMaxIteration = findMaxIteration(myList,l)
print("Max with iteration is: ", myMaxIteration)
```

The max() function returns the element with the highest values (the largest valued element in myList).

```
Max with recursion is:  73
Max with iteration is:  73
```



max(A[3], findMaxRecursion(A, 3))
42
<
max(A[2], findMaxRecursion(A, 2))
-5
<
max(A[1], findMaxRecursion(A, 1))
73
>
3

Figure 2.5: The recursion tree of the function to extract the largest number in a list of numbers

In the next program, you will create a recursive function to calculate the power of a number.

The program will accept a number (base) and an index (power) from the user, and you will use the powerFunRecursive() recursive function which will use these two arguments to calculate the power of the base number to the exponent. The same can also be achieved with iteration and an example is included.

```python
def powerFunRecursive(baseNum,expNum):
    if(expNum==1):
       return(baseNum)
    else:
       return(baseNum*powerFunRecursive(baseNum,expNum-1))

def powerFunIteration(baseNum,expNum):

    numPower = 1
    for i in range(exp):
        numPower = numPower*base
    return numPower

# main program
base = int(input("Enter number: "))
exp = int(input("Enter exponent: "))
numPowerRecursion = powerFunRecursive(base,exp)
print( "Recursion: ", base, " raised to ", exp, " = ",numPowerRecursion)
numPowerIteration = powerFunIteration(base,exp)
print( "Iteration: ", base, " raised to ", exp, " = ",numPowerIteration)
```

```
Enter number: 10
Enter exponent: 3
Recursion: 10 raised to 3 = 1000
Iteration: 10 raised to 3 = 1000
```

Infinite Recursive Function

You have to be very careful when implementing a recursive call, you must provide a way to stop repeating by finding a specific condition to avoid unlimited redundancy occurring. During infinite recursion, the system stops responding due to too many function calls which leads to memory overflow and application termination.

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. Recursive functions have two states.	○	○
2. A recursive function calls another function.	○	○
3. Recursive functions are faster to execute.	○	○
4. Calling functions makes the code block smaller.	○	○
5. Writing repetitive code requires less recursion.	○	○

2 What are the differences between iteration and recursion?

____

____

____

____

____

3 When should recursion be used?

____

____

____

____

____

4 List the advantages and disadvantages of using recursion.

5 Write a Python recursive function to calculate the nth largest number in a list.

6 Write a Python recursive function to calculate the sum of all the even numbers in a list.

Lesson 2

DFS/BFS Algorithms

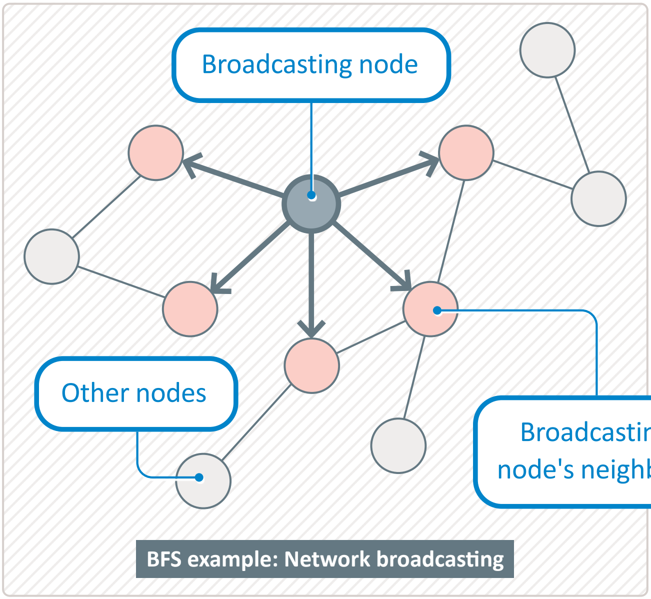
Link to digital lesson

www.ien.edu.sa

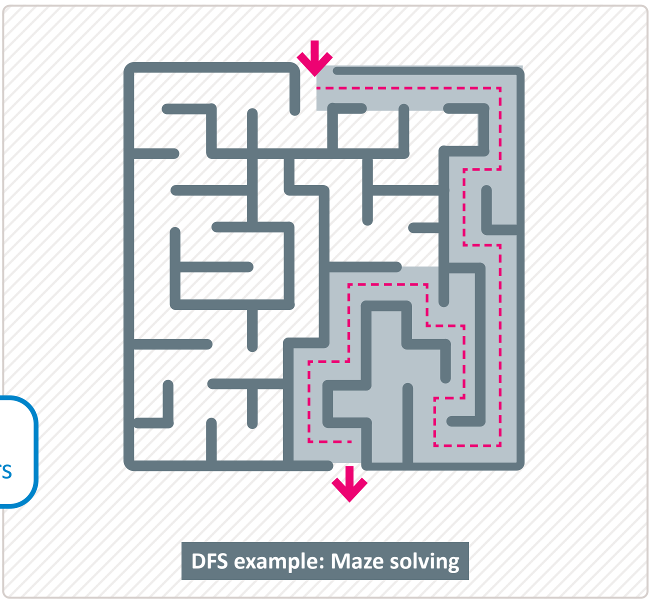
Searching in Graphs

There are cases in which you need to find a specific node in a graph (e.g. a person searching the destination city they want to travel to) or visit every node in a graph to perform a certain operation (e.g. printing the graph nodes). In order to achieve this, you need to visit every node in the graph until you find the one you need. This procedure is called graph search or graph traversal, and there are many search algorithms that help implement it, including:

- **Breadth-first search (BFS) algorithm**
- **Depth-first search (DFS) algorithm**



BFS example: Network broadcasting


DFS example: Maze solving

Breadth-first search (BFS) algorithm

The Breadth-first search (BFS) explores the graph level by level. You start from a root node (start node), then you visit the nodes that are directly connected with it, one by one. When all the nodes of the level have been visited, you move on to the next level, following the same procedure as shown in figure 2.6.

To keep track of the nodes you have visited, you use a queue. When a node is explored, you enqueue its child. Then, you dequeue the next node to be explored.

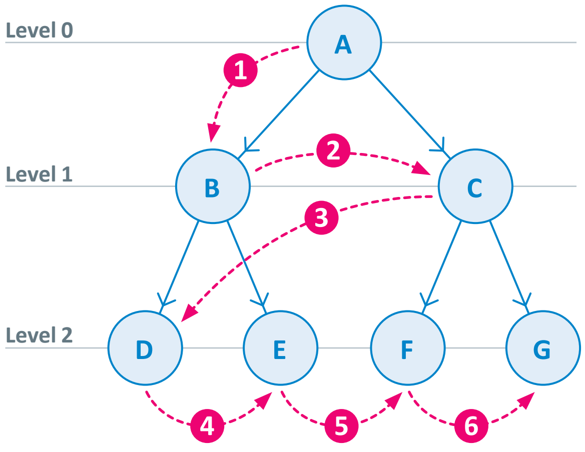
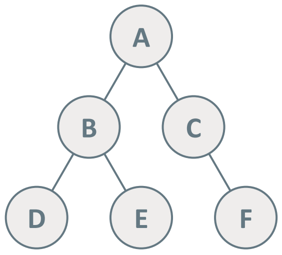
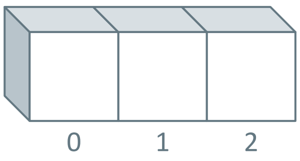


Figure 2.6: BFS Algorithm

The following example shows how the BFS algorithm works. Using the following diagram, determine which nodes to visit to get from root node A to node F.

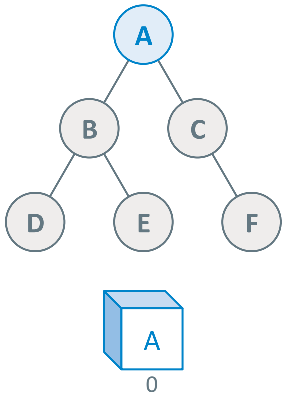
(note: use the appropriate data structure)


Graph

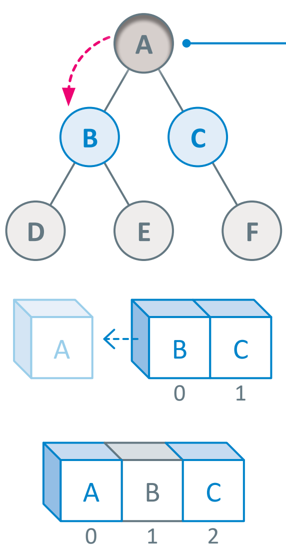

Queue

You must traverse all the nodes in layer 1 before you move on to the nodes in layer 2

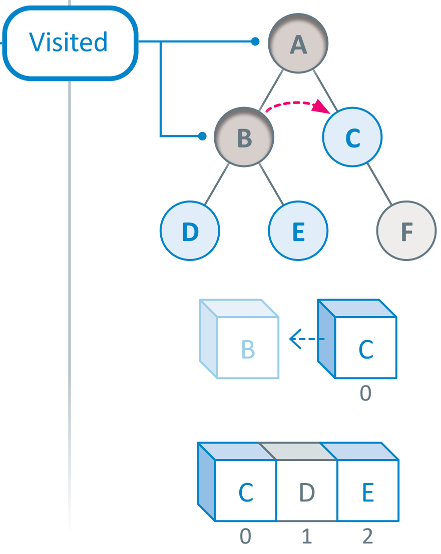
1 Starting from the root node (node A). Add the root node to the queue.



2 Remove the root node from the queue and process it. Next, add the children of the this node to the queue (nodes B and C).

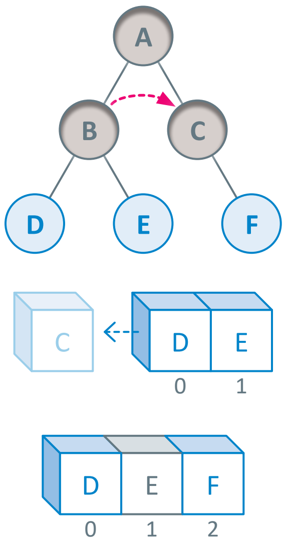


3 Remove the node at the front of the queue (node B) from the queue and process it. Next, add the children of the this node to the queue (nodes D and E).

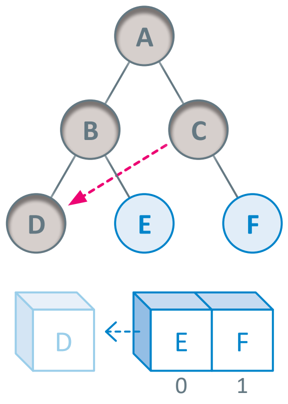


Ministry of Education
2023 - 1445

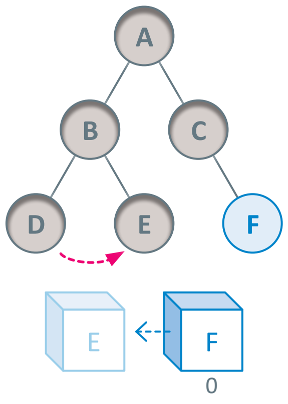
4 Remove node C and process it, then add its children.



5 Remove node D and process it. (it has no children).

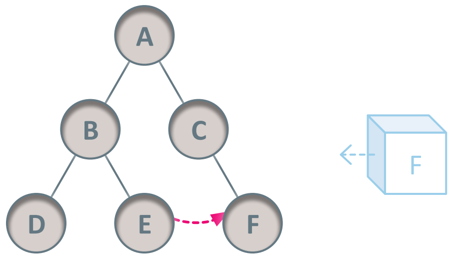


6 Remove node E and process it. (it has no children).



7 Remove node F and process it. The queue is now empty and the search is terminated.

The nodes visited using the BFS algorithm are: A, B, C, D, E, F



Let's see how you can implement the BFS algorithm in Python.

```
graph = {
    "A" : ["B","C"],
    "B" : ["D","E"],
    "C" : ["F"],
    "D" : [],
    "E" : [],
    "F" : []
}

visitedBFS = [] # List to keep track of visited nodes
queue = []      # Initialize a queue

# bfs function
def bfs(visited, graph, node):
    visited.append(node)
```

```
 queue.append(node)

 while queue:
  n = queue.pop(0)
  print (n, end = " ")

  for neighbor in graph[n]:
   if neighbor not in visited:
    visited.append(neighbor)
    queue.append(neighbor)

# main program
bfs(visitedBFS, graph, "A")
```

```
A B C D E F
```

Practical Applications of the BFS Algorithm

BFS is used by **peer-to-peer networks** to find all neighbor nodes in order to establish communication.

Social media use BFS to connect nodes of users that are related such as those with similar interests or a common location.

GPS navigation systems use BFS to find neighboring places so they can create routes for the user.

To achieve **network broadcasting** of some packets, BFS is used.

INFORMATION

The (BFS) algorithm can be developed by defining the starting point (Initial State) and the target point (Goal State) to determine the path between them.

Depth-first search (DFS) algorithm

In Depth-first search (DFS), you keep following the edges, going deeper and deeper into the graph. DFS uses a recursive procedure to traverse through the nodes. When you reach a node that has no edges to any new node, you go back to the previous node and continue the process. The DFS algorithm uses a stack data structure to keep track of the exploration trail. When a node is explored, it is pushed into the stack. When you need to go back, you pop the node from the stack as illustrated in figure 2.7.

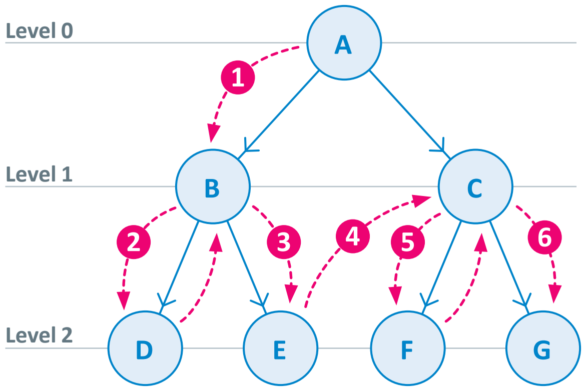
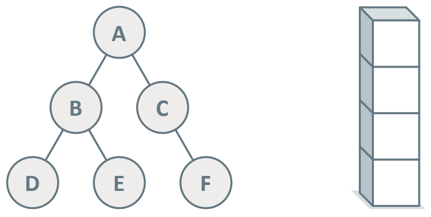


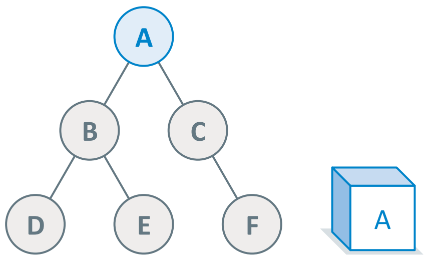
Figure 2.7: DFS Algorithm

The following example shows how the Depth-first search (DFS) algorithm works. Using the following diagram, trace the order of traversal followed by the DFS algorithm. (note: use appropriate data structure)

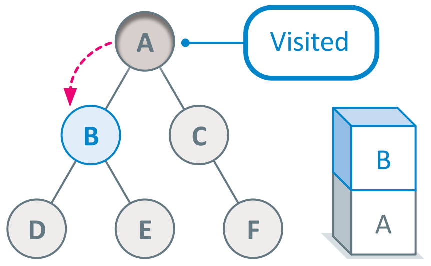


Graph Stack

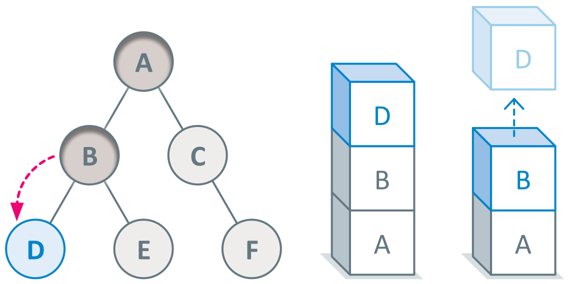
1 Process root A and add it to the stack.



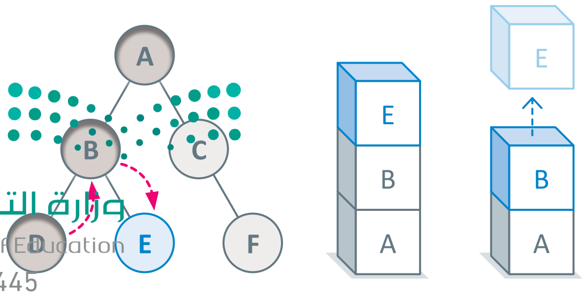
2 Process node B and add it to the stack.



3 Process node D and add it to the stack. A visited node that has no children is removed from the stack. (remove node D).



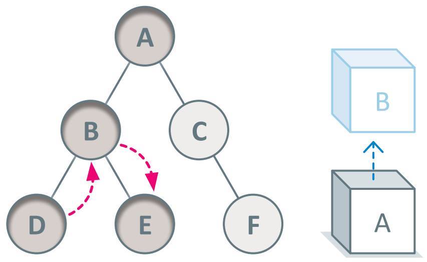
4 Process node E and add it to the stack. A visited node that has no children is removed from the stack. (remove node E).



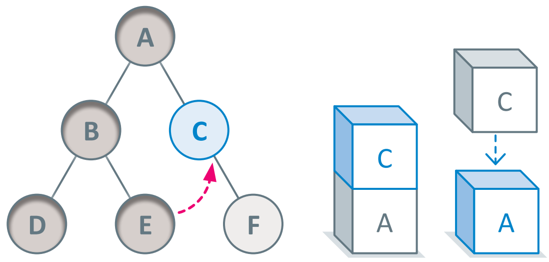
HISTORY

The first version of the Depth-First Search (DFS) algorithm was developed in the 19th century by a French mathematician as a strategy for solving mazes.

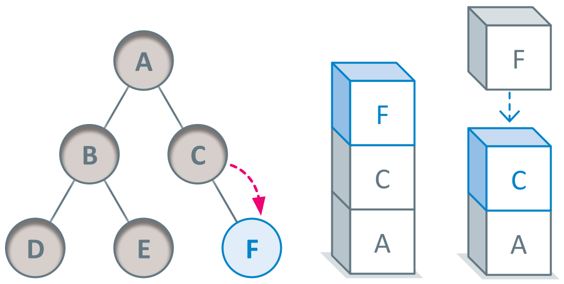
5 Remove node B.



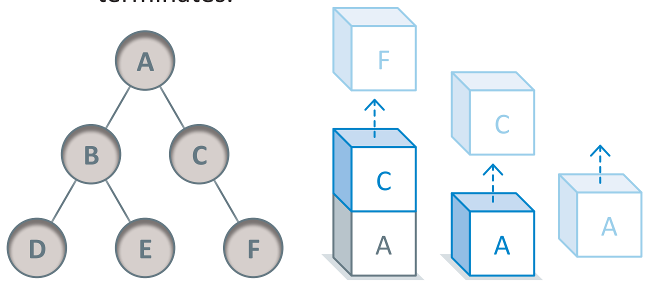
6 Process node C and add it to the stack.



7 Process node F and add it to the stack.



8 The stack is empty and the DFS accordingly terminates.



The nodes visited using the DFS algorithm are: A, B, D, E, C, F

Let's see how you can implement the Depth-first search (DFS) algorithm in Python.

```
graph = {
    "A" : ["B","C"],
    "B" : ["D","E"],
    "C" : ["F"],
    "D" : [],
    "E" : [],
    "F" : []
}

visitedDFS = [] # list to keep track of visited nodes

# dfs function
def dfs(visited, graph, node):
    if node not in visited:
        print(node, end = " ")
        visited.append(node)
        for neighbor in graph[node]:
            dfs(visited, graph, neighbor)

# main program
dfs(visitedDFS, graph, "A")
```

A stack is used indirectly through the runtime stack for tracking recursive calls.

```
A B D E C F
```

Practical Applications of the DFS Algorithm

DFS algorithm is used in **Path finding** to explore different paths in depth for maps and roads and find the best.

DFS is used to **solve mazes**, by traversing all possible routes.

Cycles in a graph can be detected using DFS by the presence of a back edge, that is passing through a node twice.

Table 2.4: Comparison of the BFS and DFS algorithms

Comparison criteria	DFS	BFS
Implementation method	Traverses according to tree depth.	Traverses according to tree level.
Data structure	Uses the stack data structure to keep track of the next location to visit.	Uses queue data structure to keep track of the next location to visit.
Use	Better when the structure of the graph is narrow and long.	Better when the structure of the graph is wide and short.
Search method	Goes to the bottom of a subtree, then backtracks.	Finds the path to the destination with the least number of edges.
First visited nodes	Children are visited before siblings.	Siblings are visited before children.

Exercises

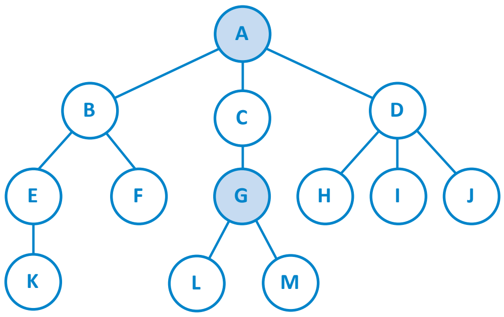
1

Read the sentences and tick ✓ True or False.	True	False
1. The BFS and DFS are implemented with the use of recursion.	○	○
2. The BFS and DFS cannot be used on tree data structures.	○	○
3. The BFS algorithm is implemented with the help of a linked list data structure.	○	○
4. The DFS algorithm can be implemented with the help of a stack data structure.	○	○
5. The BFS algorithm cannot be used in network broadcasting.	○	○

2 Explain how the BFS algorithm and the DFS algorithm work.

3 Compare the differences between the BFS and DFS algorithms.

4 In the diagram to the right, you want to go from the start node (A) to the target node (G). Apply BFS and DFS algorithms using the appropriate data structure (stack/queue), indicating which nodes are visited.



5 Write a Python function that performs BFS on a graph to check if there is a path between two given nodes.

6 Write a Python function that uses DFS to find the shortest path in a graph.

Link to digital lesson

www.ien.edu.sa

Lesson 3

Rule-based Decision Making

Rule-Based Systems

Rule-based AI systems focus on using a set of predefined rules to make decisions and solve problems. Expert systems are the most well-known example of rule-based AI. They were one of the first forms of Artificial Intelligence ever developed and were particularly popular in the 1980s and 1990s. They were often used to automate tasks that would normally require human expertise, such as diagnosing medical conditions or troubleshooting technical problems. Nowadays, rule-based systems are no longer considered to be state-of-the-art and are often outperformed by more modern AI approaches. However, they maintain their popularity in many application domains due to their ability to combine reasonable performance with an intuitive and interpretable decision-making process.

Knowledge Base

One of the key components of any rule-based AI system is the knowledge base, which is a collection of facts and rules that the system uses to make decisions. These facts and rules are typically entered into the system by human experts, who are responsible for identifying the most important information and defining the rules that the system should follow. To make a decision or solve a problem, the expert system begins by examining the facts and rules in its knowledge base and applying them to the situation at hand. If the system is unable to find a match between the facts and rules in its knowledge base, it may ask the user for additional information or refer the problem to a human expert for further assistance. Some of the main advantages and disadvantages of rule-based systems are shown in table 2.5:

Expert systems

An expert system is a type of AI that mimics the decision-making ability of a human expert. It uses a knowledge base of rules and facts and inference engines to provide advice or solve problems in a specific domain of knowledge.

Table 2.5: Main advantages and disadvantages of rule-based systems

Advantages	Disadvantages
• They can make decisions and solve problems more quickly and accurately than humans, especially when it comes to tasks that require a large amount of knowledge or data.	• They are only as good as the knowledge and rules that have been entered into their knowledge base, and they may not be able to handle situations that are outside of their area of expertise.
• They are able to operate consistently, without the biases or errors that can sometimes influence human decision-making.	• They are not able to learn or adapt in the same way that humans can and this makes them less applicable to dynamic scenarios where both the input data and logic can change significantly with time.

In this lesson, you will be introduced to rule-based systems in the context of one of their key applications: medical diagnosis. The system will provide a medical diagnosis, based on the patient's symptoms, as seen in figure 2.8. Beginning with a simple rule-based diagnostic system, you will then discover some more intelligent systems and how each iteration leads to improved results.

Iteration 1

In this first iteration, you will build a simple rule-based system that can diagnose three possible diseases: kidney stones, appendicitis, and food poisoning. The input to your system will be a simple knowledge base that maps each disease to a list of possible symptoms. This is provided in the format of a JSON file, which you load and display below.

```python
import json # a library used to save and load JSON files

# the file with the symptom mapping
symptom_mapping_file='symptom_mapping_v1.json'

# open the mapping JSON file and load it into a dictionary
with open(symptom_mapping_file) as f:
    mapping=json.load(f)

# print the JSON file
print(json.dumps(mapping, indent=2))
```

```
{
  "diseases": {
    "food poisoning": [
      "vomiting",
      "abdominal pain",
      "diarrhea",
      "fever"
    ],
    "kidney stones": [
      "lower back pain",
      "vomiting",
      "fever"
    ],
    "appendicitis": [
      "abdominal pain",
      "vomiting",
      "fever"
    ]
  }
}
```

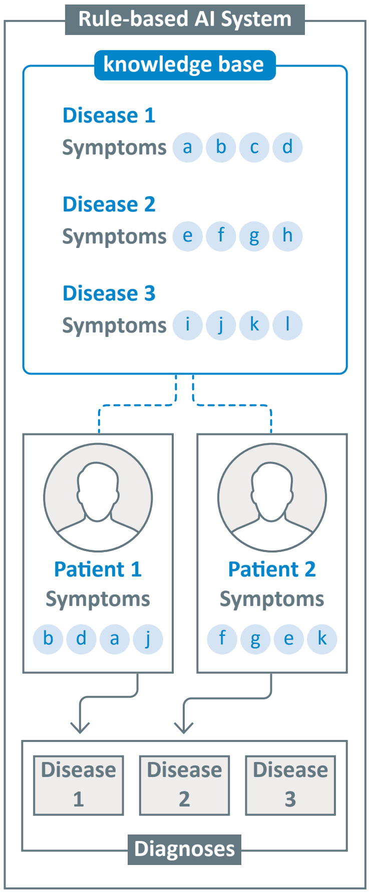


Figure 2.8: Medical diagnosis by Rule-based AI System

This first rule-based system will follow a simple rule: if the patient has at least 3 of all the possible symptoms of a disease, then the disease should be added as a possible diagnosis. Below you can find the Python function that uses this rule to make a diagnosis, given the above knowledge base and the patient's symptoms.

```python
def diagnose_v1(patient_symptoms:list):

    diagnosis=[] # the list of possible diseases

    if "vomiting" in patient_symptoms:

        if "abdominal pain" in patient_symptoms:

            if "diarrhea" in patient_symptoms:

                # 1:vomiting, 2:abdominal pain, 3:diarrhea
                diagnosis.append('food poisoning')

            elif 'fever' in patient_symptoms:

                # 1:vomiting, 2:abdominal pain, 3:fever
                diagnosis.append('food poisoning')
                diagnosis.append('appendicitis')

        elif "lower back pain" in patient_symptoms and 'fever' in patient_symptoms:

            # 1:vomiting, 2:lower back pain, 3:fever
            diagnosis.append('kidney stones')

    elif "abdominal pain" in  patient_symptoms and\
         "diarrhea" in patient_symptoms and\
         "fever" in patient_symptoms:\
        # 1:abdominal pain, 2:diarrhea, 3:fever
        diagnosis.append('food poisoning')

    return diagnosis
```

In this case, the knowledge base is hard-coded inside the function in the form of IF statements. These statements utilize the common symptoms among the three diseases to gradually arrive at a diagnosis as quickly as possible. For instance, the "vomiting" symptom is shared by all diseases. Therefore, if the first IF statement is True, then 1 of the three required symptoms for all diseases has already been accounted for. Then, you will proceed to check for "abdominal pain", which is associated with two of the diseases and continue in the same manner until all possible symptom combinations have been considered.

You can then test this function with three different patients:

```
# Patient 1
my_symptoms=['abdominal pain', 'fever', 'vomiting']
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=['vomiting', 'lower back pain', 'fever' ]
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=['fever', 'cough', 'vomiting']
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: ['food poisoning', 'appendicitis']
Most likely diagnosis: ['kidney stones']
Most likely diagnosis: []
```

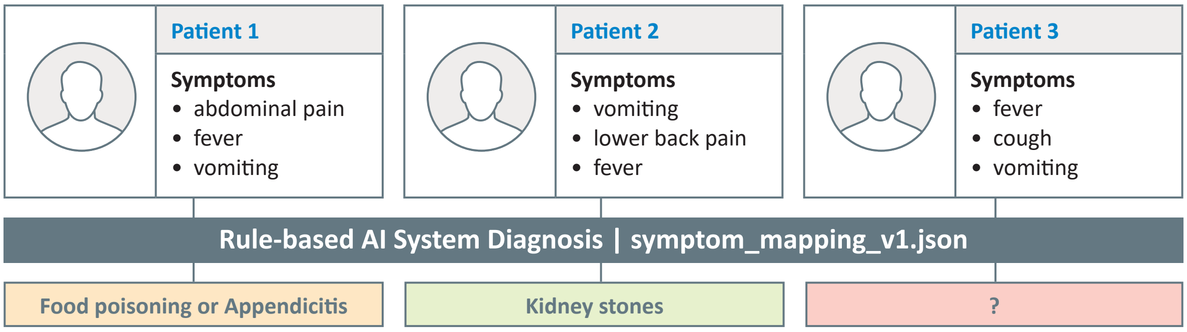


Figure 2.9: Representation of the first iteration

For Patient 1, both food poisoning and appendicitis are included in the diagnosis because the patient's three symptoms are associated with both diseases. Patient 2 is diagnosed with kidney stones, which is the only disease that matches the 3 symptoms. Finally, a diagnosis cannot be made for Patient 3, as none of the three diseases have all the 3 of the patient's symptoms.

The benefits of this first rule-based version are that it is intuitive and explainable. It is also guaranteed to consistently use its knowledge base and rules to provide a diagnosis, without bias or deviation from the standard line. However, this version also has significant disadvantages. First, the "at least 3 symptoms" rule is an oversimplified representation of how a human expert would actually make a medical diagnosis. Second, the knowledge base for this version is hard-coded in the function. Even though it was easy to create simple IF statements for such a small knowledge base, this task would become increasingly more complex and time-consuming for cases with many more diseases and symptoms.

Iteration 2

In this second iteration, you will be enhancing the flexibility and applicability of your rule-based system by making it capable of dynamically reading the knowledge base directly from a JSON file. This will eradicate the process of manually engineering symptom-specific IF statements inside the function. This is a significant improvement that will make your system applicable to larger knowledge bases with arbitrary numbers of diseases and symptoms. An example of such a knowledge base can be found below.

```
symptom_mapping_file='symptom_mapping_v2.json'

with open(symptom_mapping_file) as f:
    mapping=json.load(f)

print(json.dumps(mapping, indent=2))
```

```
{
  "diseases": {
    "covid19": [
      "fever",
      "headache",
      "tiredness",
      "sore throat",
      "cough"
    ],
    "common cold": [
      "stuffy nose",
      "runny nose",
      "sneezing",
      "sore throat",
      "cough"
    ],
   "flu": [
      "fever",
      "headache",
      "tiredness",
      "stuffy nose",
      "sneezing",
      "sore throat",
      "cough",
      "runny nose"
    ],
    "allergies": [
      "headache",
      "tiredness",
      "stuffy nose",
      "sneezing",
      "cough",
      "runny nose"
    ]
  }
}
```

This new knowledge base is only slightly larger than the previous one. However, it is clear that trying to manually create IF statements in this case would be significantly harder. For instance, the previous knowledge base had one disease with four symptoms and two diseases with three symptoms. Given the "at least 3 symptoms" rule that you applied in version 1, this led to 6 possible symptom triplets to consider. In the new knowledge base above, the four diseases have 5, 5, 8, and 6 symptoms. This leads to 96 possible triplets! In a case where you would have to deal with hundreds or even thousands of diseases, it would be impossible to create a system like the one in the first version.

In addition, there is no valid medical reason for being limited to symptom triplets. Therefore, you will also make the diagnosis logic more versatile by counting the number of matching symptoms for each disease and allowing the user to specify the number of matching symptoms that a disease must have to be included in the diagnosis.

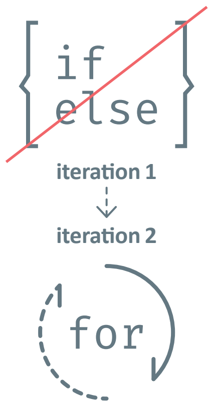


Figure 2.10: The second iteration has no hard-coded IF statements

```python
def diagnose_v2(patient_symptoms:list,
                symptom_mapping_file:str,
                matching_symptoms_lower_bound:int):

    diagnosis=[]

    with open(symptom_mapping_file) as f:
        mapping=json.load(f)

    # access the disease information
    disease_info=mapping['diseases']

    # for every disease
    for disease in disease_info:

        counter=0

        disease_symptoms=disease_info[disease]

        # for each patient symptom
        for symptom in patient_symptoms:

            # if this symptom is included in the known symptoms for the disease
            if symptom in disease_symptoms:

                counter+=1

        if counter>=matching_symptoms_lower_bound:
            diagnosis.append(disease)

    return diagnosis
```

This version has no hard-coded IF statements. After loading the symptom mapping from the JSON file, it proceeds to consider every possible disease via the first FOR loop. The loop checks each of the patient's symptoms with the known symptoms for the disease and increases a counter every time it finds a match.

```
# Patient 1
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v2(my_symptoms,'symptom_mapping_v2.json' , 3)
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v2(my_symptoms, 'symptom_mapping_v2.json' , 4)
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=['fever', 'cough', 'vomiting']
diagnosis=diagnose_v2(my_symptoms, 'symptom_mapping_v2.json' , 3)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: ['common cold', 'flu', 'allergies']
Most likely diagnosis: ['common cold']
Most likely diagnosis: []
```

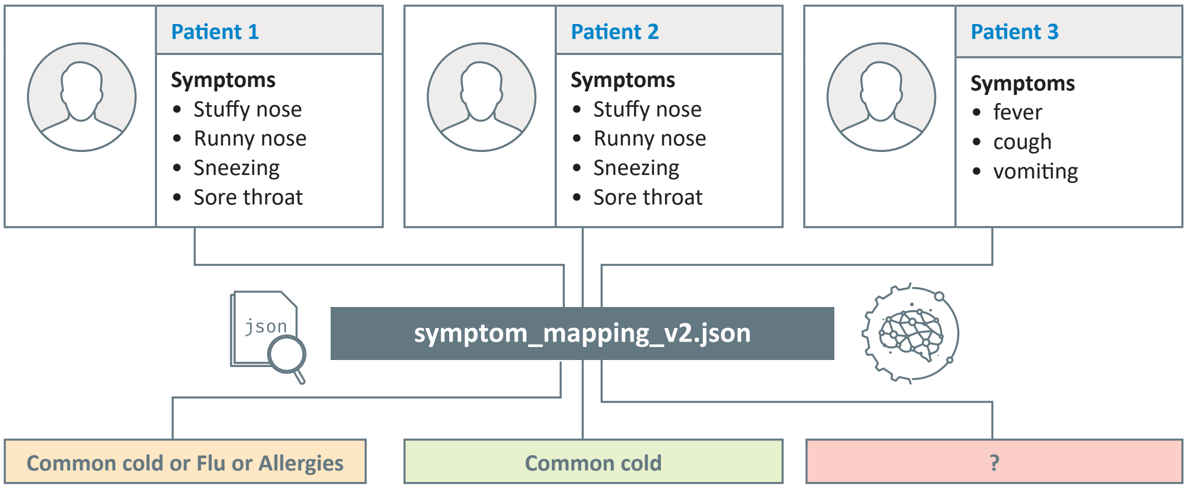


Figure 2.11: Representation of the second iteration

Observe that this second iteration is a generalized version of the first iteration. However, this iteration is much more widely applicable, as it can be used as-is with any other knowledge base of the same format, even if it includes thousands of diseases with an arbitrary number of symptoms. It also allows the user to make the diagnosis more or less strict by tuning the matching_symptoms_lower_bound parameter. This can be observed for Patients 1 and 2: even though they have the same symptoms, tuning this parameter leads to a significantly different diagnosis.

Despite these improvements, this version is still far from perfect and is still not an accurate representation of an actual medical diagnosis.

Iteration 3

In this third iteration, you will increase the intelligence of our rule-based system by giving it access to a more detailed type of knowledge base. This new type will take into account the medical fact that certain symptoms are more common than others for each disease.

```python
symptom_mapping_file='symptom_mapping_v3.json'

with open(symptom_mapping_file) as f:
    mapping=json.load(f)

print(json.dumps(mapping, indent=2))
```

```
{
  "diseases": {
    "covid19": {
      "very common": [
        "fever",
        "tiredness",
        "cough"
      ],
      "less common": [
        "headache",
        "sore throat"
      ]
    },
    "common cold": {
      "very common": [
        "stuffy nose",
        "runny nose",
        "sneezing",
        "sore throat"
      ],
      "less common": [
        "cough"
      ]
    },
    "flu": {
      "very common": [
        "fever",
        "headache",
        "tiredness",
        "sore throat",
        "cough"
      ],
      "less common": [
        "stuffy nose",
        "sneezing",
        "runny nose"
      ]
    },
    "allergies": {
      "very common": [
        "stuffy nose",
        "sneezing",
        "runny nose"
      ],
      "less common": [
        "headache",
        "tiredness",
        "cough"
      ]
    }
  }
}
```

The threshold-based logic on the number of symptoms will be abandoned and replaced with a scoring function that assigns custom weights to very common and less common symptoms. The user will also be given the flexibility to specify whatever weights they think are appropriate. The disease or diseases with the highest weighted sum will then be included in the diagnosis.

```
from collections import defaultdict

def diagnose_v3(patient_symptoms:list,
                symptom_mapping_file:str,
                very_common_weight:float=1,
                less_common_weight:float=0.5
               ):

    with open(symptom_mapping_file) as f:
        mapping=json.load(f)

    disease_info=mapping['diseases']

    # holds a symptom-based score for  each potential disease
    disease_scores=defaultdict(int)

    for disease in disease_info:

        # get the very common symptoms of the disease
        very_common_symptoms=disease_info[disease]['very common']

        # get the less common symptoms for this disease
        less_common_symptoms=disease_info[disease]['less common']

        for symptom in patient_symptoms:

            if symptom in very_common_symptoms:
                disease_scores[disease]+=very_common_weight

            elif symptom in less_common_symptoms:
                disease_scores[disease]+=less_common_weight

    # find the max score all candidate diseases
    max_score=max(disease_scores.values())

    if max_score==0:
        return []

    else:
        # get all diseases that have the max score
        diagnosis=[disease for disease in disease_scores if disease_scores
[disease]==max_score]

        return diagnosis, max_score
```

For each possible disease included in the knowledge base, this new function identifies the very common and less common symptoms exhibited by the patient. It then increases the disease's score according to the respective weights. In the end, the diseases with the maximum score are included in the diagnosis. You can now test this new implementation with a few examples:

```
# Patient 1
my_symptoms=["headache", "tiredness", "cough"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json')
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json')
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json', 1, 1)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: (['flu'], 3)
Most likely diagnosis: (['common cold'], 4)
Most likely diagnosis: (['common cold', 'flu'], 4)
```

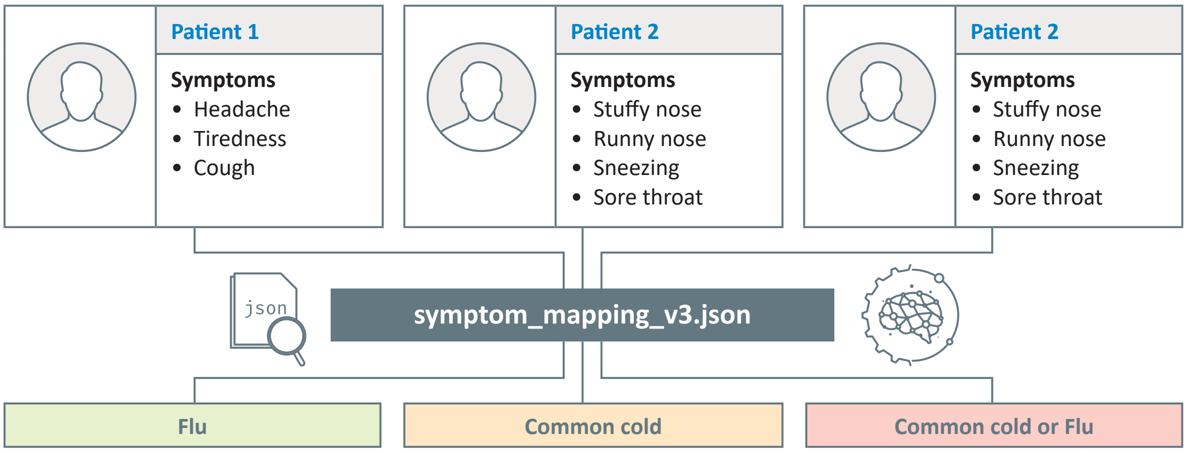


Figure 2.12: Representation of the third iteration

You may observe that, even though the 3 symptoms for Patient 1 ("headache", "tiredness", "cough") are shared by the flu, covid19, and allergies, only the flu is included in the diagnosis. This is because all three symptoms are listed as 'very common' in the knowledge base, leading to a maximum score of 3. Similarly, while Patients 2 and 3 have the same symptoms, the different weights submitted for very common and less common symptoms lead to different diagnoses. Specifically, using an equal weight for the two symptom types leads to the addition of the flu in the diagnosis.

Iteration 4

The rule-based system could be further improved by increasing the sophistication of the knowledge base and by experimenting with different scoring functions. Even though this would indeed lead to improvement, it would still require a considerable amount of time and manual effort. Thankfully, there is a way to automatically build a rule-based system that is intelligent enough to directly construct its own knowledge base and scoring function: by using machine learning. Rule-based machine learning applies a learning algorithm to automatically identify useful rules, rather than a human needing to apply prior domain knowledge to manually construct rules and curate a rule set

Instead of a hand-crafted knowledge base and a scoring function, a machine learning algorithm expects only one input: a historical dataset of patient cases. By learning directly from data, problems associated with the acquisition and validity of background knowledge are prevented. Each case consists of a patient's symptoms and a medical diagnosis made by a human expert. Given such a training dataset, the algorithm can then automatically learn how to predict the most likely diagnosis for a new patient.

```
import pandas as pd # import pandas to load and process spreadsheet-type data

medical_dataset=pd.read_csv('medical_data.csv') # load a medical dataset.

medical_dataset
```

	fever	cough	tiredness	headache	stuffy nose	runny nose	sneezing	sore throat	diagnosis
0	1	1	1	0	0	0	0	0	covid19
1	0	1	1	1	0	0	0	0	covid19
2	1	1	1	0	0	0	0	0	covid19
3	1	1	1	0	0	0	0	0	covid19
4	1	1	1	0	0	0	0	0	covid19
...	...	...	...	...	...	...	...	...	...
1995	0	1	0	0	1	0	1	1	common cold
1996	0	0	0	1	1	1	1	0	common cold
1997	0	0	1	0	1	0	0	1	common cold
1998	0	0	0	0	1	0	0	1	common cold
1999	0	1	0	0	0	0	1	1	common cold

The dataset consists of 2,000 patient cases. Each case has 8 possible symptoms: fever, cough, tiredness, headache, stuffy nose, runny nose, sneezing, and sore throat. Each of these is encoded in a separate binary column. A binary digit 1 means that the patient had the symptom, while a binary digit 0 means that the patient did not have it.

The final column includes the diagnosis made by the human expert. There are four possible diagnoses: covid19, flu, allergies, common cold.

You can easily validate this with Python code:

```
set(medical_dataset['diagnosis'])
```

Even though there are dozens of possible machine learning algorithms that can be used with such a dataset, you will use one that follows the logic-based approach: a decision tree. Specifically, you will use the DecisionTreeClassifier class from the popular sklearn Python library.

```
from sklearn.tree import DecisionTreeClassifier

def diagnose_v4(train_dataset:pd.DataFrame):

    # create a DecisionTreeClassifier
    model=DecisionTreeClassifier(random_state=1)

    # drop the diagnosis column to get only the symptoms
    train_patient_symptoms=train_dataset.drop(columns=['diagnosis'])

    # get the diagnosis column, to be used as the classification target
    train_diagnoses=train_dataset['diagnosis']

    # build a decision tree
    model.fit(train_patient_symptoms, train_diagnoses)

    # return the trained model
    return model
```

The Python implementation of this fourth version is considerably shorter and simpler than the previous ones. It simply reads the training file, uses it to build a decision tree model based on the relations between symptoms and diagnoses, and then returns the custom model. In order to properly test this version, begin by splitting our dataset into two separate training and testing sets.

```
from sklearn.model_selection import train_test_split

# use the function to split the data, get 30% for testing and 70% for training.
train_data, test_data = train_test_split(medical_dataset, test_size=0.3,
random_state=1)

#print the shapes (rows x columns) of the two datasets
print(train_data.shape)
print(test_data.shape)
```

```
(1400, 9)
(600, 9)
```

You now have 1,400 data points that will be used for training the model and 600 that will be used to test it. Begin by training and visualizing the decision tree model.

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

my_tree=diagnose_v4(train_data) # train a model

print(my_tree.classes_) # print the possible target labels (diagnoses)

plt.figure(figsize=(12,6)) # size of the visualization, in inches

# plot the tree
plot_tree(my_tree,
          max_depth=2,
          fontsize=10,
          feature_names=medical_dataset.columns[:-1]
         )
```

```
['allergies' 'common cold' 'covid19' 'flu']
```

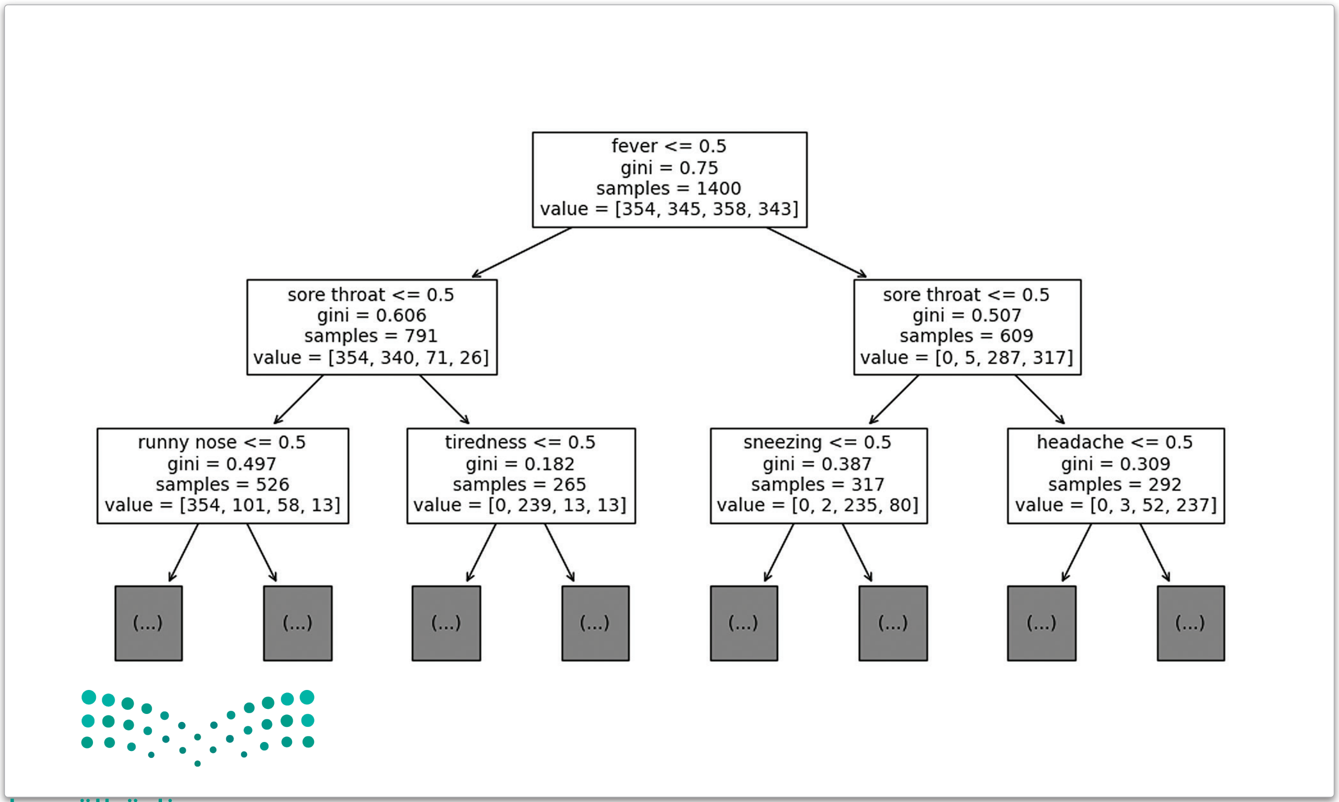


Figure 2.13: Decision tree model for the medical_data dataset, with two levels depth

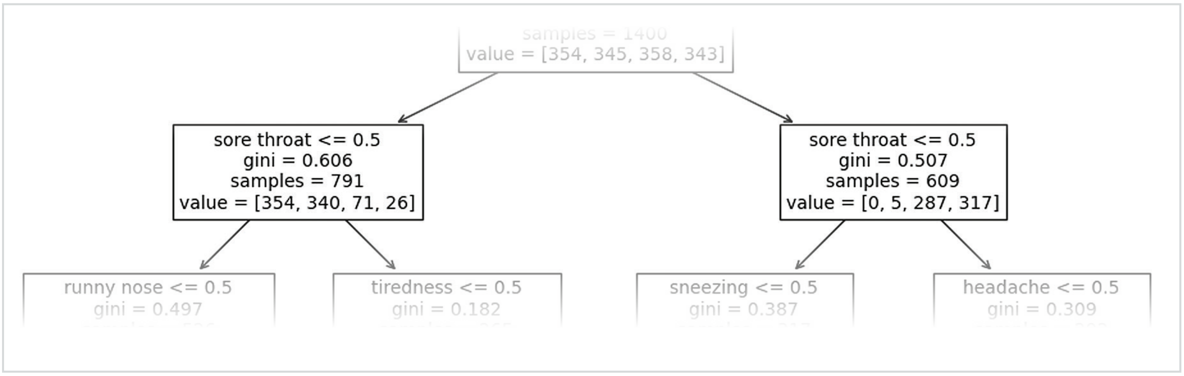
The plot_tree() function is used to visualize a decision tree. For lack of space, only the first two levels (plus the root) are visualized. This number can be easily tuned via the max_depth parameter.

```
# plot the tree
plot_tree(my_tree,
          max_depth=2,
          fontsize=10
```

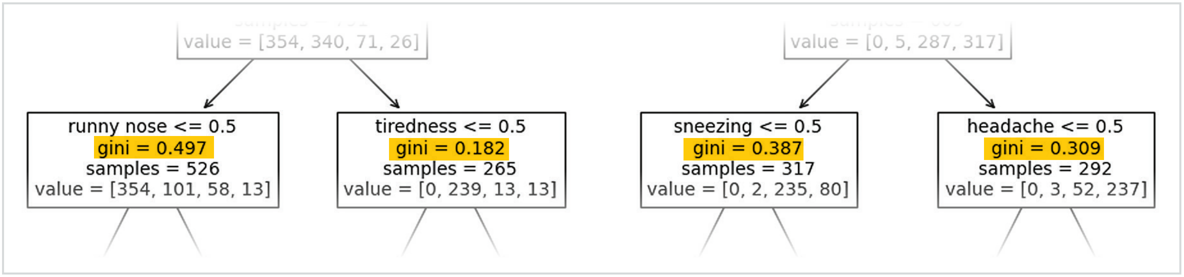
Depth of the decision tree

Each node in the tree represents a subset of the patients. For example, the root node represents the full population of the 1,400 patients in the training set. Out of these, 354, 345, 358, and 343 patients were diagnosed with allergies, the common cold, covid19 and the flu, respectively.

```
       fever <= 0.5
        gini = 0.75
      samples = 1400
value = [354, 345, 358, 343]
```



The tree is built in a top-down fashion via binary splits. The first split is based on whether or not the patient has a fever or not. Given that all symptom features are binary, a <=0.5 check is True if the patient did not have the symptom. Those that did not have a fever (left path) are further split based on whether or not they had a sore throat. Those that did not are then split based on whether or not they had a runny nose. The node at this point includes 526 cases. Out of those, 354, 101, 58, and 13 were diagnosed with allergies, the common cold, covid19, and the flu, respectively.



The splitting continues until the algorithm determines that the cases have been separated into sufficiently pure nodes. A perfectly pure node is one that only includes cases with the same diagnosis. The "gini" values marked on each node represent scores of the gini index, a popular formula used to evaluate the purity of a given node.

The gini index measures a node's impurity, namely the likelihood of the node's contents being classified in the wrong class. The lower the gini index, the more certain the algorithm can be about the classification.

You will now use this decision tree to predict the most likely diagnosis for the patients in the testing set. The testing set is used to evaluate the performance of the model. The exact evaluation method depends on whether the task is one of regression or classification. In classification problems, like the one presented here, computing a model's accuracy and confusion matrix is a common evaluation method.

- Accuracy is the proportion of correct predictions made by the classifier. A high accuracy (closer to 100%) means that the classifier is making mostly correct predictions.
- A Confusion Matrix is a table that compares the true (actual) labels in a dataset with the predictions made by the classifier. The table includes one row for each true label and one column for each predicted label. Each entry in the matrix represents the number of instances that have the corresponding true and predicted labels.

```
# functions used to evaluate a classifier
from sklearn.metrics import accuracy_score,confusion_matrix

# drop the diagnosis column to get only the symptoms
test_patient_symptoms=test_data.drop(columns=['diagnosis'])

# get the diagnosis column, to be used as the classification target
test_diagnoses=test_data['diagnosis']

# guess the most likely diagnoses
pred=my_tree.predict(test_patient_symptoms)

# print the achieved accuracy score
accuracy_score(test_diagnoses,pred)
```

```
0.8166666666666667
```

You will observe that the decision tree achieves an accuracy of 81.6%. This means that, out of all 600 test cases, the tree correctly diagnoses 490 of them. You can also print the model's confusion matrix to get a better view of the number of misclassified examples.

```
confusion_matrix(test_diagnoses,pred)
```

```
array([[143,   3,   0,   0],
       [ 48,  98,   5,   4],
       [  2,   1, 127,  12],
       [  1,   3,  31, 122]])
```

	Predicted allergies	Predicted common cold	Predicted covid19	Predicted flu
Actual allergies	**143**	3	0	0
Actual common cold	48	**98**	5	4
Actual covid19	2	1	**127**	12
Actual flu	1	3	31	**122**

Figure 2.14: Confusion matrix of predicted and actual cases

The numbers outside of the diagonal represent the model's mistakes.

For instance, given that the order of the four possible diagnoses is ['allergies', 'common cold', 'covid19', 'flu'], the matrix informs us that there were 48 cases of the common cold that were misclassified as allergies and 31 cases of the flu that were misclassified as covid19.

Even though the model is not perfect, the fact that it can achieve such a high accuracy by learning its own rule set and without the need for a manually-constructed knowledge base is impressive. Another encouraging factor is that this accuracy is achieved without trying to tune the various performance parameters of the DecisionTreeClassifier. It is thus very likely that we can improve the model even further. Another obvious way to improve is to go beyond the limitation of the rule-based model and experiment with different types of machine learning algorithms. You will explore some of these methods in the following unit.

Exercises

1 What are some advantages and disadvantages of rule-based systems?

2 What is an advantage and a disadvantage of the first iteration?

3 Add a patient to your code in the first iteration of the rule-based system with the symptoms ["vomiting", "abdominal pain", "diarrhea", "fever", "lower back pain"]. What is the diagnosis for this patient? Present your observations below.

4 In the second iteration, how many diseases does each patient's diagnosis contain, if you change the parameter matching_symptoms_lower_bound to 2, 3 and 4? Modify your code and present your observations.

5 In the third iteration, change both weights to 1 for patients 1 and 2, just like the third patient's. Modify your code and present your observations.

6 Describe briefly how each iteration is enhanced from the previous one (first to second, second to third, third to fourth).

Link to digital lesson

www.ien.edu.sa

Lesson 4

Informed Search Algorithms

Applications of Search Algorithms

Search algorithms are a key component of AI systems, as they enable the exploration of different possibilities for finding good solutions to complex problems with numerous mainstream applications. Some examples of their applications include:

- **Robotics:** A robot might use a search algorithm to find its way through a maze or to locate an object in its environment.
- **E-commerce websites:** Online shopping websites use search algorithms to match customers' queries with available products, filter results based on criteria such as price, brand, and ratings, and suggest related products.
- **Social media platforms:** Social media platforms use search algorithms to show users the most relevant posts, people, and groups based on keywords and user interests.
- **Enabling a machine to play games at a high level of skill:** A chess or Go-playing AI might use a search algorithm to evaluate different moves and choose the one that is most likely to lead to a win.
- **GPS navigation systems:** GPS navigation systems use search algorithms to find the shortest and fastest route between two locations, taking into account real-time traffic data.
- **File management systems:** Search algorithms are used in file management systems to quickly locate specific files based on their names, contents, or other attributes.

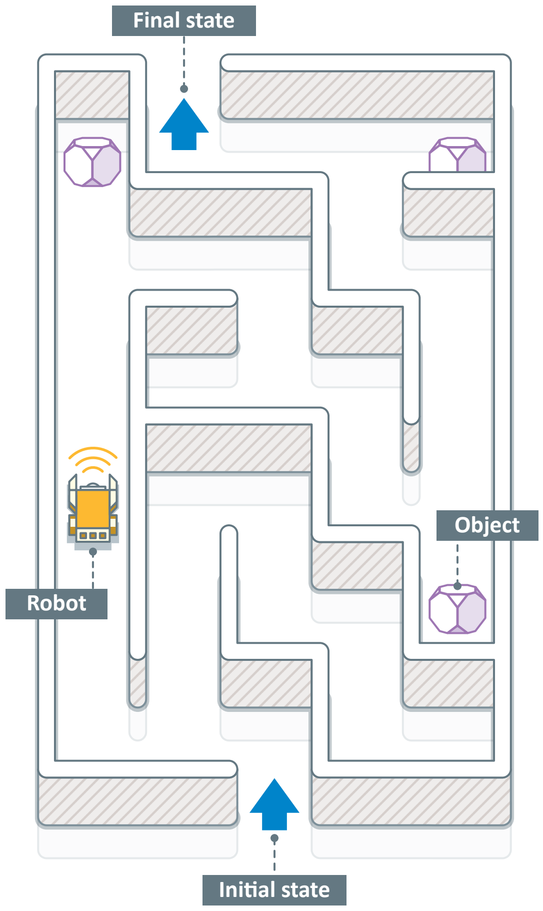


Figure 2.15: A robot uses a search algorithm to find its way

Types and Examples of Search Algorithms

There are two main types of search algorithms: **uninformed** and **informed**.

Uninformed Search Algorithms

Uninformed search algorithms, also known as blind search algorithms, have no additional information about the states of a problem beyond those provided in the problem definition and perform an exhaustive search of the search space by following a predetermined set of rules. The breadth-first search (BFS) and depth-first search (DFS) techniques covered in lesson 2 are examples of uninformed search algorithms.

For instance, DFS begins at the root node of a tree or graph and always expands to the deepest unvisited node. It proceeds in this manner until it has exhausted the entire search space by visiting all available nodes. It then reports the best solution that was found during the search. The fact that DFS always follows these rules and does not adjust its strategy regardless of what it discovers during its search makes it an uninformed algorithm.

Another notable example in this family is Iterative Deepening Depth-First Search (IDDFS), which can be viewed as a combination of the DFS and BFS algorithms, as it uses a depth-first strategy to iteratively explore the full breadth of options up to a certain node.

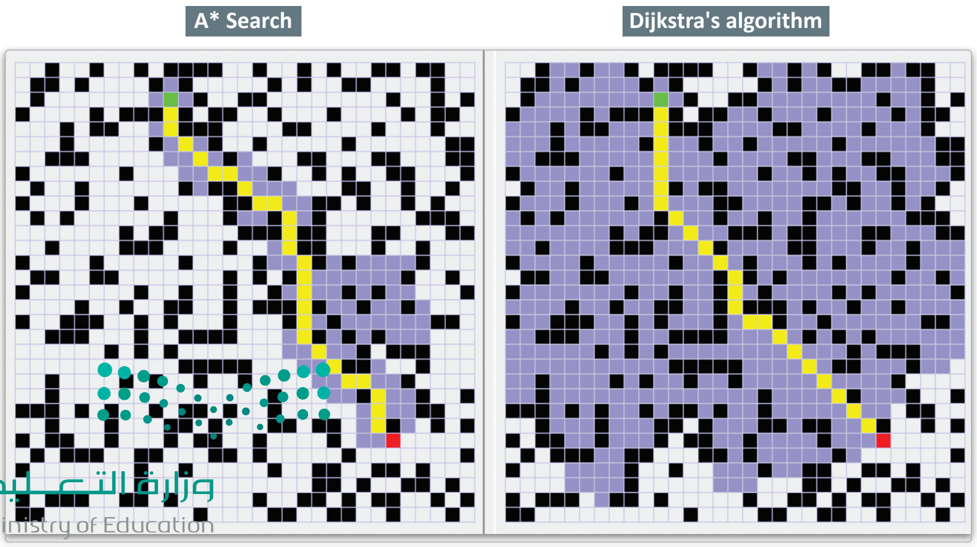
Informed Search Algorithms

In contrast to the uninformed search algorithms, informed search algorithms use information about the problem and the search space to guide their search. Examples of such algorithms include:

- **A* Search**, which uses a heuristic function to estimate the distance between each of the candidate nodes and the goal node. It then expands the candidate node with the lowest estimate. The A* Search algorithm is as good as its heuristic function. For instance, if the heuristic is guaranteed never to overestimate the actual distance to the goal, then the algorithm is guaranteed to find the optimal solution. Otherwise, the returned solution might not be the best possible one.
- **Dijkstra's algorithm**, which expands the node with the actual lowest distance to the goal in every step. Therefore, contrary to A* Search, Dijkstra actually computes the real distance and does not use heuristic estimates. While this makes Dijkstra slower than A* Search, it also means that it is always guaranteed to find the optimal solution (the shortest path from the start to the goal).
- **Hill climbing**, which starts by generating a random solution. It then tries to iteratively improve this solution by making small changes that increase a specific heuristic function. Even though this approach is not guaranteed to find the optimal solution, it is easy to implement and can be very efficient for certain types of problems.

Heuristic function

A function that ranks alternatives in search algorithms at each branching stage depending on available data to choose which branch to pursue.



The purple cells are the the visited cells, the green cell is the start location, the red cell is the finish location and the yellow cells represent the found route.

Figure 2.16: A* Search and Dijkstra's algorithm solving the same maze

In this unit you will see some visual examples and Python implementations of BFS and A* Search to demonstrate the differences between informed and uninformed search algorithms.

Creating Maze Puzzles in Python

Consider the following simple maze puzzle:

The maze is defined as a 3x3 grid. The starting position is marked by a star in the lower left corner of the maze. The goal is to reach the target cell marked by the X. The player can move to any free cell that is adjacent to their current position.

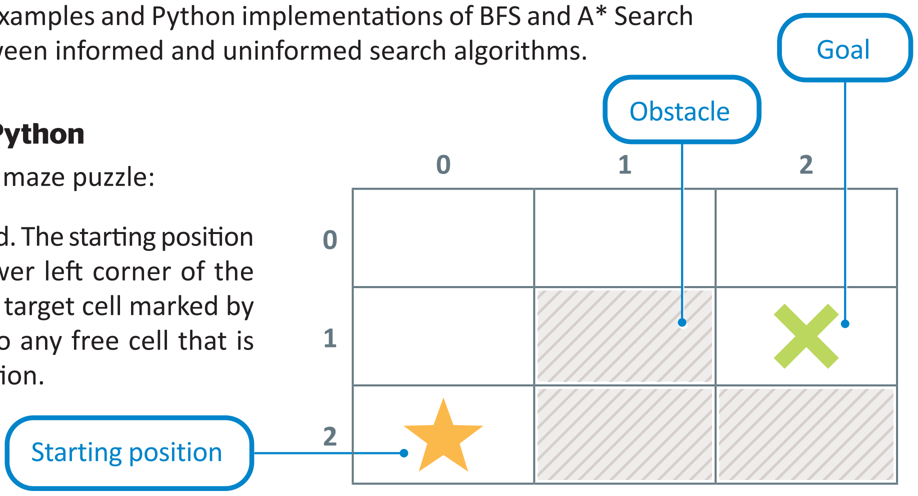


Figure 2.17: Simple maze puzzle

A cell is considered free unless it is already occupied by a block. For instance, the example maze shown above has 3 cells occupied by blocks. These blocks are colored dark grey and form an obstacle that the player has to circumvent to get to the X. The player can move to any horizontally, vertically, or diagonally adjacent free cell. For instance:

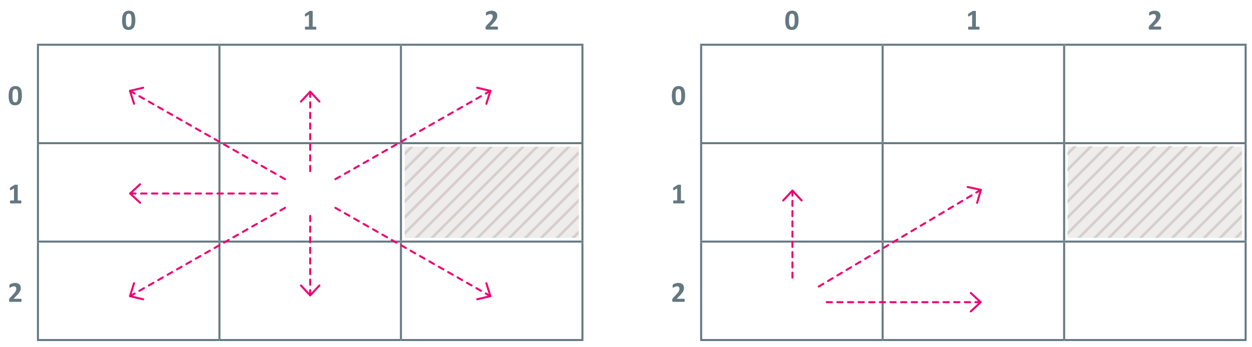


Figure 2.18: The player can move to any horizontally, vertically, or diagonally adjacent free cell

The objective is to find the shortest possible path and find it with the smallest possible number of cell visits. Even though a small 3x3 maze might seem trivial to a human player, any intelligent algorithmic solution has to work for arbitrarily large and complex mazes. For instance, consider a massive 10.000x10.000 maze with millions of blocks scattered in various complex shapes.

The following Python code can be used to create a dataset that represents the example shown in figure 2.18.

```python
import numpy as np

# create a numeric 3 x 3 matrix full of zeros.
small_maze=np.zeros((3,3))

# coordinates of the cells occupied by blocks
blocks=[(1, 1), (2, 1), (2, 2)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    small_maze[block]=1

small_maze
```

```
array([[0., 0., 0.],
       [0., 1., 0.],
       [0., 1., 1.]])
```

In this numeric representation of a maze, free and occupied cells are represented by zeros and ones, respectively. The same code can also be easily updated to create arbitrarily large and complex mazes. For example:

```python
import random

random_maze=np.zeros((10,10))

# coordinates of 30 random cells occupied by blocks
blocks=[(random.randint(0,9),random.randint(0,9)) for i in range(30)]

for block in blocks:
    random_maze[block]=1
```

The following function can be used to visualize a maze:

```python
import matplotlib.pyplot as plt # library used for visualization

def plot_maze(maze):
    ax = plt.gca()          # create a new figure
    ax.invert_yaxis()       # invert the y-axis to match the matrix
    ax.axis('off')          # hide the axis labels
    ax.set_aspect('equal') # make sure the cells are rectangular

    plt.pcolormesh(maze, edgecolors='black', linewidth=2,cmap='Accent')
    plt.show()

plot_maze(random_maze)
```

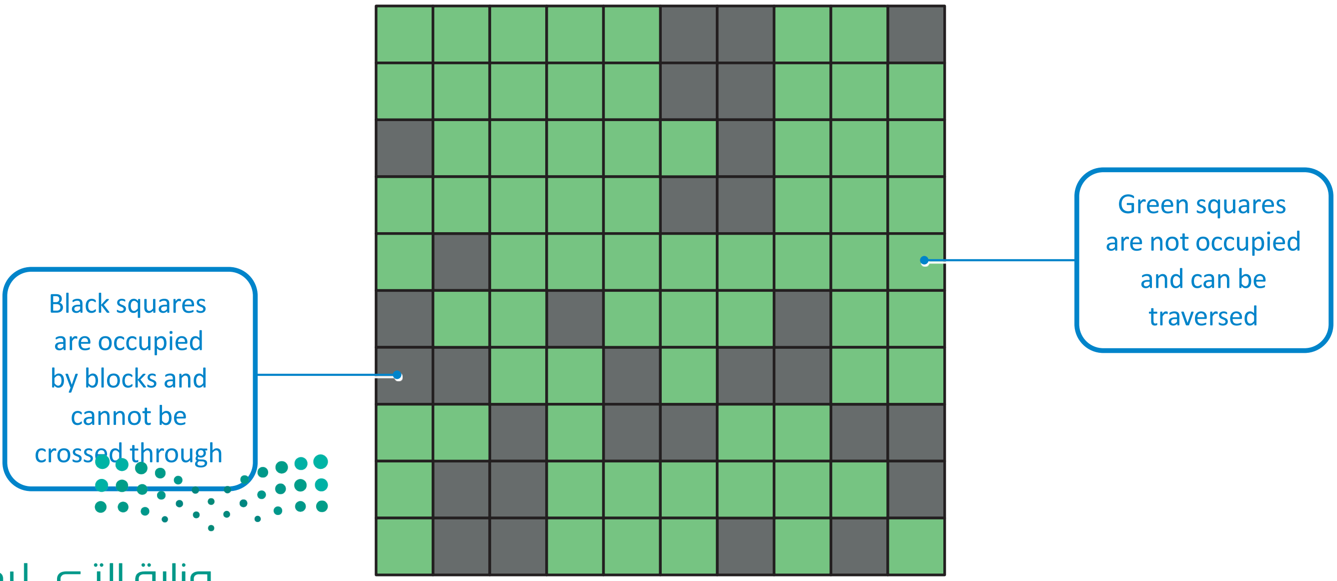
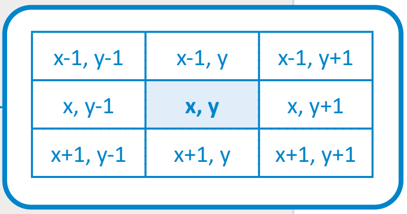

Black squares are occupied by blocks and cannot be crossed through
Green squares are not occupied and can be traversed

Figure 2.19: Visualization of a 10x10 maze with random blocks

Given any such maze, the following function can be used to return a list with all the adjacent accessible, empties and neighbors of a specific cell:

```
def get_accessible_neighbors(maze:np.ndarray, cell:tuple):

    # list of accessible neighbors, initialized to empty
    neighbors=[]

    x,y=cell

    # for each adjacent cell position
    for i,j in [(x-1,y-1),(x-1,y),(x-1,y+1),(x,y-1),(x,y+1),(x+1,y-1),(x+1,y),
(x+1,y+1)]:

        # if the adjacent cell is within the bounds of the grid and is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and
maze[(i,j)]==0:

            neighbors.append(((i,j),1))

    return neighbors
```



This implementation assumes that all possible transitions from a cell to any horizontally, vertically, or diagonally adjacent neighbor have the same cost of 1. This assumption will be revisited later in this lesson, to allow for more complex scenarios with variable transition costs.

The get_accessible_neighbors() function is required by any search algorithm that attempts to solve the maze. The following examples use the small 3x3 maze created above to verify that the function indeed returns the correct neighbors for a given cell.

```
# this cell is the northwest corner of the grid and has only 2 accessible neighbors
get_accessible_neighbors(small_maze, (0,0))
```

```
[((0, 1), 1), ((1, 0), 1)]
```

```
# the starting cell (in the southwest corner) has only 1 accessible neighbor
get_accessible_neighbors(small_maze, (2,0))
```

```
[((1, 0), 1)]
```

Given the ability to create mazes and to also retrieve the neighbors of any cell in a maze, the next step is to implement search algorithms that can solve a maze by finding the shortest path from a given start cell to a given target cell.

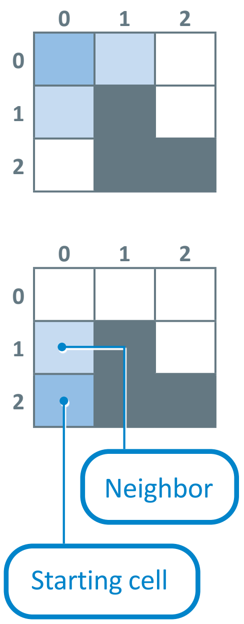


Figure 2.20: Neighbors of cells

Using BFS to Solve Maze Puzzles

The bfs_maze_solver() function described in this section uses Breadth-First-Search to solve maze puzzles with a start and target cell. This implementation utilizes the get_accessible_neighbors() function defined above to retrieve the neighboring cells that can be visited at any point during the search.

Once BFS has found the target cell, the reconstruct_shortest_path() function shown below is used to reconstruct and return the shortest path, working backward from target to start:

```python
def reconstruct_shortest_path(parent:dict, start_cell:tuple, target_cell:tuple):

    shortest_path = []

    my_parent=target_cell # start with the target_cell

    # keep going from parent to parent until the search cell has been reached
    while my_parent!=start_cell:

        shortest_path.append(my_parent) # append the parent

        my_parent=parent[my_parent] # get the parent of the current parent

    shortest_path.append(start_cell) # append the start cell to complete the path

    shortest_path.reverse() # reverse the shortest path

    return shortest_path
```

The same reconstruct_shortest_path( ) function will be used to reconstruct the solution for the A* Search algorithm described later in this lesson. Given the definitions of the get_accessible_ neighbors( ) and reconstruct_shortest_path( ) helper functions, the bfs_maze_solver() function can be implemented as follows:

```python
from typing import Callable # used to call a function as an argument of another function

def bfs_maze_solver(start_cell:tuple,
                    target_cell:tuple,
                    maze:np.ndarray,
                    get_neighbors: Callable,
                    verbose:bool=False): # by default, suppresses descriptive output text

    cell_visits=0 # keeps track of the number of cells that were visited during the search
    visited = set() # keeps track of the cells that have already been visited
    to_expand = [] # keeps track of the cells that have to be expanded

    # add the start cell to the two lists
    visited.add(start_cell)
    to_expand.append(start_cell)
    # remembers the shortest distance from the start cell to each other cell
    shortest_distance = {}
    # the shortest distance from the start cell to itself, zero
    shortest_distance[start_cell] = 0

    # remembers the direct parent of each cell on the shortest path from the start_cell to the cell
    parent = {}
    #the parent of the start cell is itself
    parent[start_cell] = start_cell

    while len(to_expand)>0:

        next_cell = to_expand.pop(0) # get the next cell and remove it from the expansion list

        if verbose:
            print('\nExpanding cell', next_cell)

        # for each neighbor of this cell
        for neighbor,cost in get_neighbors(maze, next_cell):

            if verbose:
                print('\tVisiting neighbor cell',neighbor)

            cell_visits+=1

            if neighbor not in visited: # if this is the first time this neighbor is visited

                visited.add(neighbor)
                to_expand.append(neighbor)
                parent[neighbor]= next_cell
                shortest_distance[neighbor]=shortest_distance[next_cell]+cost

                # target reached
                if neighbor==target_cell:

                    # get the shortest path to the target cell, reconstructed in reverse.
                    shortest_path = reconstruct_shortest_path(parent,
                                                              start_cell, target_cell)

                   return shortest_path, shortest_distance[target_cell],cell_visits

            else:  # this neighbor has been visited before

                # if the current shortest distance to the neighbor is longer than the shortest
                # distance to next_cell plus the cost of transitioning from next_cell to this neighbor
                if shortest_distance[neighbor]>shortest_distance[next_cell]
                                                                     +cost:

                    parent[neighbor]=next_cell
                   shortest_distance[neighbor]=shortest_distance[next_cell]+cost

    # search complete but the target was never reached, no path exists
    return None,None,None
```

The function follows the standard BFS approach of exploring all options at the current depth prior to moving to the next depth level. This implementation uses a set called visited and a list called to_expand.

The first includes all cells that have been visited at least once by the algorithm. The second list includes all the cells that have not yet been expanded, which means that their neighbors have not been visited yet. The algorithm also uses two dictionaries shortest_distance and parent. The first one maintains the length of the shortest path from the start cell to each other cell, while the second one remembers the parent of the cell on this shortest path.

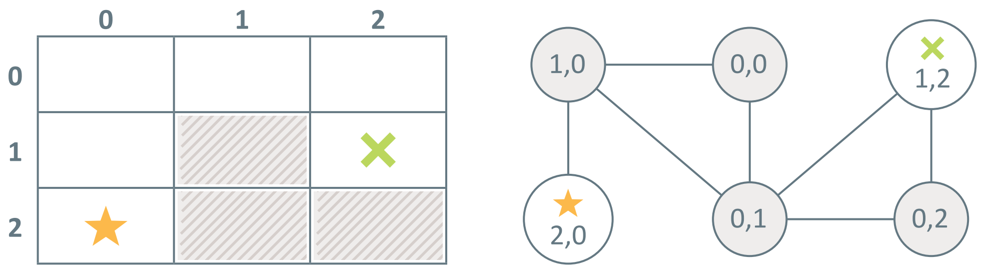
Once the target cell has been reached and the search is complete, shortest_distance[target_cell] will include the length of the solution: the length of the shortest path from start to target.

The following code uses the bfs_maze_solver() function to solve the small 3x3 maze defined above:

```
start_cell=(2,0) # start cell, marked by a star in the 3x3 maze
target_cell=(1,2) # target cell, marked by an "X" in the 3x3 maze

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                               target_cell,
                                               small_maze,
                                               get_accessible_neighbors,
                                               verbose=True)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Expanding cell (2, 0)
        Visiting neighbor cell (1, 0)

Expanding cell (1, 0)
        Visiting neighbor cell (0, 0)
        Visiting neighbor cell (0, 1)
        Visiting neighbor cell (2, 0)

Expanding cell (0, 0)
        Visiting neighbor cell (0, 1)
        Visiting neighbor cell (1, 0)

Expanding cell (0, 1)
        Visiting neighbor cell (0, 0)
        Visiting neighbor cell (0, 2)
        Visiting neighbor cell (1, 0)
        Visiting neighbor cell (1, 2)

Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 10
```

BFS successfully finds the shortest path after 10 cells visits. The search process followed by BFS can be more easily visualized if one considers a graph-based representation of the maze. Consider the following example of a simple 3x3 maze and its graph representation:



The graph representation includes one node for every non-occupied cell. The label on the nodes includes the coordinates of the corresponding matrix cell. There is an undirected edge from one node to another if their corresponding cells are accessible from each other.

One important observation about BFS is that, for **unweighted graphs**, the first path that it finds between the start cell and any other cell is guaranteed to be the one that includes the smallest number of visited cells. This means that, as long as all edges on the graph have the same weight (or, equivalently, that all transitions from one cell to another have the same cost), then the first path found to a specific node is guaranteed to be the shortest path to that node. This is why the bfs_maze_solver() stops the search and returns the result the first time it visits the target node.

However, this approach does not work for **weighted graphs**. Consider the following weighted version of the graph representation for the 3x3 maze:

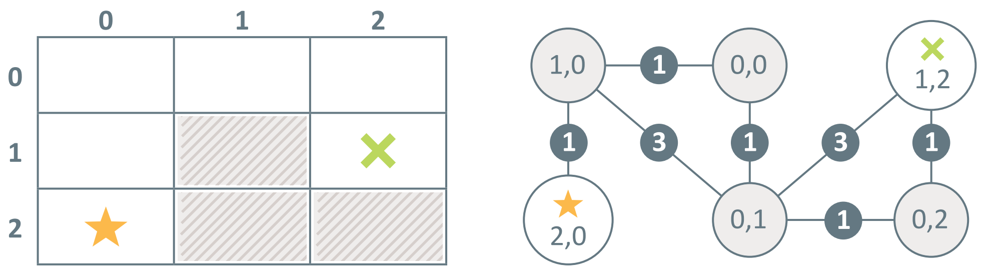


Figure 2.21: Maze and its weighted graph

In this example, all edges that correspond to vertical or horizontal moves (south, north, west, east) have a weight equal to 1. However, all edges that correspond to diagonal moves (southwest, southeast, northwest, northeast), have a weight equal to 3. In this weighted case, the shortest path is clearly [(2,0), (1,0), (0,0), (0,1), (0,2), (1,2)], which has a total distance of 1+1+1+1+1=5.

This more complex scenario can be encoded via the weighted version of the get_accessible_neighbors() function that is described below.

```
def get_accessible_neighbors_weighted(maze:np.ndarray,
                                      cell:tuple,
                                      horizontal_vertical_weight:float,
                                      diagonal_weight:float):
```

```
    neighbors=[]
    x,y=cell

    for i,j in [(x-1,y-1), (x-1,y+1), (x+1,y-1), (x+1,y+1)]: # for diagonal neighbors

        # if the cell is within the bounds of the grid and it is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), diagonal_weight))

    for i,j in [(x-1,y), (x,y-1), (x,y+1), (x+1,y)]: # for horizontal and vertical neighbors

        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), horizontal_vertical_weight))

    return neighbors
```

This function allows the user to assign a custom weight for horizontal and vertical moves, and a different custom weight for diagonal moves. If this weighted version is then used by the BFS solver, the results are as follows:

```
from functools import partial

start_cell=(2,0)
target_cell=(1,2)
horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                               target_cell,
                                               small_maze,
                                               partial(get_accessible_neighbors_weighted,
                                                      horizontal_vertical_weight=horz_vert_w,
                                                       diagonal_weight=diag_w),
                                               verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 7
Number of cell visits: 6
```

As expected, the BFS solver mistakenly reports the exact same path as before, even though it has a distance of 7 and is clearly not the shortest path. This is due to the **uninformed** nature of the BFS algorithm, in which BFS does not take the weights into account when deciding which cell to expand next. It simply applies the same breadth-first approach, which leads to the exact same solution that the algorithm found for the unweighted version of the maze.

The next section describes how this weakness can be addressed via A* Search, an **informed** and more intelligent search algorithm that adjusts its behavior based on the specified weights, and can therefore solve mazes with both weighted and unweighted transitions.

Using A* Search to Solve Maze Puzzles

Similar to BFS, A* Search expands one cell at a time, by visiting each of its accessible neighbors. However, while BFS uses a blind breadth-first approach to decide which cell to expand next, A* Search expands the cell with the smallest estimated distance to the target cell, as computed by a heuristic function.

The exact definition of the heuristic function depends on the application. For maze puzzles, a good heuristic would provide an accurate estimate of "how close" a candidate cell is to the target. As long as the employed heuristic is guaranteed to never **overestimate** the actual distance to the target (i.e. provide an estimate that is higher than the actual distance to the target), then the algorithm is guaranteed to find the shortest possible path for **both weighted and unweighted** graphs. If a heuristic sometimes overestimates distances, then A* Search will still return a solution, but it might not be the best one possible.

The simplest possible heuristic that is guaranteed to never lead to overestimation is a simple function that always produces an estimated distance of 1:

```
def constant_heuristic(candidate_cell:tuple, target_cell:tuple):

    return 1
```

While this is clearly an overly optimistic heuristic, it will never produce an estimate that is higher than the actual distance, and will therefore lead to the best possible solution. A more sophisticated heuristic that finds the best solution much faster will be introduced later in this section.

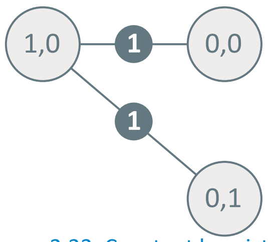


Figure 2.22: Constant heuristic

The following function uses a given heuristic function to find the cell that should be expanded next:

```
def get_best_candidate(expansion_candidates:set,
                       shortest_distance:dict,
                       heuristic:Callable):

    winner = None
    # best (lowest) distance estimate found so far. Initialized to a very large number
    best_estimate= sys.maxsize

    for candidate in expansion_candidates:

       # distance estimate from start to target, if this candidate is expanded next
       candidate_estimate=shortest_distance[candidate]+heuristic(candidate,target_cell)
        if candidate_estimate < best_estimate:
            winner = candidate
            best_estimate=candidate_estimate

    return winner
```

The above implementation utilizes a for loop to iterate over all the candidates in the set and find the best one. A more efficient implementation could use a priority queue that can produce the best candidate without having to iterate over all candidates.

The get_best_candidate() function is used as a helper module by the astar_maze_solver() function presented next. In addition to the heuristic function, this implementation also uses the get_accessible_neighbors_weighted() and reconstruct_shortest_path() helper functions defined in the previous section.

```
import sys

def astar_maze_solver(start_cell:tuple,
                    target_cell:tuple,
                    maze:np.ndarray,
                    get_neighbors: Callable,
                    heuristic:Callable,
                    verbose:bool=False):

    cell_visits=0

    shortest_distance = {}
    shortest_distance[start_cell] = 0

    parent = {}
    parent[start_cell] = start_cell

    expansion_candidates = set([start_cell])

    fully_expanded = set()

    # while there are still cells to be expanded
    while len(expansion_candidates) > 0:

        best_cell = get_best_candidate(expansion_candidates,shortest_distance,heuristic)

        if best_cell == None:  break

        if verbose: print('\nExpanding cell', best_cell)

        # if the target cell has been reached, reconstruct the shortest path and exit
        if best_cell == target_cell:
            shortest_path=reconstruct_shortest_path(parent,start_cell,target_cell)

            return shortest_path, shortest_distance[target_cell],cell_visits

        for neighbor,cost in get_neighbors(maze, best_cell):

            if verbose: print('\nVisiting neighbor cell', neighbor)

            cell_visits+=1

            # first time this neighbor is reached
            if neighbor not in expansion_candidates and neighbor not in fully_expanded:

                expansion_candidates.add(neighbor)

                parent[neighbor] = best_cell # mark the best_cell as this neighbor's parent

                # update the shortest distance for this neighbor
                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

            # this neighbor has been visited before, but a better (shorter) path to it has just been found
            elif shortest_distance[neighbor] > shortest_distance[best_cell] + cost:

                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

                parent[neighbor] = best_cell

                if neighbor in fully_expanded:

                    fully_expanded.remove(neighbor)

                    expansion_candidates.add(neighbor)

        # all neighbors of best_cell have been inspected at this point
        expansion_candidates.remove(best_cell)

        fully_expanded.add(best_cell)

    return None, None, None   # no solution was found
```

Similar to bfs_maze_solver( ), the above function also uses the same two dictionaries shortest_distance and parent to keep the length of the shortest path from the start cell to each other cell and the parent of the cell on this shortest path.

However, astar_maze_solver() follows a different approach to visiting and expanding cells. It uses the expansion_candidates to keep track of all cells that could be expanded next. In every iteration, it uses the get_best_candidate() function to select which of these candidates should be expanded next.

After the best_cell candidate has been selected, a for loop is used to visit all its neighbors. If a neighbor is visited for the first time, then best_cell becomes the neighbor's parent on the shortest path.

The same happens if the neighbor has been visited before, but best_cell offers a shorter path than the one previously found. If such a better path is indeed found, then the neighbor has to go back to the expansion_candidates set, to reevaluate the shortest path to its own neighbors.

The code below utilizes astar_maze_solver() to solve the **unweighted case** of the 3x3 maze puzzle:

```
start_cell=(2,0)
target_cell=(1,2)

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                      target_cell,
                                      small_maze,
                                      get_accessible_neighbors,
                                      constant_heuristic,
                                      verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 12
```

The A* Search solver finds the best possible shortest path after 12 cell visits. This is slightly higher than the BFS solver, which managed to find the solution in only 10 visits. This is due to the simplicity of the constant heuristic that was used to inform astar_maze_solver(). As shown later in this section, a superior heuristic can be used to help the algorithm find the solution faster.

The next step is to evaluate whether A* Search can indeed solve the weighted maze, which BFS failed to find the shortest path for:

```
start_cell=(2,0)
target_cell=(1,2)

horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                   target_cell,
                                   small_maze,
                                   partial(get_accessible_neighbors_weighted,
                                           horizontal_vertical_weight=horz_vert_w,
                                           diagonal_weight=diag_w),
                                   constant_heuristic,
                                   verbose=False)
```

```
print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2)]
Cells on the Shortest Path: 6
Shortest Path Distance: 5
Number of cell visits: 12
```

The results reveal that astar_maze_solver() manages to solve the weighted case by finding the shortest possible path [(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2)], with a total cost of 5. This demonstrates the advantage of using an informed search algorithm, which manages to get the optimal solution even when using the simplest possible heuristic.

Algorithm Comparison

The next step is to compare BFS and A* Search on a larger and more complex maze. The following Python code can be used to create a numeric representation of such a maze:

```
big_maze=np.zeros((15,15))

# coordinates of the cells occupied by blocks
blocks=[(2,8), (2,9), (2,10), (2,11), (2,12),
        (8,8), (8,9), (8,10), (8,11), (8,12),
        (3,8), (4,8), (5,8), (6,8), (7,8),
        (3,12), (4,12), (6,12), (7,12)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    big_maze[block]=1
```

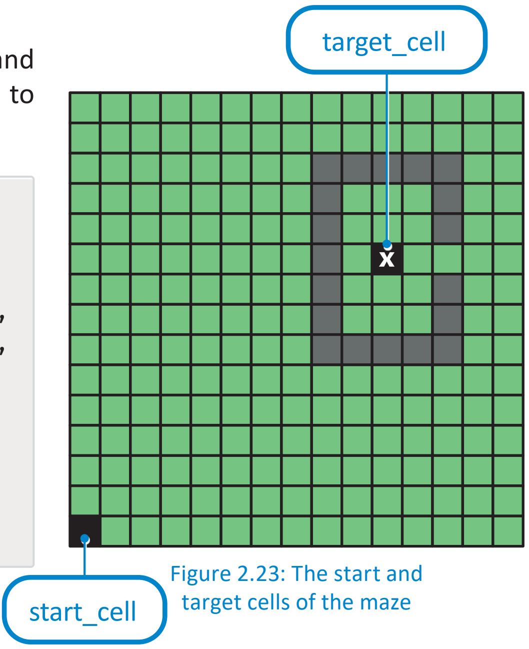


Figure 2.23: The start and target cells of the maze

This 15x15 maze has a C-shaped section of blocks that the player has to circumvent to reach the target marked by the "X". Next, the BFS and A* Search solvers are used to solve both the weighted and unweighted versions of this larger maze:

unweighted version

```
start_cell=(14,0)
target_cell=(5,10)

solution_bfs_unw, distance_bfs_unw, cell_visits_bfs_unw=bfs_maze_solver(start_cell,
                                                        target_cell,
                                                        big_maze,
                                                        get_accessible_neighbors,
                                        verbose=False)

print('\nBFS unweighted.')
print('\nShortest Path:', solution_bfs_unw)
print('Cells on the Shortest Path:', len(solution_bfs_unw))
print('Shortest Path Distance:', distance_bfs_unw)
print('Number of cell visits:', cell_visits_bfs_unw)

solution_astar_unw, distance_astar_unw, cell_visits_astar_unw=astar_maze_solver(
                                        start_cell,
                                        target_cell,
                                        big_maze,
                                        get_accessible_neighbors,
                                        constant_heuristic,
                                        verbose=False)

print('\nA* Search unweighted with a constant heuristic.')
print('\nShortest Path:', solution_astar_unw)
print('Cells on the Shortest Path:', len(solution_astar_unw))
print('Shortest Path Distance:', distance_astar_unw)
print('Number of cell visits:', cell_visits_astar_unw)
```

```
BFS unweighted.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8,
6), (8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (4, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1237

A* Search unweighted with a constant heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (10, 5), (10,
6), (9, 7), (9, 8), (10, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (6, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1272
```

weighted version

```
start_cell=(14,0)
target_cell=(5,10)

horz_vert_w=1
diag_w=3

solution_bfs_w, distance_bfs_w, cell_visits_bfs_w=bfs_maze_solver(start_cell,
                                     target_cell,
                                    big_maze,
                                    partial(get_accessible_neighbors_weighted,
                                            horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    verbose=False)

print('\nBFS weighted.')
print('\nShortest Path:', solution_bfs_w)
print('Cells on the Shortest Path:', len(solution_bfs_w))
print('Shortest Path Distance:', distance_bfs_w)
print('Number of cell visits:', cell_visits_bfs_w)

solution_astar_w, distance_astar_w, cell_visits_astar_w=astar_maze_solver(start_cell,
                                    target_cell,
                                    big_maze,
                                    partial(get_accessible_neighbors_weighted,
                                            horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    constant_heuristic,
                                    verbose=False)

print('\nA* Search weighted with constant heuristic.')
print('\nShortest Path:', solution_astar_w)
print('Cells on the Shortest Path:', len(solution_astar_w))
print('Shortest Path Distance:', distance_astar_w)
print('Number of cell visits:', cell_visits_astar_w)
```

```
BFS weighted.

Shortest Path: [(14, 0), (14, 1), (14, 2), (13, 2), (13, 3), (12, 3), (12,
4), (11, 4), (11, 5), (10, 5), (10, 6), (9, 6), (9, 7), (9, 8), (9, 9), (9,
10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (4, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 30
Number of cell visits: 1235

A* Search weighted with constant heuristic.

Shortest Path: [(14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (9,
1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (9,
10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (5, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1245
```

The results are consistent with the ones reported for the small maze:

- Both BFS and A* Search find the shortest path for the unweighted version.
- BFS finds the solution in fewer visits (1237 vs. 1272 for A* Search).
- BFS fails to find the shortest path for the weighted version, as it reports a path with a distance of 30.
- A* Search finds the shortest path for the weighted version, reporting a path with a distance of 25.

The following code can be used to visualize the shortest path found by the BFS and A* Search algorithms on the weighted version:

```python
maze_bfs_w=big_maze.copy()

for cell in solution_bfs_w:
    maze_bfs_w[cell]=2

plot_maze(maze_bfs_w)
```

```python
maze_astar_w=big_maze.copy()

for cell in solution_astar_w:
    maze_astar_w[cell]=2

plot_maze(maze_astar_w)
```

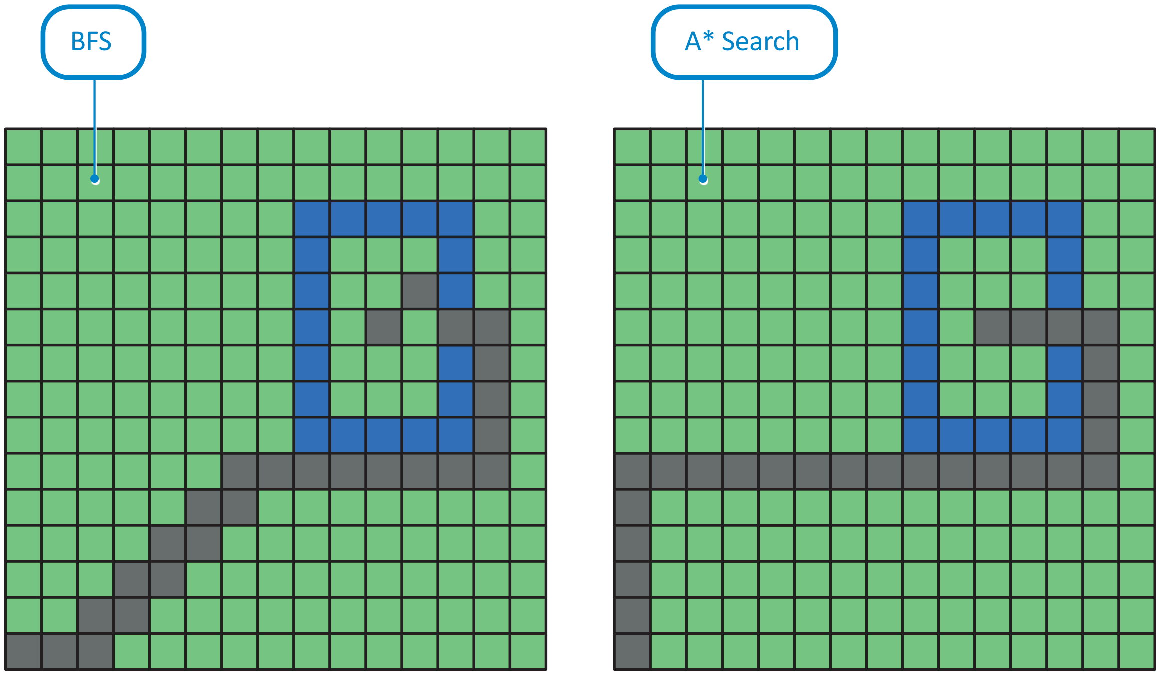


Figure 2.24: Comparison of BFS and A* Search solutions

The visualizations verify that the informed nature of A* Search allows it to avoid diagonal moves, as they have a higher cost than horizontal and vertical ones. On the other hand, the uninformed BFS ignores the cost of each move and reports a much more expensive solution. A general comparison of uninformed and informed algorithms is seen in table 2.6:

Table 2.6: Comparison of uninformed and informed algorithms

Comparison criteria	Uninformed	Informed
Computational complexity	They are more computationally complex.	Their computational cost is lower.
Efficiency	They are slower than informed algorithms.	They perform searches quicker.
Performance	Impractical for solving large-scale search problems.	Better at handling large-scale search problems.
Effectiveness	The optimal solution is achieved.	Generally, adequate solutions are accepted.

Still, the results showed that BFS could find the optimal solution faster (with fewer cell visits) in the unweighted case. This can be addressed by providing A* Search with a smarter heuristic. A popular heuristic in distance-based applications is the Manhattan Distance, defined as the sum of the absolute differences between the coordinates of the two given points. An example is shown in the figure below:

Manhattan Distance

```
Manhattan (A, B) = |x1-x2| + |y1-y2|
```

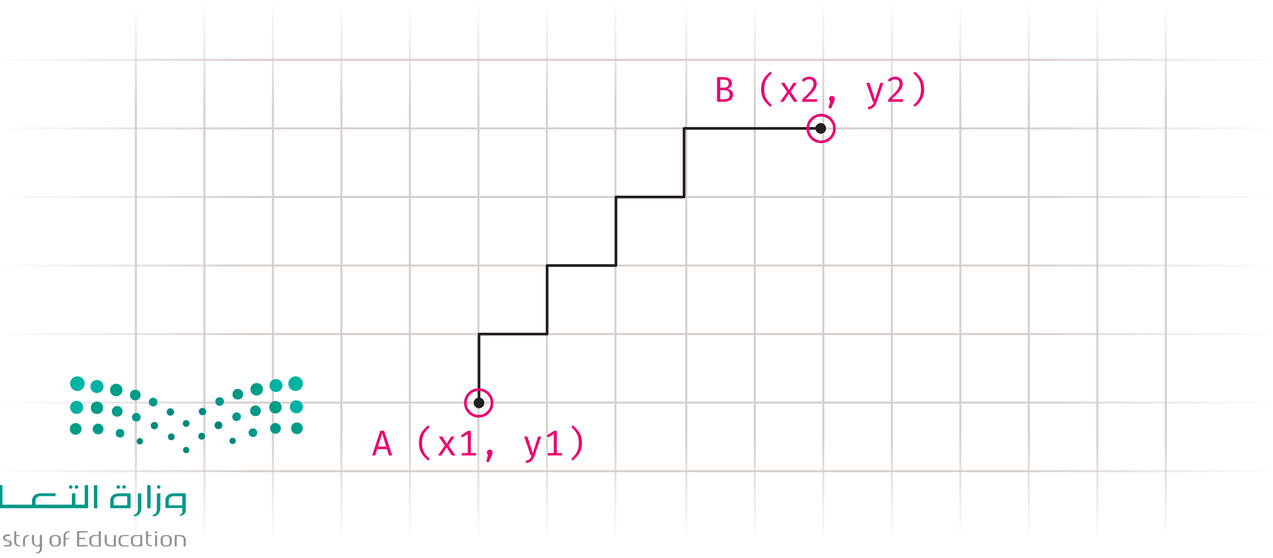


Figure 2.25: Manhattan distance

This can be easily implemented as a python function as follows:

```
def manhattan_heuristic(candidate_cell:tuple,target_cell:tuple):

    x1,y1=candidate_cell
    x2,y2=target_cell
    return  abs(x1 - x2) + abs(y1 - y2)
```

The following code can be used to test if this smarter heuristic can be used to help astar_maze_solver() search the space much faster for both weighted and unweighted scenarios:

```
start_cell=(14,0)
target_cell=(5,10)

solution_astar_unw_mn, distance_astar_unw_mn, cell_visits_astar_unw_mn=astar_
maze_solver(start_cell,
            target_cell,
            big_maze,
            get_accessible_neighbors,
            manhattan_heuristic,
            verbose=False)

print('\nA* Search unweighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_unw_mn)
print('Cells on the Shortest Path:', len(solution_astar_unw_mn))
print('Shortest Path Distance:', distance_astar_unw_mn)
print('Number of cell visits:', cell_visits_astar_unw_mn)

horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution_astar_w_mn, distance_astar_w_mn, cell_visits_astar_w_mn=astar_maze_
solver(start_cell,
       target_cell,
       big_maze,
       partial(get_accessible_neighbors_weighted,
               horizontal_vertical_weight=horz_vert_w,
               diagonal_weight=diag_w),
       manhattan_heuristic,
       verbose=False)

print('\nA* Search weighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_w_mn)
print('Cells on the Shortest Path:', len(solution_astar_w_mn))
print('Shortest Path Distance:', distance_astar_w_mn)
print('Number of cell visits:', cell_visits_astar_w_mn)
```

```
A* Search unweighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8,
6), (8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (5, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 865

A* Search weighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (14, 1), (13, 1), (12, 1), (12, 2), (12, 3), (12,
4), (12, 5), (12, 6), (12, 7), (11, 7), (11, 8), (10, 8), (9, 8), (9, 9),
(9, 10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (5, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1033
```

The results verify that the Manhattan Distance heuristic can indeed help A* Search find the shortest possible paths with a significantly lower number of cell visits for both weighted and unweighted scenarios. In fact, the use of this more intelligent heuristic led to a significantly lower visit number than the one required for the BFS algorithm.

The table 2.7 summarizes the results for the different algorithm variants on the big maze:

Table 2.7: Comparison of algorithms performance

	BFS	A* Search with Constant Heuristic	A* Search with Manhattan Heuristic
weighted	dist=30, 1235 visits	dist=25, 1245 visits	dist=25, 1033 visits
unweighted	dist=18, 1237 visits	dist=18, 1272 visits	dist=18, 865 visits

The table demonstrates the advantages of using increasingly more intelligent algorithms to solve search-based problems like the one presented in this lesson:

- Switching from an uninformed (BFS) to an informed (A* Search) search algorithm delivered better results and allowed for the solution of more complex problems.
- The intelligence of informed search algorithms can be further increased by using better heuristics that allow them to find the optimal solution significantly faster.

Exercises

1 Identify two applications of search algorithms.

2 Identify a difference between uninformed and informed search algorithms and mention an example of each algorithm.

3 Explain briefly how the A* algorithm works.

4 Modify your code by changing the diagonal weight from 3 to 1.5. What do you observe? Does the shortest path change for the cases of BFS and A* Search?

5 Modify your code by swapping the starting cell with the target cell coordinates. What do you observe? Is the path the same as before for the weighted cases of BFS and A* Search?

Project

1 Modify the code of the weighted BFS and A* Search algorithms by changing the horizontal and vertical weights to 3 and the diagonal weights to 5. Also change the starting point to (7, 2).

2 What is the new shortest distance path and the number of cell visits of the unweighted versions of the BFS and A* Search algorithms with the constant heuristic function? Find these values and present your observations.

3 Follow the same steps for the weighted versions of the BFS and A* Search algorithms with the constant heuristic function.

4 Repeat the process for the unweighted and weighted versions of the A* Search algorithms with the manhattan heuristic function.

Wrap up

Now you have learned to:

> Employ recursion to solve problems.

> Apply advanced graph traversing algorithms.

> Implement both simple and advanced rule-based systems.

> Design an AI model.

> Measure the effectiveness of your AI model.

> Use search algorithms to solve simulations of real-life problems.

KEY TERMS

A* Search	Informed Search	Rule-Based Systems
Algorithm Performance	Knowledge Base	Scoring Function
Breadth-First Search (BFS)	Maze Solving	Search Algorithms
Confusion Matrix	Model Training	Uninformed Search
Depth-First Search (DFS)	Path Finding	Unweighted Graph
Heuristic Function	Recursion	Weighted Graph

3. Natural Language Processing (NPL)

In this unit, you will learn an end-to-end process of training a supervised and an unsupervised learning model for understanding the sentiment of a given piece of text. At the end, you will learn how machine learning can be used to support applications related to Natural Language Processing (NLP).

Learning Objectives

In this unit, you will learn to:

> Define supervised learning.

> Train a supervised learning model to understand text.

> Define unsupervised learning.

> Train an unsupervised learning model to understand text.

> Create a simple chatbot.

> Generate text using the Natural Language Processing (NLP) techniques.

Tools

> Jupyter Notebook

Lesson 1

Supervised Learning

Link to digital lesson

www.ien.edu.sa

Using Supervised Learning to Understand Text

Natural Language Processing (NLP) is a field of AI that focuses on enabling computers to understand, interpret, and generate human language. NLP is concerned with tasks such as text classification, sentiment analysis, machine translation, and question-answering. This lesson will focus specifically on how supervised learning, one of the main types of machine learning (ML), can be used to automatically understand and make useful predictions about a text's properties.

You already learned in unit 1 that AI is an umbrella term that includes Machine Learning and Deep Learning, as you can see in figure 3.1. AI is a broad field of computer science that focuses on creating intelligent machines, while machine learning is a subset of AI that focuses on building algorithms and models that allow machines to learn from data without being explicitly programmed.

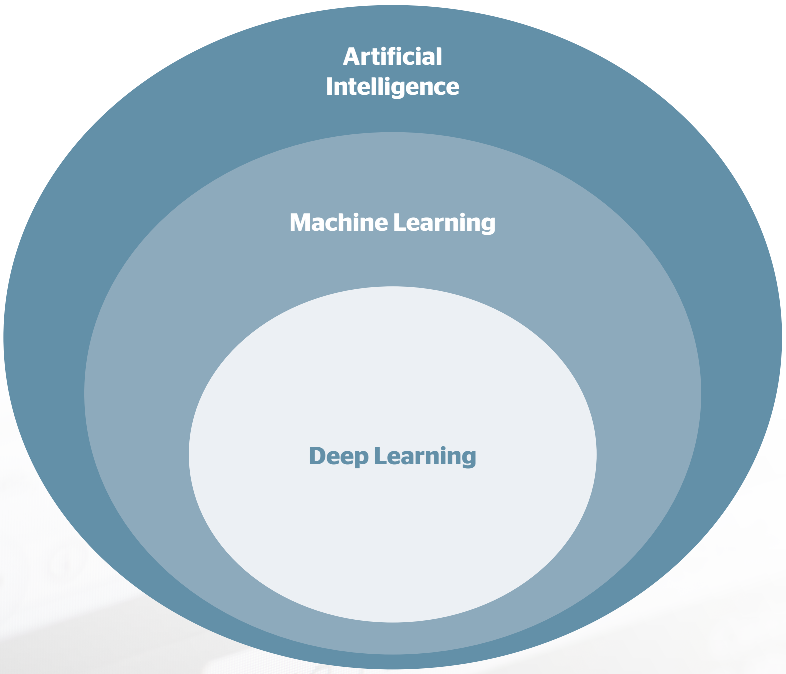


Figure 3.1: Fields under the AI umbrella

Deep learning

Deep learning is a type of machine learning that uses deep neural networks to automatically learn from large amounts of data. It allows computers to recognize patterns and make decisions in a more humanlike way, by building complex models of the data.

Machine Learning

Machine learning is a subfield of AI that focuses on developing algorithms that enable computers to learn from data, rather than following explicit programming instructions. It involves training computer models to recognize patterns and make predictions based on input data, allowing the model to improve its accuracy over time. This allows machines to perform tasks such as classification, regression, clustering, and recommendation, without being explicitly programmed for each task.

Machine learning can be broadly categorized into three main types:

Supervised learning a type of machine learning where the algorithm learns from labeled training data, with the goal of making predictions on new data, not present in the training or test sets, as shown in figure 3.2. Examples:

- Image classification (e.g. recognizing objects in photos)
- Fraud detection (e.g. identifying suspicious financial transactions)
- Spam filtering (e.g. identifying unwanted email messages)

Unsupervised learning a type of machine learning where the algorithm works with unlabeled data, trying to find patterns and relationships in the data. Examples:

- Anomaly detection (e.g. detecting unusual patterns in data)
- Clustering (e.g. grouping similar data points together)
- Dimensionality reduction (e.g. selecting the dimensions that reduce data complexity)

Reinforcement learning a type of machine learning where an agent interacts with its environment and learns by trial and error, receiving rewards or punishments for its actions. Examples:

- Game playing (e.g. playing chess or Go)
- Robotics (e.g. teaching a robot to navigate its environment)
- Resource allocation (e.g. optimizing resource usage in a network)

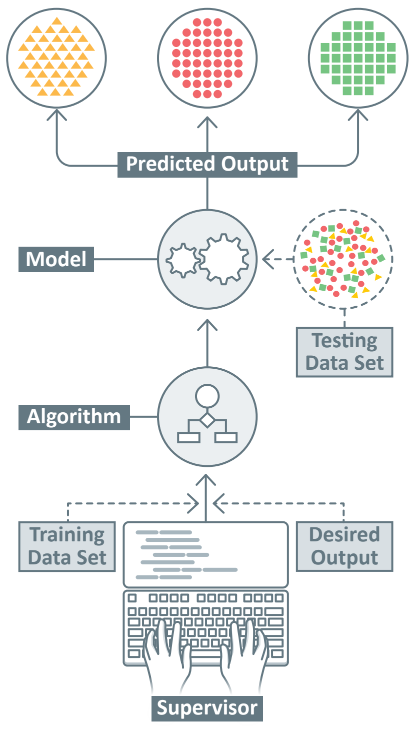


Figure 3.2: Supervised learning representation

Here is table 3.1 summarizing the advantages and disadvantages of each type of machine learning:

Table 3.1: Advantages and disadvantages of Machine Learning types

Advantages	Disadvantages
Supervised Learning	
• Well-established and widely used. • Easy to understand and implement. • Can handle both linear and non-linear data.	• Requires labeled data, which can be expensive to obtain. • Limited to the task it was trained for, and may not generalize well to new data. • Difficult to adapt to other problems if the model is too complex.
Unsupervised Learning	
• Does not require labeled data, making it more flexible. • Can discover hidden patterns in data. • Can handle high-dimensional and complex data.	• Harder to understand and interpret than supervised learning. • Limited to exploratory analysis, and may not be suitable for decision-making tasks. • Difficult to adapt to other problems if the model is too complex.
Reinforcement Learning	
• Flexible, and can handle complex and dynamic environments. • Can learn from experience and improve over time. • Suitable for decision-making tasks, such as game playing and robotics.	• More complex than supervised or unsupervised learning. • Challenging to design reward functions that accurately capture the desired behavior. • May require large amounts of training data and computational resources.

Supervised Learning

Supervised Learning is a type of ML that involves the use of labeled data to train an algorithm to make predictions. The algorithm is trained on a labeled dataset and then tested on an unseen dataset. Supervised learning is commonly used in NLP for tasks such as text classification, sentiment analysis, and named entity recognition. In these tasks, the algorithm is trained on a labeled dataset where each example is labeled with the correct category or sentiment. If the labels are numeric, then the supervised learning task is referred to as "regression". If the labels are discrete, the task is referred to as "classification".

Supervised Learning

In supervised learning, you use manually curated and labeled datasets to train computer algorithms to predict new values.

Regression

For instance, regression can focus on predicting the sale price of a house based on its size, location, and number of bedrooms. It can also be used to predict the demand for a product based on historical sales data and advertising expenditure. In an NLP context, regression can use the available text to predict the sentiment score of a movie review or the popularity of a social media post.

Classification

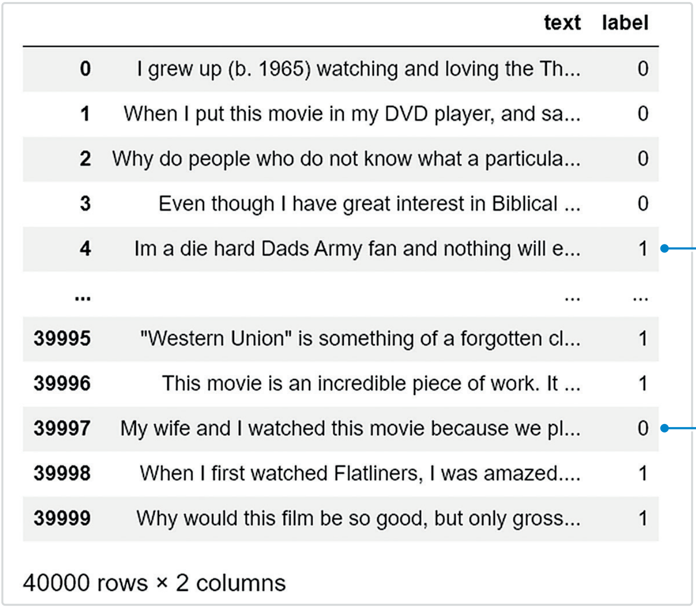
Classification, on the other hand, can be used in applications such as diagnosing a medical condition based on symptoms and test results. When it comes to understanding text, supervised learning can be used to classify or predict categories or labels based on the words and phrases within a document. For example, a supervised learning model might be trained to classify an email as spam or not spam based on the words and phrases used in the email. Another popular application is sentiment classification, which focuses on predicting whether the overall sentiment of a given document is negative or positive. This application is used as a working example in this unit, to demonstrate all the steps in the end-to-end process of building and using a supervised learning model.

In this unit you will use a dataset of movie reviews from the popular website IMDb.com. The dataset has already been split into two parts, one to be used for training the model and one to be used for testing. To load the data into a DataFrame, you will use the Pandas Python library that you have used before. The Pandas library is a popular tool for manipulating spreadsheet data. The following code is used to import the library into your program and then load the two datasets:

```
%%capture  # capture is used to suppress the installation output.

# install the pandas library, if it is missing.
!pip install pandas
import pandas as pd
```

Pandas is a popular library used to read and process spreadsheet-like data.

```python
# load the train and testing data.
imdb_train_reviews=pd.read_csv('imdb_data/imdb_train.csv')
imdb_test_reviews=pd.read_csv('imdb_data/imdb_test.csv')

imdb_train_reviews
```



	text	label
0	I grew up (b. 1965) watching and loving the Th...	0
1	When I put this movie in my DVD player, and sa...	0
2	Why do people who do not know what a particula...	0
3	Even though I have great interest in Biblical ...	0
4	Im a die hard Dads Army fan and nothing will e...	1
...	...	...
39995	"Western Union" is something of a forgotten cl...	1
39996	This movie is an incredible piece of work. It ...	1
39997	My wife and I watched this movie because we pl...	0
39998	When I first watched Flatliners, I was amazed....	1
39999	Why would this film be so good, but only gross...	1

40000 rows × 2 columns

Figure 3.3: Labelled training dataset

As you can see in figure 3.3, the DataFrame dataset has two columns:

- text review.
- label.

A "0" label represents the negative review, while a "1" label represents the positive one.

The next step is to assign the text and label columns to separate variables, from the training and testing examples in the DataFrame dataset:

```python
# extract the text from the 'text' column for both training and testing.
X_train_text=imdb_train_reviews['text']
X_test_text=imdb_test_reviews['text']

# extract the labels from the 'label' column for both training and testing.
Y_train=imdb_train_reviews['label']
Y_test=imdb_test_reviews['label']
X_train_text  # training data in text format
```

The X, Y notations are typically used in supervised learning to represent the input data used to make the prediction (X) and the target labels (Y).

```
0        I grew up (b. 1965) watching and loving the Th...
1        When I put this movie in my DVD player, and sa...
2        Why do people who do not know what a particula...
3        Even though I have great interest in Biblical ...
4        Im a die hard Dads Army fan and nothing will e...
                              ...
39995    "Western Union" is something of a forgotten cl...
39996    This movie is an incredible piece of work. It ...
39997    My wife and I watched this movie because we pl...
39998    When I first watched Flatliners, I was amazed....
39999    Why would this film be so good, but only gross...
Name: text, Length: 40000, dtype: object
```

Figure 3.4: Snapshot of the training examples (X_train_text) from the DataFrame dataset.

Data Preparation and Pre-Processing

Even though this raw text format as in figure 3.5 is intuitive to the human reader, it is unusable by supervised learning algorithms. Instead, algorithms require such documents to be converted into a numeric vector format. The vectorization process can be implemented in multiple different methods, and it has a great impact on the performance of the trained model.

Sklearn Library

The supervised model will be built with sklearn (also known as "scikit-learn"), a popular Python library for machine learning. It provides a range of tools and algorithms for tasks such as classification, regression, clustering, and dimensionality reduction. One useful tool within sklearn is the CountVectorizer, which can be used to preprocess and vectorize text data.

CountVectorizer

The CountVectorizer converts a collection of text documents into a matrix of token counts, where each row represents a document and each column represents a particular token. Tokens can be individual words, phrases or even more complex constructs that capture various patterns in the underlying text data. The entries in the matrix indicate the number of times each token appears in each document. This is also known as "bag-of-words" (BoW) representation, as the order of the words is not preserved and only the counts of the words are retained. Even though the BoW representation is an oversimplification of human language, it can achieve very competitive results in practice.

Vectorization

Vectorization is the process of converting strings of words or phrases (text) to a corresponding vector of real numbers, that is used to encode properties of the text using a format that ML algorithms can understand.

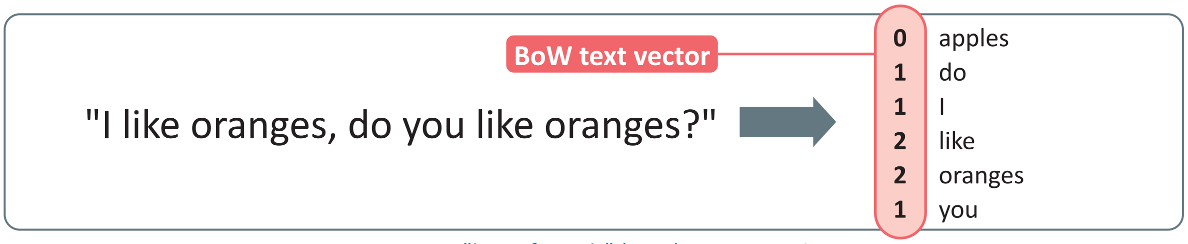


Figure 3.5: "bag-of-words" (BoW) representation

The following code uses the CountVectorizer tool to vectorize the IMDb training dataset:

```
from sklearn.feature_extraction.text import CountVectorizer

# the min_df parameter is used to ignore terms that appear in less than 10 reviews.
vectorizer_v1 = CountVectorizer(min_df=10)

vectorizer_v1.fit(X_train_text) # fit the vectorizer on the training data.
# use the fitted vectorizer to vectorize the data.
X_train_v1 = vectorizer_v1.transform(X_train_text)

X_train_v1
```

```
<40000x23392 sparse matrix of type '<class 'numpy.int64'>'
	with 5301561 stored elements in Compressed Sparse Row format>
```

```
# expand the sparse data into a sparse matrix format, where each column represents a different word.
X_train_v1_dense=pd.DataFrame(X_train_v1.toarray(),
                  columns=vectorizer_v1.get_feature_names_out())
X_train_v1_dense
```

	00	000	007	01	02	04	05	06	07	08	...	zoo	zoom	zooming	zooms	zorro	zu	zucco	zucker	zulu	über
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
39995	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
39996	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
39997	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
39998	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
39999	0	2	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

40000 rows × 23392 columns

Figure 3.6: Vectorizing the training dataset

This "dense" matrix format represents the 40,000 reviews in the training data. It also has a column for each of the words that appear in at least 10 reviews (enforced via the min_df parameter). As can be seen above, this creates a total of 23,392 columns, sorted in alphanumeric order. The matrix entry in position [i,j] represents the number of times that the j_th word appears in the i_th review.

Even though this matrix could directly be used by a supervised learning algorithm, it is highly inefficient in terms of memory usage. This is due to the fact that the vast majority of the entries in this matrix are equal to 0. This happens because only a very small percentage of the 23,392 possible words will actually appear in each review. To address this inefficiency, the CountVectorizer tool stores the vectorized data in a sparse format, which only remembers the non-zero entries in each column.

The code below uses the getsizeof() function, which returns the size of a Python object in bytes, to demonstrate the memory savings of the sparse format for the IMDb data:

```
from sys import getsizeof
print('\nMegaBytes of RAM memory used by the raw text format:',
      getsizeof(X_train_text)/1000000)
print('\nMegaBytes of RAM memory used by the dense matrix format:',
      getsizeof(X_train_v1_dense)/1000000)
print('\nMegaBytes of RAM memory used by the sparse format:',
      getsizeof(X_train_v1)/1000000)
```

```
MegaBytes of RAM memory used by the raw text format: 54.864133

MegaBytes of RAM memory used by the dense matrix format: 7485.440144

MegaBytes of RAM memory used by the sparse format: 4.8e-05
```

As expected, the sparse format requires far less memory, more specifically 0.000048 megabytes. The dense matrix occupies 7 gigabytes. This matrix will not be used again and can thus be deleted to free up this significant amount of memory:

```
# delete the dense matrix.
del X_train_v1_dense
```

Build a Prediction Pipeline

Now that the training data has been vectorized, the next step is to build a first prediction pipeline. One example of a classifiers to use for document prediction is a Naive Bayes classifier. The Naive Bayes classifier uses the probabilities of certain words or phrases occurring in a document to predict the likelihood of the document belonging to a certain class. The "naive" part of the name comes from the assumption that the presence of a particular word in a document is independent of the presence of any other word. This is a strong assumption, but it allows the algorithm to be trained very quickly and effectively.

Classifier

In ML, a classifier is a model that is used to distinguish data points into different categories or classes. The goal of a classifier is to learn from labeled training data, and then make predictions about the class label for new data.

The following code uses the implementation of the Naive Bayes Classifier (MultinomialNB) from the sklearn library to train a supervised learning model on the vectorized IMDb training data:

```
from sklearn.naive_bayes import MultinomialNB

model_v1=MultinomialNB() # a Naive Bayes Classifier

model_v1.fit(X_train_v1, Y_train) # fit the classifier on the vectorized training data.

from sklearn.pipeline import make_pipeline

# create a prediction pipeline: first vectorize using vectorizer_v1, then use model_v1 to predict.
prediction_pipeline_v1 = make_pipeline(vectorizer_v1, model_v1)
```

For example, this code will produce a result array with the first element being "1" for a positive review and "0" for a negative review:

```
prediction_pipeline_v1.predict(['One of the best movies of the year. Excellent
                                 cast and very interesting plot.',
                                'I was very disappointed with his film. I
                                 lost all interest after 30 minutes' ])
```

```
array([1, 0], dtype=int64)
```

The pipeline correctly predicts a positive and negative label for first and second review, respectively. The built-in function predict_proba() can be used to obtain the probabilities that the pipeline assigns to each of the two possible labels. The first element is the probability that "0" will be assigned and the second element is the probability that "1" will be assigned:

```
prediction_pipeline_v1.predict_proba(['One of the best movies of the year. Ex
                                       cellent cast and very interesting plot.',
                                       'I was very disappointed with his film.
                                        I lost all interest after 30 minutes' ])
```

```
array([[0.08310769, 0.91689231],
       [0.83173475, 0.16826525]])
```

The model is 8.3% certain the first review is negative and 91.6% certain it is positive. Likewise, it is 83.1% certain the second review is negative and 16.8% certain it is positive.

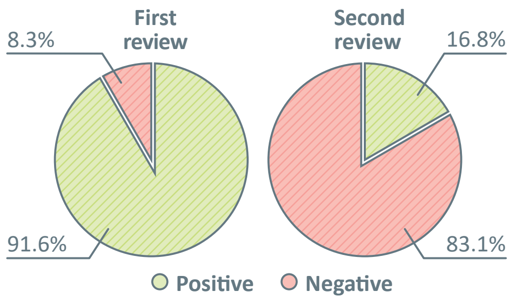


Figure 3.7: Pie charts showing the review percentages

The next step is to test the accuracy of this new pipeline on the reviews in the IMDb testing set. The output is an array all the result labels for the review given in the test data:

```
# use the pipeline to predict the labels of the testing data.
predictions_v1 = prediction_pipeline_v1.predict(X_test_text) # vectorize the text
data, then predict.

predictions_v1
```

```
array([0, 0, 0, ..., 0, 0, 0], dtype=int64)
```

Python provides multiple tools to analyze and visualize the results of classification pipelines. Examples include the accuracy_score() function from sklearn and the "confusion matrix" visualization from the scikit-plot library. There are also other evaluation metrics such as precision, recall, specificity, sensitivity, F1 score, depending on the use case, which can be computed from the confusion matrix. The following output is an approximation of how accurate the prediction was:

```
from sklearn.metrics import accuracy_score
accuracy_score(Y_test, predictions_v1) # get the achieved accuracy.
```

```
0.8468
```

```
%%capture
!pip install scikit-plot; # install the scikit-plot library, if it is missing.
import scikitplot; # import the library

class_names=['neg','pos'] # pick intuitive names for the 0 and 1 labels.

# plot the confusion matrix.
scikitplot.metrics.plot_confusion_matrix(
                                [class_names[i] for i in Y_test],
                                [class_names[i] for i in predictions_v1],
                                 title="Confusion Matrix", # title to use
                                 cmap="Purples", # color palette to use
                                 figsize=(5,5) # figure size
                                 );
```

actual labels

predicted labels

The confusion matrix contains the counts of actual vs. predicted classifications. In a binary classification task (i.e.a problem with two labels, such as the IMDb task), the confusion matrix will have four cells:

True Negatives (upper left):
the number of times the classifier correctly predicted the negative class.

False Negatives (upper right):
the number of times the classifier incorrectly predicted the negative class.

False Positives (lower left):
the number of times the classifier incorrectly predicted the positive clas.

True Positives (lower right):
the number of times the classifier correctly predicted the positive class.

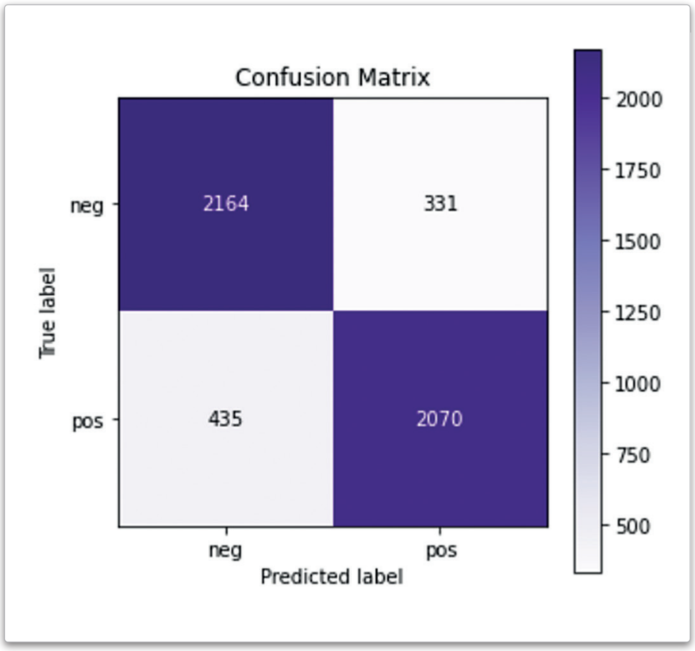


Figure 3.8: Confusion matrix results of the Naive Bayes classifier on the testing data using the IMDb dataset.

The results reveal that even though this first pipeline achieves a competitive accuracy of 84.68%, it still misclassifies hundreds of reviews. You have 331 incorrect predictions in the upper right quarter and 435 incorrect predictions in the lower left corner. This totals 766 incorrect predictions. The first step toward improving performance is to study the behavior of the prediction pipeline, in order to reveal how it processes and understands text.

Accuracy

Accuracy is the ratio of correct predictions to the total number of prediction.

$$\text{Accuracy} = \frac{(\text{True Positives} + \text{True Negatives})}{(\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives})}.$$

Explaining Black-Box Predictors

The Naive Bayes Classifier uses simple mathematical formulas to combine the probabilities of thousands of words and deliver its predictions. Despite its simplicity, it is still unable to deliver an intuitive, user-friendly explanation of exactly how it predicts a positive or negative label for a specific piece of text.

Compare that to decision tree classifiers which are more intuitive, as they represent the learned decision rules in a tree like structure, making it easier for people to understand how the classifier arrived at its predictions. The tree structure also allows for a visual representation of the decisions being made at each branch, which can be useful in understanding the relationships between input features and the target variable.

The lack of explainability is an even bigger challenge for more complex algorithms, such as those based on ensembles (combinations of multiple algorithms) or neural networks. Without explainability, supervised learning algorithms are reduced to black-box predictors: even though they understand the text well enough to predict its label, they are unable to communicate how they make their decisions.

A significant amount of research has been devoted to addressing this challenge by designing explainability methods that can interpret black-box models. One of the most popular methods is LIME (Local Interpretable Model-Agnostic Explanations).

LIME (Local Interpretable Model-Agnostic Explanations)

LIME is a method for explaining the predictions made by black-box models. It does this by looking at one data point at a time and making small changes to it to see how it affects the model's prediction. LIME then uses this information to train a simple and understandable model, such as a linear regression, to explain the prediction. For text data, LIME identifies the words or phrases that have the biggest impact on the prediction. A Python implementation is shown below:

```
%%capture

!pip install lime # install the lime library, if it is missing
from lime.lime_text import LimeTextExplainer

# create a local explainer for explaining individual predictions
explainer_v1 = LimeTextExplainer(class_names=class_names)

# an example of an obviously negative review
easy_example='This movie was horrible. The actors were terrible and the plot
was very boring.'

# use the prediction pipeline to get the prediction probabilities for this example
print(prediction_pipeline_v1.predict_proba([easy_example]))
```

```
[[0.99874831 0.00125169]]
```

As expected, the predictor delivers a very confident negative prediction for this easy example.

```
# explain the prediction for this example.
exp = explainer_v1.explain_instance(easy_example.lower(),
                                    prediction_pipeline_v1.predict_proba,
                                    num_features=10)
# print the words with the strongest influence on the prediction.
exp.as_list()
```

```
[('terrible', -0.07046118794796816),
 ('horrible', -0.06841672591649835),
 ('boring', -0.05909016205135171),
 ('plot', -0.024063095577996376),
 ('was', -0.014436071624747861),
 ('movie', -0.011956911011210977),
 ('actors', -0.011682594571408675),
 ('this', -0.009712387273986628),
 ('very', 0.008956707731803237),
 ('were', -0.008897098392433257)]
```

The score of each word represents a coefficient in the simple linear regression model that was used to deliver the explanation.

Focus the explainer on the 10 most influential features.

A more visual representation can be obtained as follows:

```
# visualize the impact of the most influential words.
fig = exp.as_pyplot_figure()
```

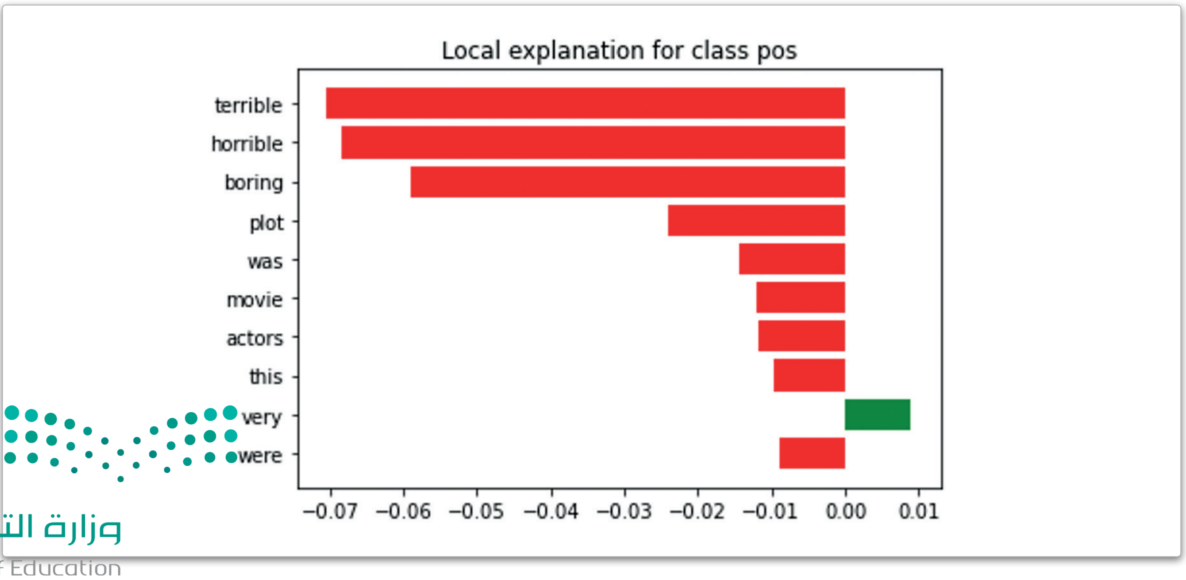
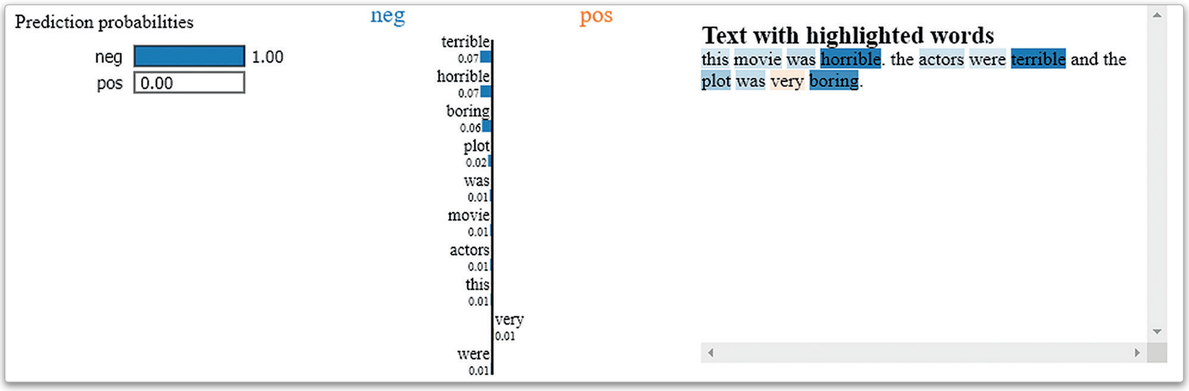


Figure 3.9: The words with the highest influence on the prediction

A negative coefficient increases the probability of the negative class, while a positive coefficient decreases it. For instance, the words 'horrible', 'terrible', and 'boring' had the strongest impact on the model's decision to predict a negative label. The word 'very' slightly pushed the model in a different (positive) direction, but it was not nearly enough to change the decision. To a human observer, it might look strange that sentiment-free words such as 'plot' or 'was' seem to have relatively high coefficients. However, it is important to remember that machine learning does not always follow human common sense. These high coefficients may indeed reveal flaws in the algorithm's logic and could be responsible for some of the predictor's mistakes. Alternatively, the coefficients may be indicative of latent but informative predictive patterns. For instance, it may indeed be the case that human reviewers are more likely to use the word 'plot' or use past tense ('was') when speaking in a negative context. The LIME Python library can also visualize the explanations in other ways.

For example:

```
exp.show_in_notebook()
```



this movie was horrible. the actors were terrible and the plot was very boring.

Figure 3.10: Othervisualrepresentations

The review used in the previous example was obviously negative and easy to predict. Consider the following more challenging review which can confuse the algorithm, taken from the testing set of the IMDb data:

```
# an example of a positive review that is mis-classified as negative by prediction_pipeline_v1
mistake_example= X_test_text[4600]
mistake_example
```

```
"I personally thought the movie was pretty good, very good acting by
Tadanobu Asano of Ichi the Killer fame. I really can't say much about the
story, but there were parts that confused me a little too much, and overall
I thought the movie was just too lengthy. Other than that however, the
movie contained superb acting great fighting and a lot of the locations
were beautifully shot, great effects, and a lot of sword play. Another
solid effort by Tadanobu Asano in my opinion. Well I really can't say
anymore about the movie, but if you're only outlook on Asian cinema is
Crouching Tiger Hidden Dragon or House of Flying Daggers, I would suggest
you trying to rent it, but if you're a die-hard Asian cinema fan I would
say this has to be in your collection very good Japanese film."
```

```
# get the correct labels of this example.
print('Correct Label:', class_names[Y_test[4600]])

# get the prediction probabilities for this example.
print('Prediction Probabilities for neg, pos:',
      prediction_pipeline_v1.predict_proba([mistake_example]))
```

```
Correct Label: pos
Prediction Probabilities for neg, pos: [[0.8367931 0.1632069]]
```

Even though this is clearly a positive review, the pipeline reported a very confident negative prediction with a probability of 83%. The explainer can now be used to provide insight into why the predictor made this erroneous decision:

```
# explain the prediction for this example.
exp = explainer_v1.explain_instance(mistake_example, prediction_pipeline_
v1.predict_proba, num_features=10)

# visualize the explanation.
fig = exp.as_pyplot_figure()
```

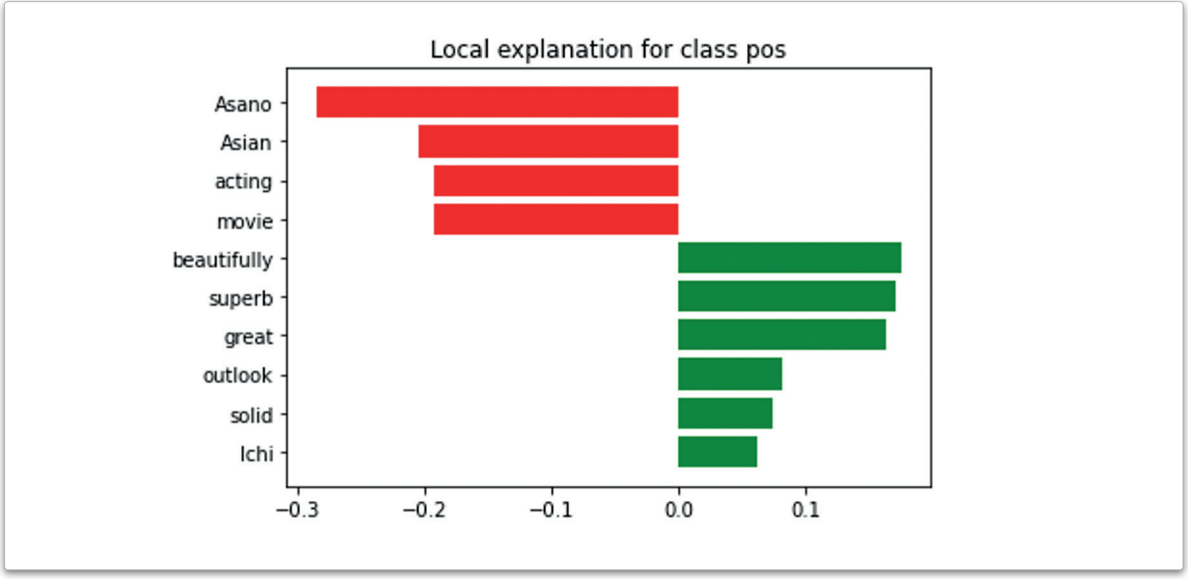


Figure 3.11: Words that influenced the erroneous decision

Even though the predictor correctly captures the positive influence of certain words such as 'beautifully', 'great', and 'superb', it ultimately makes a negative decision based on multiple words that seem to have no obvious negative sentiment (e.g. 'Asano', 'Asian', 'movie', 'acting').

This demonstrates significant flaws in the logic that the predictor utilizes to classify the vocabulary in the text of the given reviews. The next section demonstrates how improving this logic can significantly boost the predictor's performance.

Improving Text Vectorization

The first version of the prediction pipeline used the CountVectorizer tool to simply count the number of times that each word appears in each review. This approach ignores two important facts about human language:

- The meaning and importance of a word can change based on the words that surround it.
- The frequency of a word within a document is not always an accurate representation of its importance. For instance, even though two occurrences of the word 'great' may be a strong positive indicator in a document with 100 words, it is far less important in a larger document with 1000 words.

Regular expression

A regular expression is a pattern of text used for matching and manipulating strings and provides a concise and flexible way to specify text patterns and is widely used in text processing and data analysis.

This section will demonstrate how text vectorization can be improved to take these two facts into account. The following code imports three different Python libraries that will be used to achieve this:

- **nltk** and **gensim**: two popular libraries used for various Natural Language Processing (NLP) tasks.
- **re**: a library used to search and process text using regular expressions.

```
%%capture

!pip install nltk # install nltk
!pip install gensim # install gensim

import nltk # import nltk
nltk.download('punkt') # install nltk's tokenization tool, used to split a text into sentences.

import re  # import re

from gensim.models.phrases import Phrases, ENGLISH_CONNECTOR_WORDS # import tools
from the gensim library.
```

Detecting Phrases

The following function can be used to split a given document into a list of tokenized sentences, where each tokenized sentence is represented as a list of words:

Tokenization

The process of breaking up textual data into pieces such as words, sentences, symbols and other elements called tokens.

```
# convert a given doc to a list of tokenized sentences.
def tokenize_doc(doc:str):
    return [re.findall(r'\b\w+\b',
            sent.lower()) for sent in nltk.sent_tokenize(doc)]
```

The sent_tokenize() function splits the doc into a list of sentences.

The sent_tokenize() function from the nltk library splits the document into a list of sentences. Each sentence is then lowercased and fed to the findall() function of the **re** library, which locates occurrences of the '\b\w+\b' regular expression. You will test it on the string provided on the **raw_text** variable. In this expression:

- \w matches all alphanumeric characters (a-z, A-Z, 0-9) and the underscore character.
- \w+ is used to capture "one or more" \w characters. So, in the string "hello123_world", the pattern \w+ would match the words "hello", "123", and "world".
- \b represents the boundary between a \w character and a non-\w character, as well as at the start or end of the given string. For example, the pattern \bcat\b would match the word "cat" in the string "The cat is cute", but it would not match the word "cat" in the string "The category is pets".

Let's see an example of tokenization using the tokenize_doc( ) function.

```
raw_text='The movie was too long. I fell asleep after the first 2 hours.'

tokenized_sentences=tokenize_doc(raw_text)

tokenized_sentences
```

```
[['the', 'movie', 'was', 'too', 'long'],
 ['i', 'fell', 'asleep', 'after', 'the', 'first', '2', 'hours']]
```

The tokenize_doc() function can now be combined with the Phrases tool from the gensim library to create a phrase model, a model that can identify multi-word phrases in a given sentence. The following code utilizes the IMDB training data (X_train_text) to build such a model:

```
sentences=[] # list of all the tokenized sentences across all the docs in this dataset

for doc in X_train_text: # for each doc in this dataset
    sentences+=tokenize_doc(doc) # get the list of tokenized sentences in this doc

# build  a phrase model on the given data
imdb_phrase_model = Phrases(sentences, ❶
                            connector_words=ENGLISH_CONNECTOR_WORDS, ❷
                            scoring='npmi', ❸
                            threshold=0.25).freeze() ❹
```

As shown above, the Phrases() function accepts four parameters:

❶ The list of tokenized sentences from the given document collection.

❷ A list of common english words that appear frequently in phrases (e.g. 'of', 'the'), that do not have any positive or negative value, but can add sentiment depending on the context, so they are treated differently.

❸ A scoring function is used to determine if a sequence of words should be included in the same phrase. The code above uses the popular Normalized Pointwise Mutual Information (NPMI) measure for this purpose. NPMI is based on the co-occurrence frequency of the words in a candidate phrase and takes a value between -1 (complete independence) and +1 (complete co-occurrence).

❹ A threshold for the scoring function. Phrases with a lower score are ignored. In practice, this threshold can be tuned to identify the value that yields the best results for a downstream application (e.g. predictive modeling).

The freeze() suffix converts the phrase model into an unchangeable ("frozen") but much faster format.

When applied to the two tokenized sentence examples shown above, this phrase model produces the following results:

```
imdb_phrase_model[tokenized_sentences[0]]
```

```
['the', 'movie', 'was', 'too_long']
```

```
imdb_phrase_model[tokenized_sentences[1]]
```

```
['i', 'fell_asleep', 'after', 'the', 'first', '2_hours']
```

The phrase model identifies three phrases: 'too_long', 'fell_asleep', and '2_hours'. All three carry more information than their individual words.

For example, 'too_long' clearly carries a negative sentiment, even though the words 'too' or 'long' by themselves do not. Similarly, even though seeing the word 'asleep' in a movie review is likely negative evidence, the phrase 'fell_asleep' delivers a much clearer message. Finally, '2_hours' captures a much more specific context than the words '2' and 'hours'.

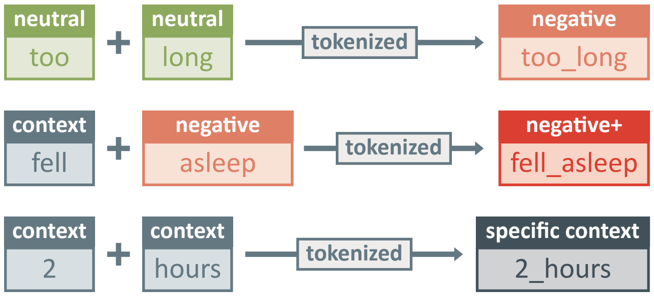


Figure 3.12: Positive and negative sentiments before and after tokenization

The following function uses this phrase-detection capability to annotate phrases in a given document:

```
def annotate_phrases(doc:str, phrase_model):

    sentences=tokenize_doc(doc)# split the document into tokenized sentences.

    tokens=[] # list of all the words and phrases found in the doc
    for sentence in sentences: # for each sentence
        # use the phrase model to get tokens and append them to the list.
        tokens+=phrase_model[sentence]
    return ' '.join(tokens) # join all the tokens together to create a new annotated document.
```

The following code uses the annotate_phrases() function to annotate both the training and testing reviews from IMDb dataset:

```
#annotate all the test and train reviews.
X_train_text_annotated=[annotate_phrases(doc,imdb_phrase_model) for doc in X_
train_text]
X_test_text_annotated=[annotate_phrases(text,imdb_phrase_model)for text in X_
test_text]
```

```
# an example of an annotated document from the imdb training data
X_train_text_annotated[0]
```

```
'i_grew up b 1965 watching and loving the thunderbirds all my_mates at
school watched we played thunderbirds before school during lunch and
after school we all wanted to be virgil or scott no_one wanted to be alan
counting down from 5 became an art_form i took my children to see the movie
hoping they would get_a_glimpse of what i_loved as a child how bitterly
disappointing the only high_point was the snappy theme_tune not that it
could compare with the original score of the thunderbirds thankfully early
saturday_mornings one television_channel still plays reruns of the series
gerry_anderson and his_wife created jonatha frakes should hand in his
directors chair his version was completely hopeless a waste of film utter_
rubbish a cgi remake may_be acceptable but replacing marionettes with homo_
sapiens subsp sapiens was a huge error of judgment'
```

Using TF-IDF for Text Vectorization

The frequency of a word within a document is not always an accurate representation of its importance. A better way to represent frequency is the popular TF-IDF measure. TF-IDF, which stands for "Term Frequency Inverse Document Frequency", uses a simple mathematical formula to determine the importance of tokens (i.e. words or phrases) in a document based on two factors:

- the frequency of the token in the document, as measured by the number of times the token appears in the document divided by the total number of tokens in the documents
- the token's inverse document frequency, computed by dividing the total number of documents in the dataset by the number of documents that contain the token.

The first factor avoids the overestimation of the importance of terms that appear in longer documents. The second factor penalizes terms that appear in too many documents, which helps to adjust for the fact that some words are more common than others.

Term Frequency Inverse Document Frequency (TF-IDF)

TF-IDF is a statistical method which is used to determine the importance of tokens in a document.

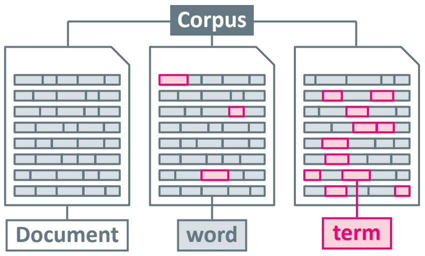


Figure 3.13: Words and terms in document

$$\frac{\text{number of documents in data set}}{\text{number of documents containing term}} = \text{IDF}$$

$$\frac{\text{times of term appears in document}}{\text{number of words in the document}} = \text{TF}$$

TF * IDF = Value

TfidfVectorizer Tool

The sklearn library provides a tool that supports this type of TF-IDF vectorization. The TfidfVectorizer tool can be used to vectorize a phrase.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Train a TF-IDF model with the IMDb training dataset
vectorizer_tf = TfidfVectorizer(min_df=10)
vectorizer_tf.fit(X_train_text_annotated)
X_train_tf = vectorizer_tf.transform(X_train_text_annotated)
```

This new vectorizer can now be input to the same Naive Bayes Classifier to build a new predictive pipeline and apply it to the IMDb testing data:

```
# train a new Naive Bayes Classifier on the newly vectorized data.
model_tf =MultinomialNB()
model_tf.fit(X_train_v2, Y_train)

# create a new prediction pipeline.
prediction_pipeline_tf = make_pipeline(vectorizer_tf, model_tf)

# get predictions using the new pipeline.
predictions_tf = prediction_pipeline_tf.predict(X_test_text_annotated)

# print the achieved accuracy.
accuracy_score(Y_test, predictions_tf)
```

```
0.8858
```

This new pipeline achieves an accuracy of 88.58%, a significant improvement over the 84.68% reported by the previous one. This improved pipeline can now be used to revisit the test example that was misclassified by the first pipeline:

```
# get the review example that confused the previous algorithm
mistake_example_annotated=X_test_text_annotated[4600]

print('\nReview:',mistake_example_annotated)

# get the correct labels of this example.
print('\nCorrect Label:', class_names[Y_test[4600]])

# get the prediction probabilities for this example.
print('\nPrediction Probabilities for neg, pos:',prediction_pipeline_
tf.predict_proba([mistake_example_annotated]))
```

```
Review: i_personally thought the movie was_pretty good very_good acting by
tadanobu_asano of ichi_the_killer fame i really can_t say much about the
story but there_were parts that confused me a little_too much and overall
i_thought the movie was just too lengthy other_than that however the movie
contained superb_acting great fighting and a lot of the locations were
beautifully_shot great effects and a lot of sword play another solid effort
by tadanobu_asano in my_opinion well i really can_t say anymore about the
movie but if_you re only outlook on asian_cinema is crouching_tiger hidden_
dragon or house of flying_daggers i_would suggest_you trying to rent_it but
if_you re a die_hard asian_cinema fan i_would say this has to be in your_
collection very_good japanese film

Correct Label: pos

Prediction Probabilities for neg, pos: [[0.32116538 0.67883462]]
```

The new pipeline confidently predicts the correct positive label for this review. The following code uses the LIME explainer to explain the logic behind this prediction:

```
# create an explainer.
explainer_tf = LimeTextExplainer(class_names=class_names)

# explain the prediction  of the second pipeline for this example.
exp = explainer_tf.explain_instance(mistake_example_annotated, prediction_
pipeline_tf.predict_proba, num_features=10)

# visualize the results.
fig = exp.as_pyplot_figure()
```

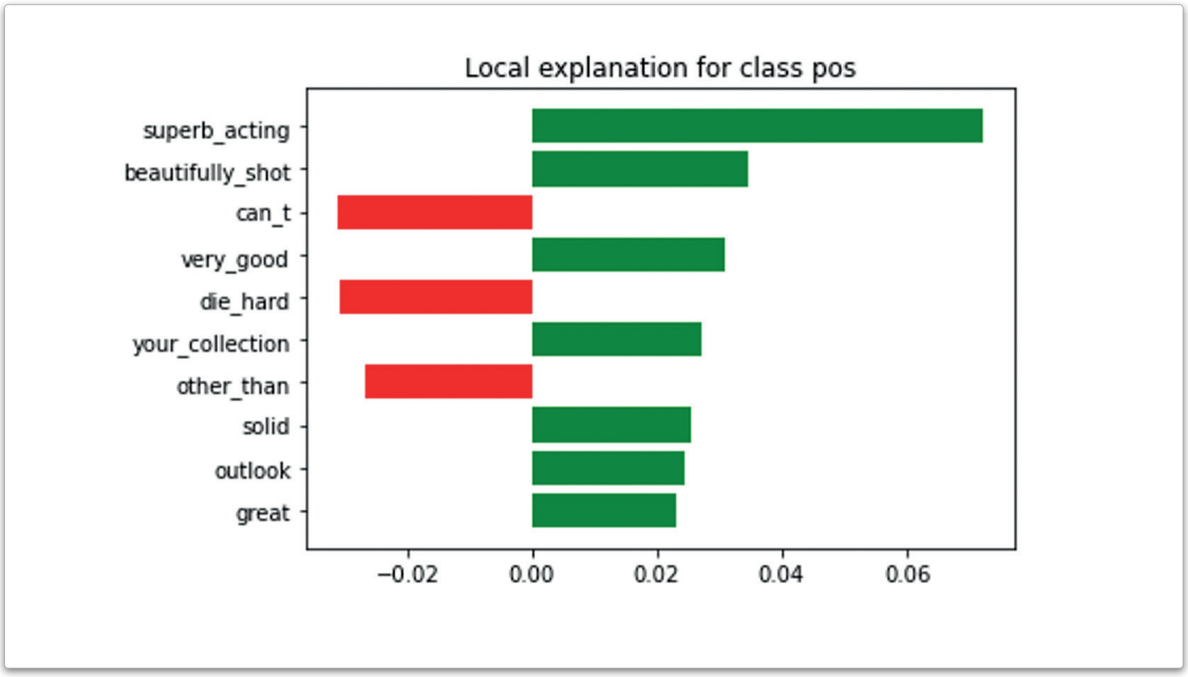


Figure 3.14: Word influence for TF-IDF and Naive Bayes Classifier combination

The results verify that the new pipeline follows a significantly more intelligent logic. It correctly identifies the positive sentiment of phrases like 'superb_acting', 'beautifully_shot' and 'very good'. It is also not misguided by the words that erroneously drove the first pipeline toward a negative prediction.

The performance of the predictive pipeline can be further improved in multiple ways, such as replacing the Naive Bayes classifier with more sophisticated methods and tuning the parameters of these methods to maximize their potential. Another option would be to experiment with alternative vectorization techniques that are not based on token frequency, such as the word and document embeddings that will be explored in the following lesson.

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. In supervised learning, you use labeled datasets to train the model.	○	○
2. Vectorization is a technique of converting data from numeric vector format to raw data.	○	○
3. The sparse format requires far less memory than the dense matrix.	○	○
4. The Naive Bayes Classifier algorithm is used to build a prediction pipeline.	○	○
5. The frequency of a word within a document is the only accurate representation of its importance.	○	○

2 Explain the reason the dense matrix format requires more space in the memory than the sparse format.

3 Analyze how the two mathematical factors in TD-IDF are utilized to inspect the importance of a word in a document.

4 You are given a numPy array X_train_text that includes one document in each row. You are also given a second array Y_train that includes the labels for the documents in X_train_text. Complete the following code so that it uses TF-IDF to vectorize the data, trains a MultinomialNB classification model on the vectorized version, and then combines the vectorizer and classification model into a single prediction pipeline.

```
from _______________.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import ________________

vectorizer = ______________(min_df=10)

vectorizer.fit(__________________) # fits the vectorizer on the training data

X_train = vectorizer.______________(X_train_text) # uses the fitted vectorizer to vectorize the data
model_MNB=MultinomialNB() # a Naive Bayes Classifier

model_MNB.fit(X_train, _________________) # fits the classifier on the vectorized training data

prediction_pipeline = make_pipeline(_________________, _________________)
```

5 Complete the following code so that it builds LimeTextExplainer for the prediction pipeline that you built in the previous exercise and uses the explainer to explain the prediction for a specific text example.

```
from _______________ import LimeTextExplainer
text_example="I really enjoyed this movie, the actors were excellent"
class_names=['neg','pos'] # creates a local explainer for explaining individual predictions

explainer = _______________(class_names=class_names) # explains the prediction for this example

exp = explainer.___________(text_example.lower(),prediction_pipeline.___________,
                  _________________=10) # focuses the explainer on the 10 most influential features

print(exp.________________) # prints the words with the highest influence on the prediction
```

Lesson 2

Unsupervised Learning

Link to digital lesson

www.ien.edu.sa

Unsupervised Learning to Understand Text

Unsupervised learning is a type of machine learning where the model is not given any labeled training data. Instead, the model is only given a set of examples and must find patterns and relationships within the data on its own. In the context of understanding text, unsupervised learning can be used to discover latent structures and patterns within a dataset of text documents. There are many different techniques that can be used for unsupervised learning of text data, including clustering algorithms, dimensionality reduction techniques, and generative models.

Clustering algorithms can be used to group together similar documents, while dimensionality reduction techniques can be used to reduce the dimensionality of the data and identify important features. Generative models, on the other hand, can be used to learn the underlying distribution of the data and generate new text that is similar to the original dataset.

Clustering Algorithms

Clustering algorithms can group similar customers based on their behavior, demographics, or purchasing history for targeted marketing and increased customer retention.

Dimensionality Reduction Techniques

Dimensionality reduction is used in image compression to reduce the number of pixels in an image to minimize the amount of data needed to represent the image while preserving its main features.

Generative Models

Generative models are used in anomaly detection applications where anomalies are detected in data by learning the normal patterns of the data using a generative model.

Unsupervised Learning

In unsupervised learning, you give to the model large amounts of data that are not labeled and it has to find patterns in the unstructured data through observation and clustering.

Dimensionality Reduction

Dimensionality reduction is a technique in machine learning and data analysis to reduce the number of features (dimensions) in a dataset while retaining as much information as possible.

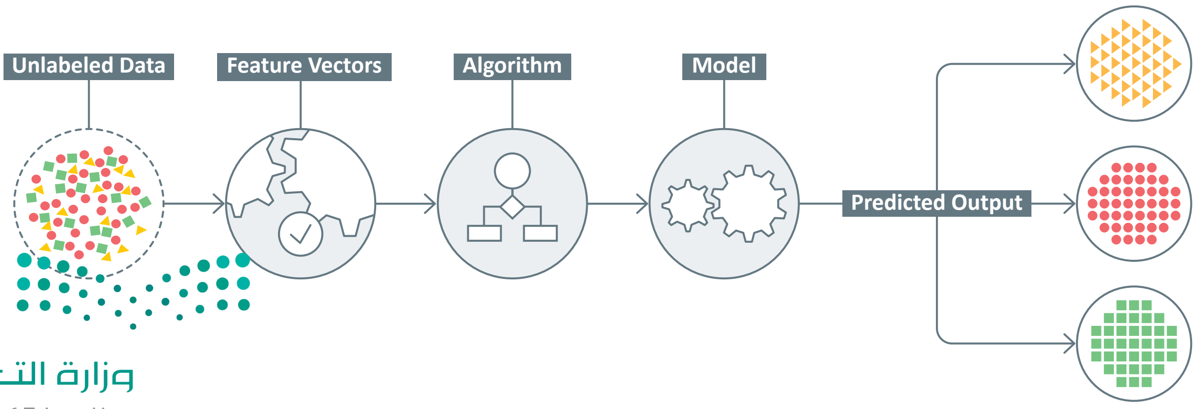


Figure 3.15: Unsupervised learning representation

One of the key advantages of using unsupervised learning is that it can be used to identify patterns and relationships that may not be immediately apparent to a human observer. This can be especially useful for understanding large datasets of unstructured text, where manual analysis may be impractical.

In this unit, you will use an openly available dataset of news articles from the BBC to demonstrate some key techniques for unsupervised learning (Greene & Cunningham, 2006). The following code is used to load the dataset, which is organized into five different news folders representing articles from different news sections: business, politics, sports, technology, and entertainment. These five labels will not be used to inform any of the algorithms presented in this unit. Instead, they will only be used for visualization and validation purposes.

Each news folder includes hundreds of text files, with each file including the content of a single specific article. The dataset is already loaded into the Jupyter Notebook, and the codeblock will open the dataset and extract all the documents and required labels to two list data structures, respectively.

Cluster

A cluster is a group of similar things. In machine learning, grouping unlabeled data in homogeneous clusters is called clustering.

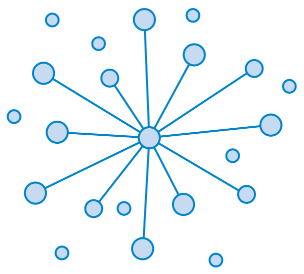


Figure 3.16: Representation of a cluster

BBC open dataset

https://www.kaggle.com/datasets/shivamkushwaha/bbc-full-text-document-classification

D. Greene and P. Cunningham. "Practical Solutions to the Problem of Diagonal Dominance in Kernel Document Clustering",Proc. ICML 2006. All rights, including copyright, in the content of the original articles are owned by the BBC.

```
# used to list all the files and subfolders in a given folder
from os import listdir
# used for generating random number
import random shuffling lists

bbc_docs=[] # holds the text of the articles
bbc_labels=[] # holds the news section for each article

for folder in listdir('bbc'): # for each news-section folder
    for file in listdir('bbc/'+folder): # for each text file in this folder

        # open the text file, use encoding='utf8' because articles may include non-ascii characters
        with open('bbc/'+folder+'/'+file,encoding='utf8',errors='ignore') as f:
            bbc_docs.append(f.read()) # read the text of the article and append to the docs list
        # use the name of the folder (news section) as a label for this doc
        bbc_labels.append(folder)
# shuffle the docs and labels lists in parallel
merged = list(zip(bbc_docs, bbc_labels)) # link the two lists
random.shuffle(merged) # shuffle them in parallel (with the same random order)
bbc_docs, bbc_labels = zip(*merged) # separate them again into individual lists.
```

Document Clustering

Now that the dataset has been loaded, the next step is to experiment with various unsupervised methods. Clustering is arguably the most popular type of method in this domain. Given a collection of unlabeled documents, the goal of clustering is to group documents that are similar to one another, while separating documents that are dissimilar.

Document Clustering

Document clustering is a method which groups textual documents into clusters based on their content similarity.

Table 3.2: Factors that determine the quality of the results

1	The way in which the data has been vectorized. Even though TF-IDF is an established technique in this space, this unit will also explore more sophisticated alternatives.
2	The exact definition of document-to-document similarity. For vectorized text data, the Euclidean and Cosine distance measures are the most popular. The former will be used in the examples presented in this unit.
3	The selected number of clusters. Agglomerative Clustering (AC) provides an intuitive method for selecting the appropriate number of clusters for a given dataset, which is a key challenge for clustering tasks.

Selecting the Number of Clusters

Selecting the correct number of clusters is a crucial step for any clustering task. Unfortunately, the vast majority of clustering algorithms expect the practitioner to provide the correct number of clusters as part of the input. The selected number can have a significant impact on the quality and interpretability of the results.

There are several approaches that can be used to select the number of clusters.

- One common approach is to use a measure of cluster "compactness". This can be done by calculating the sum of the distances between the points within each cluster, and selecting the number of clusters that minimizes this sum.
- Another approach is to use a measure of the "separation" between the clusters, such as the average distance between points in different clusters, accordingly, the number of clusters raised from this average is determined.

In practice, the above approaches often contradict each other by recommending different numbers. This is an especially common challenge when working with text data, whose structure is often difficult to discern.

Euclidean Distance

Euclidean distance is a straight-line distance between two points in a multidimensional space. It is calculated as the square root of the sum of the squares of the differences between the corresponding dimensions of the points. Euclidean distance is used in clustering to measure the similarity between two data points.

Cosine Distance

Cosine distance measures the cosine similarity between two data points. It calculates the cosine of the angle between two vectors representing the data points and is often used in text data clustering. The cosine similarity value is between -1 and 1, with -1 indicating the complete opposite and 1 indicating the same direction.

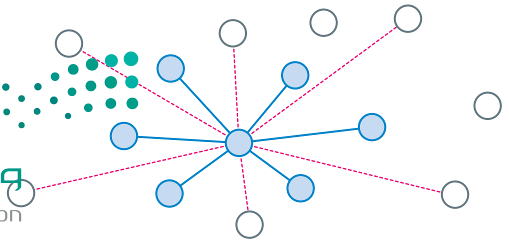


Figure 3.17: Machine calculating the distances between points

The number of clusters in unsupervised learning determines how many groups or categories the algorithm will divide the data into. Choosing the right number of clusters is important because it affects the accuracy and interpretability of the results. If clusters are too high, the groups may be too specific and not meaningful. If the number of clusters is too low, the groups may be too broad and not capture the underlying structure of the data. It is important to strike a balance between having enough clusters to capture meaningful patterns but not so many that the results become too complex to understand.

Hierarchical Clustering

Hierarchical clustering is a clustering algorithm for grouping data into clusters based on similarity. In hierarchical clustering, the data points are organized into a tree-like structure, where each node represents a cluster, and the parent node represents a merger of its child nodes.

The following code imports specific libraries that will be used for the end-to-end hierarchical clustering:

```python
# used for tfi-df vectorization, as seen in the previous unit
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import AgglomerativeClustering # used for agglomerative clustering

# used to visualize and support hierarchical clustering tasks
import scipy.cluster.hierarchy as hierarchy

# set the color palette to be used by the 'hierarchy' tool.
hierarchy.set_link_color_palette
(['blue','green','red','yellow','brown','purple','orange','pink','black'])

import matplotlib.pyplot as plt # used for general visualizations
```

Text Vectorization

Similar to the supervised methods that were presented in the previous unit, many methods for unsupervised learning also require raw text to be vectorized into a numeric format.

The following code uses the TfidfVectorizer tool (which was also used in the previous lesson) for this purpose:

```python
vectorizer = TfidfVectorizer(min_df=10) # apply tf-idf vectorization, ignore words that
appear in more than 10 docs.

text_tfidf=vectorizer.fit_transform(bbc_docs) # fit and transform in one line

text_tfidf
```

```
<2225x5867 sparse matrix of type '<class 'numpy.float64'>'
        with 392379 stored elements in Compressed Sparse Row format>
```

As can be seen above, the document data have now been converted into the sparse numeric format that was also used in the previous lesson.

The following code uses the TSENVisualizer tool from the yellowbrick library to project and visualize the vectorized documents within a 2-dimensional space:

```
%%capture
!pip install yellowbrick
from yellowbrick.text import TSNEVisualizer
```

Dimensionality Reduction

Dimensionality reduction can be useful in a number of applications, such as:

- **Visualizing high-dimensional data:** It can be difficult to visualize data in a high-dimensional space, so reducing the number of dimensions can make it easier to visualize and understand the data.
- **Reducing the complexity of a model:** A model with fewer dimensions may be simpler and easier to understand, and the training process is faster.
- **Improving the performance of a model:** Dimensional reduction can help remove noise and redundancy from the data, which can improve the performance of a model.

t-Distributed Stochastic Neighbor Embedding (T-SNE)

T-SNE (t-Distributed Stochastic Neighbor Embedding) is an unsupervised machine learning algorithm for dimensionality reduction.

Table 3.3: Dimensionality reduction techniques

Technique	Description	Example of use
Feature selection	Feature selection involves selecting a subset of the original features.	Medical datasets may have hundreds of columns per patient case. Only a few of these features can help the model diagnose correctly. Other traits are unrelated to the diagnosis and may distract the model. Feature selection discards all but the most discriminating features.
Feature transformation	Feature transformation involves combining or transforming the original features to create a new set of features. The original used features can be dropped as they have become redundant.	Consider predicting a patient's stay on admission, we can create additional features for the model from the current features of the patient's medical records. For example, compute the number of lab tests ordered during the past week, or the number of visits during the past month. Another example is computing the area of a rectangle from it's height and width.
Manifold learning	Manifold learning techniques, such as t-SNE and UMAP (Uniform Manifold Approximation and Projection), are unsupervised learning techniques that aim to preserve the structure of the data in a lower-dimensional space.	They can convert a high-dimensional image into a lower-dimensional space while keeping its primary characteristics and structure. Since it takes up less space, this compressed representation may be stored and sent, and the original image can be rebuilt with minimal loss of information

One of the key features of t-SNE is that it tries to preserve the local structure of the data as much as possible, so that similar data points are nearby in the low-dimensional representation. It does this by minimizing the divergence between two probability distributions: the distribution of the high-dimensional data and the distribution of the low-dimensional data.

The vectorized BBC dataset is indeed high-dimensional, as it includes a separate dimension (column) for each of the unique words that appear in the data.

The total number of dimensions can be computed as follows:

```
print('Number of unique words in the BBC documents vectors:',
                   len(vectorizer.get_feature_names_out()))
```

```
Number of unique words in the BBC documents vectors: 5867
```

The following code can now be used to project these 5,867 dimensions into just two (the X and Y coordinates of the plot). This code will create a scatter plot diagram, where each color represents one of five news sections.

```
tsne = TSNEVisualizer(colors=['blue','green','red','yellow','brown'])
tsne.fit(text_tfidf,bbc_labels)
tsne.show();
```

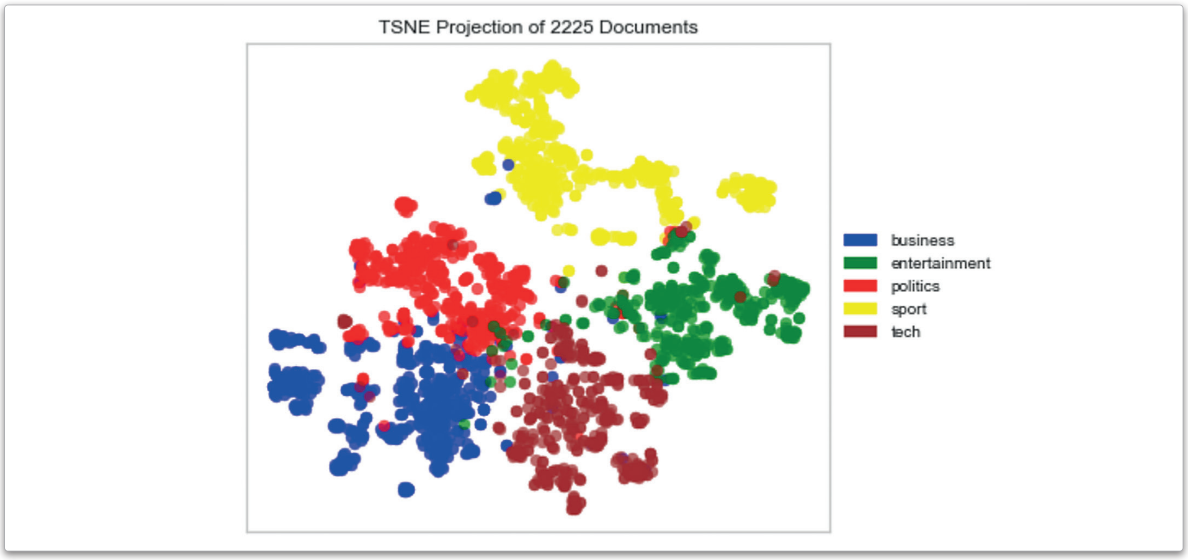


Figure 3.18: TSNE projection

This visualization uses the original "ground-truth" label (news section) of each document to reveal the dispersion of each label across the 2D projected vectorization space. The figure reveals that, even though there are some impurities in certain pockets of the space, the five news sections are generally well-separated. Later, an improved vectorization that reduces these impurities will be described.

Agglomerative Clustering (AC)

Agglomerative Clustering (AC), also called hierarchical clustering, is one of the most popular and effective methods in this space, it addresses this challenge by providing an intuitive visual method for selecting the appropriate number of clusters. AC follows a bottom-up approach. It begins by computing the distance between all pairs of data points. It then selects the two closest points and merges them into a single cluster. This process is repeated until all of the data points have been merged into a single cluster or until the desired number of clusters has been reached.

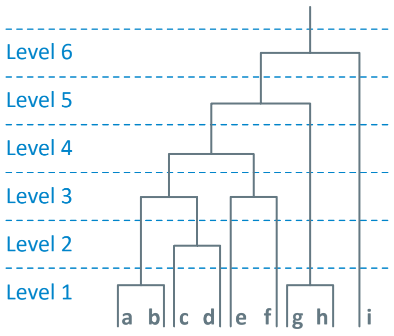


Figure 3.19: Agglomerative Clustering (AC)

fx Linkage() Function

Python implements Agglomerative Clustering with the linkage() function.

Two parameters are provided for the linkage() function:

- The vectorized text data. The toarray() function is used to convert the data to its dense format, as required by this function.
- The distance metric that should be used to decide which clusters to merge next during the agglomerative process. There are many different options to choose from for a distance metric depending on the needs and preferences of the user, like Euclidian, Manhattan, etc. For this project you will use the **ward** distance metric.

The following code uses the linkage() function from the 'hierarchy' tool (imported above) to apply this process to the vectorized BBC data:

```
plt.figure() # create a new empty figure

# iteratively merge points and clusters until all points belong to a single cluster
# return the linkage of the produced tree
linkage_tfidf=hierarchy.linkage(text_tfidf.toarray(),method='ward')

# visualize the linkage
hierarchy.dendrogram(linkage_tfidf)

# show the figure
plt.show()
```

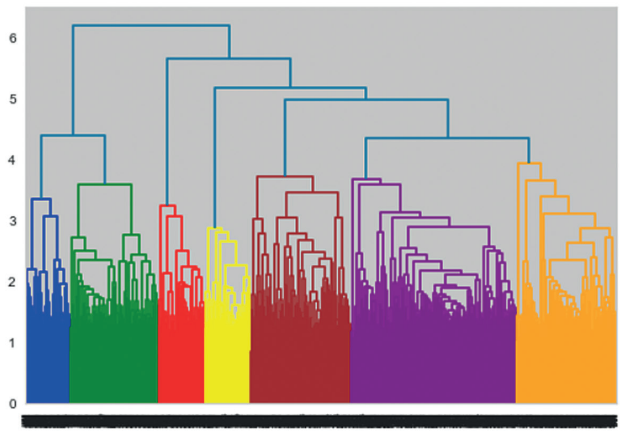


Figure 3.20: Hierarchy dendrogram for the BBC data

Ward Distance

The example above uses the popular Ward distance metric for the second parameter. The Ward distance is based on the concept of within-cluster variance, defined as the sum of the distances between the points in a cluster. In each iteration, the method evaluates every possible merge by computing the within-cluster variance before and after the merge. It then performs the merge that leads to the lowest variance increase. Even though Ward is one of the multiple options, it has been shown to work well for text data.

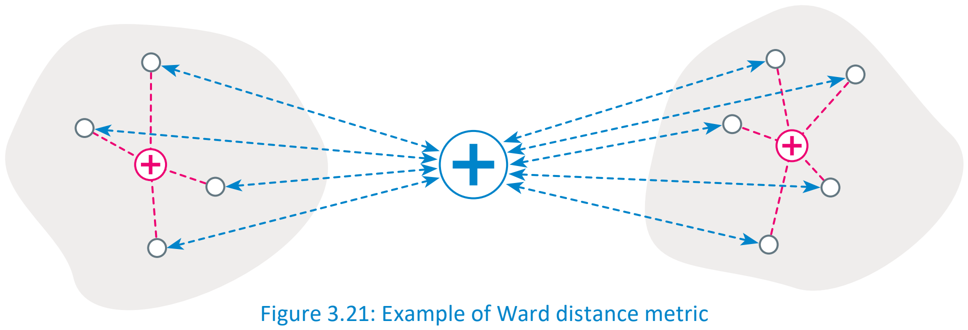


Figure 3.21: Example of Ward distance metric

The dendrogram in figure 3.20 provides an intuitive way of selecting the number of clusters. In this example, the library suggests using 7 clusters, each highlighted with a different color.

The practitioner can either adopt this suggestion or use the dendrogram to pick a different number. For instance, the blue and green pair was merged last with the cluster group of all the other colors. Therefore, choosing 6 clusters would merge purple and orange, while choosing 5 clusters would also merge blue and green.

Dendrogram

The dendrogram is a tree diagram which shows the hierarchical relationship between data. Usually, it is created as an output from hierarchical clustering.

The following code adopts the tool's suggestions and uses the AgglomerativeClustering tool from the sklearn library to cut the tree after the 7 clusters have been created:

```
AC_tfidf=AgglomerativeClustering(linkage='ward',n_clusters=7) # prepare the tool,
set the number of clusters.

AC_tfidf.fit(text_tfidf.toarray()) # apply the tool to the vectorized BBC data.

pred_tfidf=AC_tfidf.labels_ # get the cluster labels.

pred_tfidf
```

```
array([6, 2, 4, ..., 6, 3, 5], dtype=int64)
```

Note that the original "ground-truth" label (news section) of each document has not been used at all during this process. Instead, clustering was done exclusively based on the text of each document. Having such ground-truth labels can be useful in practice, as it allows for the validation of the clustering results. The current "ground-truth" labels are the ones on the **bbc_labels** list.

The following code uses the ground-truth labels and three different scoring functions from the sklearn library to evaluate the quality of the produced clustering:

- The **Homogeneity score** takes values between 0 and 1 and is maximized when all the points of each cluster have the same ground-truth label. Equivalently, each cluster contains only data points of a single class.
- The **Adjusted Rand score** takes values between -0.5 and 1.0 and is maximized when all the data points with the same label are in the same cluster and all points with different labels are in different clusters.
- The **Completeness score** also takes values between 0 and 1 and is maximized when all data points of a given class are assigned to the same cluster.

```
from sklearn.metrics import homogeneity_score,adjusted_rand_
score,completeness_score

print('\nHomogeneity score:',homogeneity_score(bbc_labels,pred_tfidf))
print('\nAdjusted Rand score:',adjusted_rand_score(bbc_labels,pred_tfidf))
print('\nCompleteness score:',completeness_score(bbc_labels,pred_tfidf))
```

```
Homogeneity score: 0.6224333236569846

Adjusted Rand score: 0.4630492696176891

Completeness score: 0.5430590192420555
```

Closer to 1 means that the group of texts in the cluster belongs to 1 label.

Closer to 1 means better 1-1 mapping of clusters to labels.

To complete the analysis, the data is re-clustered using 5 clusters, which are equal to the actual number of ground truth labels:

```
AC_tfidf=AgglomerativeClustering(linkage='ward',n_clusters=5)
AC_tfidf.fit(text_tfidf.toarray())
pred_tfidf=AC_tfidf.labels_

print('\nHomogeneity score:',homogeneity_score(bbc_labels,pred_tfidf))
print('\nAdjusted Rand score:',adjusted_rand_score(bbc_labels,pred_tfidf))
print('\nCompleteness score:',completeness_score(bbc_labels,pred_tfidf))
```

```
Homogeneity score: 0.528836079209762

Adjusted Rand score: 0.45628412883628383

Completeness score: 0.6075627851312266
```

Providing the AC clustering the actual number of labels, gives a better Completeness score, meaning the clustering is more representative.

Even though the score results reveal that the combination of Agglomerative Clustering with TF-IDF vectorization produces reasonable results, the quality of the clustering can be improved. The next section demonstrates how vectorization techniques based on neural networks can lead to superior results.

Word Vectorization with Neural Networks

TF-IDF vectorization is based on counting and normalizing the frequency of words across the documents in the dataset. Even though this can lead to good results, frequency-techniques have a significant limitation, as they completely ignore the semantic connection between words. For example, even though the words 'trip' and 'journey' are synonyms, frequency-based vectorization would treat them as completely separate and independent features. Similarly, even though the words 'apple' and 'fruit' are semantically related (as apples are a type of fruit), this relation will also be ignored.

This limitation can significantly impact downstream applications that use this type of vectorization. Consider these two sentences:

- "I have a very high fever, so I have to visit a doctor."
- "My body temperature has risen significantly, so I need to see a healthcare professional."

Even though these two sentences describe the exact same scenario, they do not share any informative words. Therefore, any clustering algorithm that is based on TF-IDF (or any other frequency-based) vectorization would fail to see their similarity and would likely not place them in the same cluster.

Word2Vec

This limitation can be addressed via methods that consider the semantic similarity between words. One of the most popular methods for this purpose is Word2Vec, which uses an architecture based on neural networks.

Word2Vec is based on the intuition that semantically similar words will typically be surrounded by the same context words. Therefore, given that the neural network uses the hidden embedding of each word to predict its context, similar words should be mapped to similar embeddings.

In practice, Word2Vec models are pre-trained on millions of documents to learn high-quality word embeddings. Such pre-trained models can then be downloaded and used in any text-based application.

The following code uses the gensim library to download a popular pre-trained model that has been trained on a very large dataset from Google News:

```
import gensim.downloader as api
model_wv = api.load('word2vec-google-news-300')
fox_emb=model_wv['fox']
print(len(fox_emb))
```

```
300
```

This model maps each word to an embedding with 300 dimensions.

Stopwords

Stopwords are common words in a language often removed during the text pre-processing step in NLP tasks such as word vectorization. These words include articles, conjunctions, and prepositions and are not typically considered useful for determining the meaning or context of a text.

Embedding

Embedding represents words or tokens in a continuous vector space where semantically similar words are mapped to nearby points.

The first 10 dimensions of the numeric "fox" embedding are displayed below:

```
fox_emb[:10]
```

```
array([-0.08203125, -0.01379395, -0.3125    , -0.04125977,  0.05493164,
       -0.12988281, -0.10107422, -0.00164795,  0.15917969,  0.12402344],
      dtype=float32)
```

The model can use the embeddings of the words to evaluate their similarity. Consider the following example, which compares the word 'car' with other words of decreasing similarity. Similarity values are always between 0 and 1.

```
pairs = [
    ('car', 'minivan'),
    ('car', 'bicycle'),
    ('car', 'airplane'),
    ('car', 'street'),
    ('car', 'apple'),
]
for w1, w2 in pairs:
    print(w1, w2, model_wv.similarity(w1, w2))
```

```
car minivan 0.69070363
car bicycle 0.5364484
car airplane 0.42435578
car street 0.33141237
car apple 0.12830706
```

The following code can be used to find the 5 most similar words of a given word:

```
print(model_wv.most_similar(positive=['apple'], topn=5))
```

```
[('apples', 0.720359742641449), ('pear', 0.6450697183609009),
('fruit', 0.6410146355628967), ('berry', 0.6302295327186584), ('pears',
0.613396167755127)]
```

Visualization can be used to further validate the embeddings of this pre-trained model. This can be achieved by:

- Selecting a sample of words from the BCC dataset.
- Using t-SNE to reduce the 300-dimensional embedding of each word to a 2-dimensional point.
- Visualizing the points as a scatter plot in 2-dimensional space.

```
%%capture
import nltk # import the nltk library for nlp.
import re # import the re library for regular expressions.
import numpy as np # used for numeric computations
from collections import Counter # used to count the frequency of elements in a given list
from sklearn.manifold import TSNE # Tool used for Dimensionality Reduction.

# download the 'stopwords' tool from the nltk library. It includes very common words for different
languages
nltk.download('stopwords')

from nltk.corpus import stopwords # import the 'stopwords' tool.

stop=set(stopwords.words('english')) # load the set of english stopwords.
```

The following function is then used to select a sample of representative words from the BBC dataset. Specifically, the code selects the top 50 most frequent words from each of the 5 BBC news sections, excluding stopwords (very common English words) and words that are not included in the pre-trained Word2Vec model.

Some very common and frequent English words considered stopwords include "a", "the", "is" and "are".

```
def get_sample(bbc_docs:list,
               bbc_labels:list
               ):

    word_sample=set() # a sample of words from the BBC dataset

    # for each BBC news section
    for label in ['business', 'entertainment', 'politics', 'sport', 'tech']:

        # get all the words in this news section, ignore stopwords.
        # for each BBC doc and for each word in the BBC doc
        # if the word belongs to the label and is not a stopword and is included in the Word2Vec model
        label_words=[word for i in range(len(bbc_docs))
                     for word in re.findall(r'\b\w\w+\b',bbc_docs[i].lower())
                         if bbc_labels[i]==label and
                            word not in stop and
                            word in model_wv]

        cnt=Counter(label_words) # count the frequency of each word in this news section.

        # get the top 50 most frequent words in this section.
        top50=[word for word,freq in cnt.most_common(50)]
        # add the top50 words to the word sample.
        word_sample.update(top50)


    word_sample=list(word_sample) # convert the set to a list.
    return word_sample

word_sample=get_sample(bbc_docs,bbc_labels)
```

Finally, you can use a method with t-SNE to reduce the 300-dimensional embeddings of the words in the sample into 2-dimensional points. The points are then visualized via a simple scatter plot.

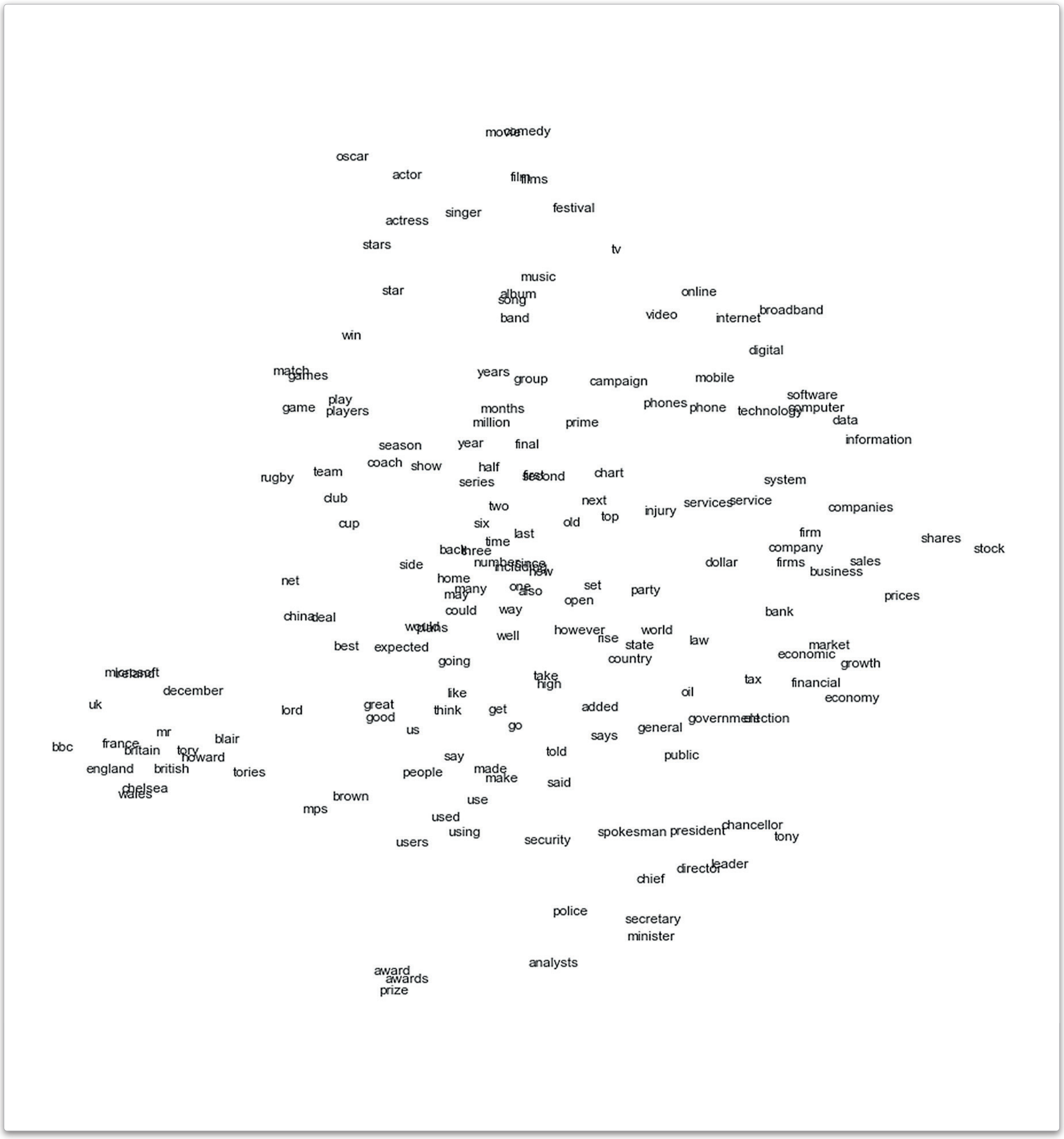


Figure 3.22: Representation of the most frequent words from BBC dataset

The plot verifies that the Word2Vec embeddings successfully capture the semantic associations between words, as indicated by intuitive word groups such as:

- economy, economic, business, financial, sales, bank, firm, firms
- Internet, mobile, phones, phone, broadband, online, digital
- actor, actress, film, comedy, films, festival, band, movie
- game, team, match, players, coach, injury, club, rugby

Sentence Vectorization with Deep Learning

Even though Word2Vec can be used to model individual words, clustering requires the vectorization of entire documents. One of the most popular methods for this purpose is Sentence-BERT (SBERT), which is based on deep learning methods.

Bidirectional Encoder Representations from Transformers (BERT)

BERT is a powerful language representation model developed by Google. Pre-training and fine-tuning are the main factors to which BERT can apply transfer learning; the ability to retain information for one problem and apply it to solve the other. Pre-training is done by feeding the model a massive amount of unlabeled data for multiple tasks, such as masked language prediction (random words in an input text are masked, and the task is to predict these words). For fine-tuning, the BERT model is first initialized with the pre-trained parameters, and all of the parameters are fine-tuned using labeled datasets from the downstream tasks. Each downstream task has separate fine-tuned models, even though they are initialized with the same pre-trained parameters. For example, the fine-tuning sentiment analysis model is different from the question-answering model. Interestingly, the models will have little to no architectural difference after the fine-tuning step.

SBERT

SBERT is a modified version of BERT. Similar to Word2Vec, BERT is trained to predict words based on the context of their sentence. On the other hand, SBERT is trained to predict whether two sentences are semantically similar.

SBERT can be effectively used to create embeddings for pieces of text that are longer than a sentence such as paragraphs or short documents or articles in the BBC dataset that is used in this unit.

Even though all three models are based on neural networks, BERT and SBERT follow significantly different and more complex architectures than Word2Vec.

Sentence_transformers Library

The 'sentence_transformers' library implements the full functionality of the SBERT model. The library comes with several pre-trained SBERT models, each trained on a different dataset and with different objectives. The following code loads one of the most popular general-purpose pre-trained models and uses it to create embeddings for the documents in the BBC dataset:

```
%%capture
!pip install sentence_transformers
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2') # load the pre-trained model.

text_emb = model.encode(bbc_docs) # embed the BBC documents.
```

The same TSNEVisualizer tool that was used earlier in this unit to visualize the vectorized documents produced by the TF-IDF vectorizer can now be used for the embeddings produced by SBERT:

```
tsne = TSNEVisualizer(colors=['blue','green','red','yellow','brown'])
tsne.fit(text_emb,bbc_labels)
tsne.show();
```

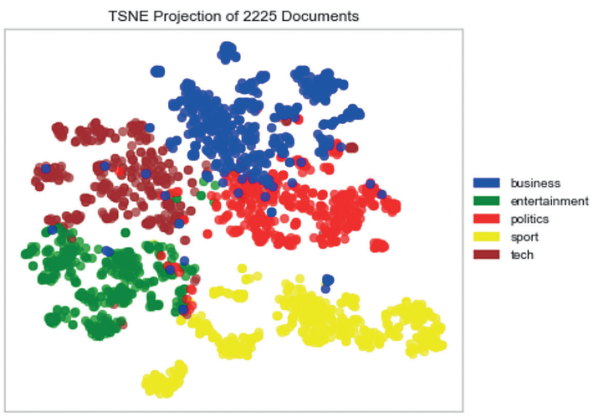


Figure 3.23: TSNE Projection of embeddings by SBERT

The figure reveals that SBERT leads to a more distinct separation of the different news sections, with fewer impurities than TF-IDF. The next step is to to use the embeddings to inform the Agglomerative Clustering algorithm:

```
plt.figure() # create a new figure.

# iteratively merge points and clusters until all points belong to a single cluster. Return the the linkage of
the produced tree.
linkage_emb=hierarchy.linkage(text_emb, method='ward')

hierarchy.dendrogram(linkage_emb) # visualize the linkage.
plt.show() # show the figure.
```

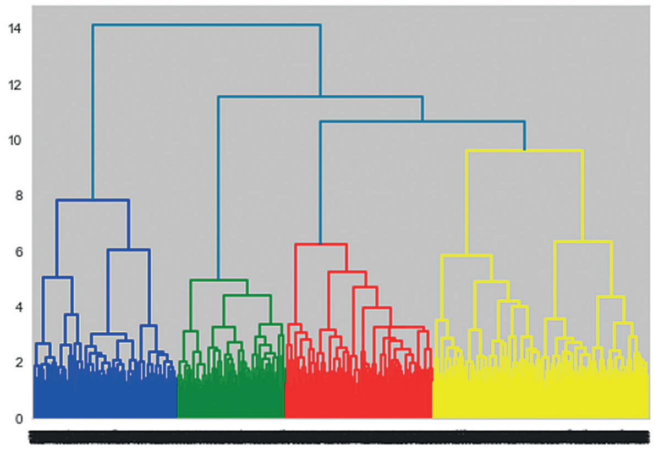


Figure 3.24: Hierarchy dendrogram for SBERT

The dendrogram tool suggests the use of 4 clusters, each marked with a different color in the figure 3.24. The following code uses this suggestion to compute the clusters and compute the evaluation metrics:

```
AC_emb=AgglomerativeClustering(linkage='ward',n_clusters=4)
AC_emb.fit(text_emb)
pred_emb=AC_emb.labels_

print('\nHomogeneity score:',homogeneity_score(bbc_labels,pred_emb))
print('\nAdjusted Rand score:',adjusted_rand_score(bbc_labels,pred_emb))
print('\nCompleteness score:',completeness_score(bbc_labels,pred_emb))
```

```
Homogeneity score: 0.6741395570357063

Adjusted Rand score: 0.6919474005627763

Completeness score: 0.7965514907905805
```

If the data is re-clustered using the correct number of 5 clusters, then the yellow cluster marked in the figure above would be split into two. The results are then as follows:

```
AC_emb=AgglomerativeClustering(linkage='ward',n_clusters=5)
AC_emb.fit(text_emb)
pred_emb=AC_emb.labels_

print('\nHomogeneity score:',homogeneity_score(bbc_labels,pred_emb))
print('\nAdjusted Rand score:',adjusted_rand_score(bbc_labels,pred_emb))
print('\nCompleteness score:',completeness_score(bbc_labels,pred_emb))
```

```
Homogeneity score: 0.7865655030556284

Adjusted Rand score: 0.8197670431956582

Completeness score: 0.7887580797775077
```

The results verify that using SBERT for text vectorization leads to significantly improved clustering results when compared with TF-IDF. In fact, even if the number of clusters is set to 5 (the correct value) for TF-IDF and to 4 for SBERT, SBERT still scores much higher for all three metrics. The gap then becomes even larger if the number is set to 5 for both approaches.

This is a testament to the potential of neural networks, whose sophisticated architecture allows them to understand the complex semantic patterns found in text data.

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. In Unsupervised learning, you use labeled datasets to train the model.	○	○
2. Unsupervised learning requires the vectorization of the data.	○	○
3. SBERT is more optimal than TD-IDF for word vectorization.	○	○
4. Agglomerative Clustering follows a up-bottom approach to cluster selecting.	○	○
5. SBERT is trained to predict whether two sentences are semantically different.	○	○

2 Show examples of applications for which Dimensionality Reduction can be used. Describe the techniques that are used in Dimensionality Reduction.

3 Describe the functionality of TF-IDF vectorization.

4 You are given a numPy array 'Docs' that includes one text document in each row. You are also given an array 'labels' that includes the label for each doc in Docs. Complete the following code so that it uses a pre-trained SBERT model to compute the embeddings for all the documents in Docs and then uses the TSNEVisualizer tool to visualize the embeddings in 2-dimensional space, using a different color for each of the four possible labels.

```
from sentence_transformers import ____________________

from _______________ import TSNEVisualizer model =_________________('all-MiniLM-
L6-v2') # loads the pre-trained model.

docs_emb = model.________________(Docs) # embeds the docs

tsne = ____________________(___________________=['blue','green','red','yellow'])

tsne._______________(___________________,___________________)

tsne.show();
```

5 Complete the following code so that it uses Word2Vec to replace every word in a given sentence with its most similar one.

```
import gensim.downloader as ___________________
import re

model_wv = ________________.___________________('word2vec-google-news-300')
old_sentence='My name is John and I like basketball.'
new_sentence=''

for word in re.___________________(r'\b\w\w+\b',old_sentence.lower()):

       replacement=model_wv.______________(positive=['apple'], ______________=1)[0]

       new_sentence+=____________________
sentence=new_sentence.strip()
```

Lesson 3

Generating Text

Link to digital lesson

www.ien.edu.sa

Natural Language Generation

Natural Language Generation (NLG) is a sub-field of natural language processing (NLP) that focuses on generating human-like text using computer algorithms. The goal of NLG is to produce written or spoken language that is natural and understandable to humans, without the need for human intervention. There are several different approaches to NLG, including template-based, rule-based, and machine learning-based methods.

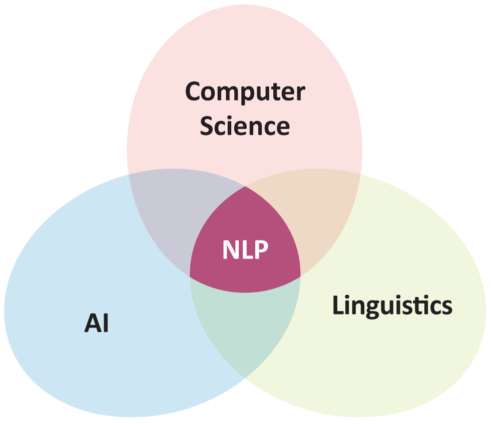


Figure 3.25: NLP Venn diagram

Natural Language Processing (NLP)

Natural Language Processing (NLP) is a branch of AI which gives computers the ability to simulate human natural languages.

Natural Language Generation (NLG)

Natural Language Generation (NLG) is the process of generating human-like text using AI.

Table 3.4: The impact of NLG

	NLG could be used to automatically generate news articles, reports, or other written content, freeing up time for humans to focus on more creative or higher-level tasks.
	It could also be used to improve the efficiency and effectiveness of customer service chatbots, enabling them to provide more natural and helpful responses to customer inquiries.
	NLG has the potential to increase accessibility for people with disabilities or language barriers, by enabling them to communicate with machines in a way that is natural and intuitive for them.

There are four types of NLGs:

Template-Based NLG

Template-based NLG involves the use of predefined templates that specify the structure and content of the generated text. These templates are filled in with specific information to generate the final text. This approach is relatively simple and can be effective at generating text for specific, well-defined tasks. On the other hand, it may struggle with more open-ended tasks or tasks that require a high degree of variability in the generated text. For example, a weather report template might look like this: "Today in **[city]**, it is **[temperature]** degrees with **[weather condition]**."

Selection-Based NLG

Selection-Based NLG involves the selection of a subset of sentences or paragraphs to create a representative summary of a much larger corpus. Even though this approach does not generate new text, it is very popular in practice. This is because, by sampling from a pool of sentences that have been written by humans, it removes the risk of generating unpredictable or poorly formed text. For example, a selection-based weather report generator might have a database of phrases such as "It is hot outside," "The temperature is rising," and "Expect sunny skies."

Rule-Based NLG

Rule-based NLG uses a set of predefined rules to generate text. The rules might specify how to combine words and phrases to form sentences, or how to choose words based on the context in which they are being used. They are often used to create customer service chatbots. Rule-based systems can be simple to implement. They can also be inflexible and may not produce very natural-sounding output.

Machine Learning-Based NLG

Machine learning-based NLG involves training a machine learning model on a large dataset of human-generated text. The model learns the patterns and structure of the text, and can then generate new text that is similar in style and content. This approach can be more effective for tasks that require a high degree of variability in the generated text. This approach may require a larger amount of training data and computational resources.

Using Template-Based NLG

Template-Based NLG is relatively simple and can be effective at generating text for specific, well-defined tasks, such as generating reports or descriptions of data.

One advantage of template-based NLG is that it can be relatively easy to implement and maintain. The templates can be designed by humans, and do not require the use of complex machine learning algorithms or large amounts of training data. This makes template-based NLG a good choice for tasks where the structure and content of the generated text are well-defined and do not need to vary significantly.

NLG templates can be based on any predefined linguistic construct. One common practice is to create a template that requires words with a specific part-of-speech tag to be placed in specific slots within a sentence.

Part of Speech (POS) Tags

Part of speech tags, also known as POS tags, are labels that are assigned to words in a text to indicate their grammatical role, or part of speech, in the sentence. For example, a word may be tagged as a noun, verb, adjective, adverb, etc. Part of speech tags are used in NLP to analyze and understand the structure and meaning of a text.

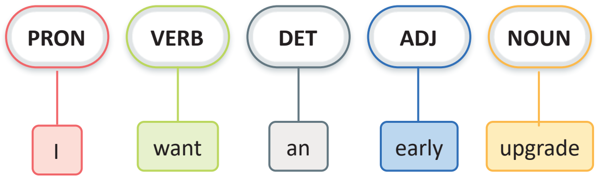


Figure 3.26: Example of POS process

Syntax Analysis

Syntax analysis is often used along with POS tags in template-based NLG, to ensure that the templates can lead to realistic text. Syntax analysis involves identifying the parts of speech of the words in the sentence, and the relationships between them, to determine the grammatical structure of the sentence. A sentence includes different types of syntax elements. For example:

- The **predicate** is the part of the sentence that contains the verb. It typically expresses what is being done or what is happening.
- The **subject** is the part of the sentence that performs the action expressed by the verb, or that is affected by the action.
- The **direct object** is a noun or pronoun that refers to the person or thing that is directly affected by the action expressed by the verb.

The following code uses the wonderwords library, which follows this syntax-based approach, to provide some examples of template-based NLG:

```
%%capture

!pip install wonderwords
# used to generate template-based randomized sentences
from wonderwords.random_sentence import RandomSentence

# make a new generator with specific words
generator=RandomSentence(
                         # specify some nouns
                         nouns=["lion", "rabbit", "horse","table"],
                         verbs=["eat","run","laugh"], # specify some verbs.
                         adjectives=['angry','small']) # specify some adjectives.

# generates a sentence with the following template: [subject (noun)] [predicate (verb)]
generator.bare_bone_sentence()
```

```
'The table runs.'
```

```
# generates a sentence with the following template:
# the [(adjective)] [subject (noun)] [predicate (verb)] [direct object (noun)]
generator.sentence()
```

```
'The small lion runs rabbit.'
```

The above examples show that, while template-based NLG can be used to generate sentences with a specific pre-approved structure, these sentences may be not be that meaningful in practice. Even though the quality of the results can be significantly improved by defining more sophisticated templates and placing more constraints on vocabulary use, this approach is not practical for generating realistic text on a large scale. Rather than manually creating predefined templates, a different approach to template-based NLG is to use the structure and vocabulary of any real sentence as a more dynamic template. The paraphrase() function adopts this approach.

fx Paraphrase() Function

Given a paragraph of text, the function first splits the text into sentences. Then tries to replace each word in the sentence with another semantically similar word. Semantic similarity is evaluated via the Word2Vec model that was introduced in the previous lesson.

To avoid cases where Word2Vec recommends replacements that are very similar to the original word (e.g. replacing "apple" with "apples"), the function uses the popular fuzzywuzzy library to evaluate the lexical similarity between the original word and a candidate to replace it.

The function itself is then shown below:

```
def paraphrase(text:str, # text to be paraphrased
               stop:set, # set of stopwords
               model_wv,# Word2Vec Model
               lexical_sim_ubound:float, # upper bound on lexical similarity
               semantic_sim_lbound:float # lower bound on semantic similarity
              ):

    words=word_tokenize(text) # tokenizes the text to words

    new_words=[] # new words that will replace the old ones.

    for word in words: # for every word in the text

        word_l=word.lower() # lower-case the word.

        # if the word is a stopword or is not included in the Word2Vec model, do not try to replace it.
        if word_l in stop or word_l not in model_wv:
            new_words.append(word) # append the original word

        else: # otherwise

            # get the 10 most similar words, as per the Word2Vec model.
            # returned words are sorted from most to least similar to the original.
            # semantic similarity is always between 0 and 1.
            replacement_words=model_wv.most_similar(positive=[word_l],
topn=10)
            # for each candidate replacement word
            for rword, sem_sim in replacement_words:
                # get the lexical similarity between the candidate and the original word.
                # the partial_ratio function returns values between 0 and 100.
                # it compares the shorter of the two words with all equal-sized substrings
                # of the original word.
                lex_sim=fuzz.partial_ratio(word_l,rword)

                # if the lexical sim is less than the bound, stop and use this candidate.
                if lex_sim<lexical_sim_ubound:
                    break
```

fuzz denotes the fuzzywuzzy library.

```
            # quality check: if the chosen candidate is not semantically similar enough to
            # the original, then just use the original word.
            if sem_sim<semantic_sim_lbound:
                new_words.append(word)
            else: # use the candidate.
                new_words.append(rword)

    return ' '.join(new_words) # re-join the new words into a single string and return.
```

Returns a paraphrased version of the given text.

The following code imports all the tools required to support the paraphrase() function and in the white box below is displayed the output of the **paraphrase** method for the text assigned to the text variable:

```
%%capture

import gensim.downloader as api # used to download and load a pre-trained Word2Vec model
model_wv = api.load('word2vec-google-news-300')

import nltk
# used to split a piece of text into words. Maintains punctuations as separate tokens
from nltk import word_tokenize
nltk.download('stopwords') # downloads the stopwords tool of the nltk library
# used to get list of very common words in different languages
from nltk.corpus import stopwords
stop=set(stopwords.words('english')) # gets the list of english stopwords

!pip install fuzzywuzzy[speedup]
from fuzzywuzzy import fuzz

text='We had dinner at this restaurant yesterday. It is very close to my
house. All my friends were there, we had a great time. The location is
excellent and the steaks were delicious. I will definitely return soon, highly
recommended!'
# parameters: target text, stopwords, Word2Vec model, upper bound on lexical similarity, lower bound
on semantic similarity
paraphrase(text, stop, model_wv, 80, 0.5)
```

```
'We had brunch at this eatery Monday. It is very close to my bungalow. All
my acquaintances were there, we had a terrific day. The locale is terrific
and the tenderloin were delicious. I will certainly rejoin quickly, hugely
advised!'
```

As with any template-based approach, the results can be improved by adding more constraints to correct some of the less intuitive replacements shown above. However, the example above demonstrates that even this simple function can produce very realistic text.

Using Selection-Based NLG

In this section, you will see a practical approach to selecting a sample of representative sentences from a given document. The approach exemplifies the use and benefits of selection-based NLG and relies on two key building blocks:

- The Word2Vec model, which will be used to identify pairs of semantically similar words.
- The Networkx library, a popular python library used to create and process different types of network data.

The input document that will be used in this chapter is a news article written after the final match of the FIFA World Cup 2022.

```
# reads the input document that we want to summarize
with open('article.txt',encoding='utf8',errors='ignore') as f: text=f.read()

text[:100] # shows the first 100 characters of the article
```

```
'It was a consecration, the spiritual overtones entirely appropriate.
Lionel Messi not only emulated '
```

First, the text is tokenized using the **re** library and the same regular expression that was used in the previous Units:

```
import re # used for regular expressions

# tokenize the document, ignore stopwords, focus only on words included in the Word2Vec model.
tokenized_doc=[word for word in re.findall(r'\b\w\w+\b',text.lower()) if word
not in stop and word in model_wv]

# get the vocabulary (set of unique words).
vocab=set(tokenized_doc)
```

Networkx Library

The vocabulary of the document can now be modeled as a weighted graph. Python's Networkx library provides an extensive set of tools for creating and analyzing graphs. In Selection-Based NLG, representing the vocabulary of a document as a weighted graph can help to capture the relationships between words and facilitate the selection of relevant phrases and sentences. In a weighted graph, each node represents a word or a concept, and the edges between nodes represent relationships between these concepts. The weights on the edges represent the strength of these relationships, allowing the NLG system to determine which concepts are most strongly related. When generating text, the weighted graph can be used to find the most relevant phrases and sentences based on the relationships between words. For example, the system might use the graph to find the most relevant words and phrases to describe a particular entity and then use these words to select the most appropriate sentence from its database.

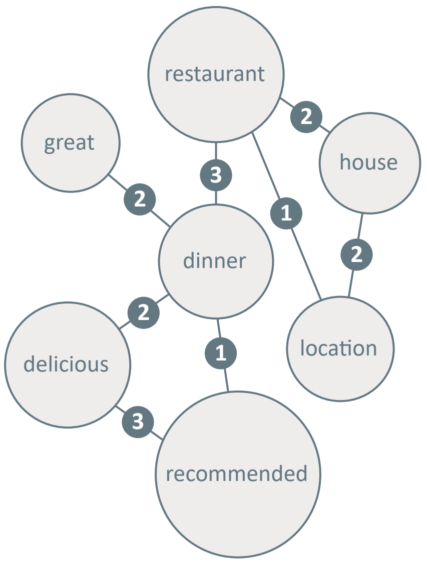


Figure 3.27: Example of a Networkx weighted graph

fx Build_graph() Function

The build_graph() function uses NetworkX to create a graph that includes:

- One node for each word in a given vocabulary.
- An edge between every two words. The weight on the edge is equal to the semantic similarity between the words, as computed by Doc2Vec which is an NLP tool for representing documents as a vector and is a generalization of the word2vec method

The function returns a graph with one node for each word in the given vocabulary. There is also an edge between two nodes if their Word2Vec similarity is higher than the given threshold.

```
# tool used to create combinations (e.g. pairs, triplets) of the elements in a list
from itertools import combinations
import networkx as nx # python library for processing graphs

def build_graph(vocab:set, # set of unique words
                model_wv # Word2Vec model
               ):
    # gets all possible pairs of words in the doc
    pairs=combinations(vocab,2)

    G=nx.Graph() # makes a new graph

    for w1,w2 in pairs: # for every pair of words w1, w2
        sim=model_wv.similarity(w1, w2) # gets the similarity between the two words
        G.add_edge(w1,w2,weight=sim)

    return G

# creates a graph for the vocabulary of the World Cup document
G=build_graph(vocab,model_wv)
# prints the weight of the edge (semantic similarity) between the two words
G['referee']['goalkeeper']
```

```
{'weight': 0.40646762}
```

Given such a word-based graph, a set of words that are all semantically similar to each other can be represented as a cluster of nodes connected to each other by high-weight edges. Such node clusters are also referred to as "communities". The graph output is a simple set of vertices and set of weighted edges. No clustering has been done yet to create the "communities". Figure 3.28 uses different colors to mark the communities in an example graph:

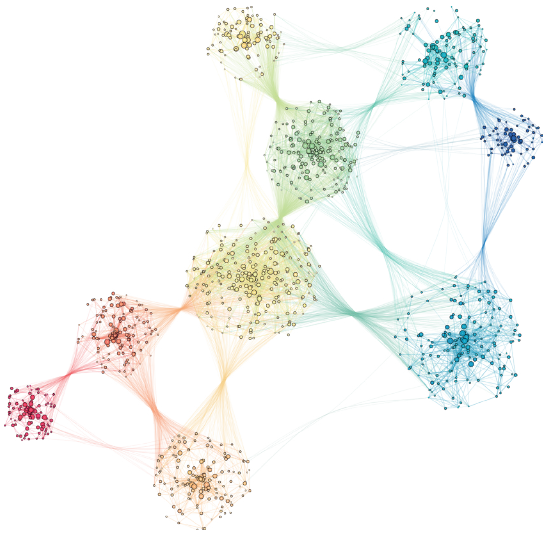


Figure 3.28: Communities in a graph

Louvain Algorithm

The Networkx library includes multiple algorithms for analyzing the graph and finding such communities. One of the most effective options is the Louvain algorithm, which works by iteratively moving nodes between communities until it finds the community structure that best represents the linkage of the underlying network.

fx Get_communities() Function

The following function uses the Louvain algorithm to find the communities in a given word-based Graph. The function also computes an importance score for each community. Then it returns two dictionaries:

- word_to_community, which maps each word to its community.
- community_scores, which maps each community to an importance score.

The score is equal to the sum of the frequencies of all the words in the community. For example, if a community includes three words that appear 5, 8, and 6 times in the document, the community's score is equal to 19. Conceptually, the score represents the part of the document that is "covered" by the community.

```python
from networkx.algorithms.community import louvain_communities
from collections import Counter # used to count the frequency of elements in a list

def get_communities( G, # the input graph
                     tokenized_doc:list): # the list of words in a tokenized document

     # gets the communities in the graph
    communities=louvain_communities(G, weight='weight')
    word_cnt=Counter(tokenized_doc)# counts the frequency of each word in the doc

    word_to_community={}# maps each word to its community

    community_scores={}# maps each community to a frequency score

    for comm in communities: # for each community
         # convert it from a set to a tuple so that it can be used as a dictionary key.
        comm=tuple(comm)
        score=0  # initialize the community score to 0.

        for word in comm: # for each word in the community

            word_to_community[word]=comm  # map the word to the community

            score+=word_cnt[word]  # add the frequency of the word to the community's score.

        community_scores[comm]=score  # map the community to the score.

    return word_to_community, community_scores
```

```
word_to_community, community_scores = get_communities(G,tokenized_doc)
word_to_community['player'][:10] # prints 10 words from the community of the word 'team'
```

```
('champion',
 'stretch',
 'finished',
 'fifth',
 'playing',
 'scoring',
 'scorer',
 'opening',
 'team',
 'win')
```

Now that all the words have been mapped to a community and each community is associated with an importance score, the next step is to use this information to evaluate the importance of each sentence in the original document. The evaluate_sentences() function is designed for this purpose.

fx Evaluate_sentences() Function

The function starts by splitting the document into sentences. It then computes an importance score for each sentence, based on the words that it includes. Each word inherits the importance score of the community that it belongs to.

For example, consider a sentence with 5 words w1, w2, w3, w4, w5. Words w1 and w2 belong to a community with a score of 25, w3 and w4 belong to a community with a score of 30, and w5 belongs to a community with a score of 15. The total score of the sentence is then 25+25+30+30+15=125. The function then uses these scores to rank the sentences in descending order, from most to least important.

```
from nltk import sent_tokenize # used to split a document into sentences

def evaluate_sentences(doc:str, # original document
                       word_to_community:dict,# maps each word to its community
                       community_scores:dict, # maps each community to a score
                       model_wv): # Word2Vec model

    # splits the text into sentences
    sentences=sent_tokenize(doc)
    scored_sentences=[]# stores (sentence, score) tuples

    for raw_sent in sentences: # for each sentence

        # get all the words in the sentence, ignore stopwords and focus only on words that are in the
Word2Vec model
        sentence_words=[word
        for word in re.findall(r'\b\w\w+\b',raw_sent.lower()) # tokenizes
            if word not in stop and   # ignores stopwords
```

```
                word in model_wv] # ignores words that are not in the Word2Vec model

        sentence_score=0 # the score of the sentence

        for word in sentence_words: # for each word in the sentence

            word_comm=word_to_community[word] # get the community of this word
            sentence_score+=community_scores[word_comm] # add the score of this
community to the sentence score.

        scored_sentences.append((sentence_score,raw_sent)) # stores this sentence and
its total score

    # scores the sentences by their score, in descending order
    scored_sentences=sorted(scored_sentences,key=lambda x:x[0],reverse=True)

    return scored_sentences


scored_sentences=evaluate_sentences(text,word_to_community,community_
scores,model_wv)
len(scored_sentences)
```

```
61
```

The original doc includes a total of 61 sentences. The following code can now be used to get the top 3 most important of these sentences:

```
for i in range(3):
    print(scored_sentences[i],'\n')
```

```
(3368, 'Lionel Messi not only emulated the deity of Argentinian football,
Diego Maradona, by leading the nation to World Cup glory; he finally
plugged the burning gap on his CV, winning the one title that has eluded
him – at the fifth time of asking, surely the last time.')

(2880, 'He scored twice in 97 seconds to force extra-time; the first a
penalty, the second a sublime side-on volley and there was a point towards
the end of regulation time when he appeared hell-bent on making sure that
the additional period would not be needed.')

(2528, 'It will go down as surely the finest World Cup final of all time,
the most pulsating, one of the greatest games in history because of how
Kylian Mbappé hauled France up off the canvas towards the end of normal
time.')
```

```
print(scored_sentences[-1]) # prints the last sentence with the lowest score
print()
print(scored_sentences[30]) # prints a sentence at the middle of the scoring scale
```

```
(0, 'By then it was 2-0.')

(882, 'Di María won the opening penalty, exploding away from Ousmane
Dembélé before being caught and Messi did the rest.')
```

The results verify this approach can indeed successfully identify representative sentences that capture the main points of the original document, while assigning lower scores to less informative sentences. The same approach can be applied as is to generate a summary of any given document.

Using Rule-Based NLG to Create a Chatbot

In this section, you will build a course-recommendation chatbot by combining a simple knowledge base of questions and answers with the SBERT neural model. This demonstrates the transfer learning used in SBERT as the same architecture of SBERT (all-MiniLM-L6-v2) will now be fine-tuned to a task other than sentiment analysis: NLG.

1. Load the Pre-Trained SBERT Model

The first step is to load the pre-trained SBERT model:

```
%%capture
from sentence_transformers import SentenceTransformer, util
model_sbert = SentenceTransformer('all-MiniLM-L6-v2')
```

2. Create a Simple Knowledge Base

The second step is to create a simple knowledge base to capture the question-answer script that the chatbot will follow. The script includes 4 questions (Q1-Q4) and their respective answers (A1-A4). Each answer consists of a list of options. The second cell represents the next question that the chatbot will get to. If it is the final question, the second cell will have **None**. These options represent the possible answers that are considered acceptable for the corresponding questions. For example, the answer to question Q2 has two possible options (["Java",None] and ["Python",None]).

Each option consists of two values:

- The actual text of the acceptable answer (e.g. "Java" or "Courses on Marketing").
- An ID that points that to the next question that the chatbot should ask if the option is selected. For example, if the user selects the ["Courses on Engineering","3"] option as a response to Q1 then the next question that will be asked is Q3.

This simple knowledge can be easily extended to add more Q/A levels and make the chatbot more intelligent.

```
QA={
       "Q1":"What type of courses are you interested in?",
       "A1":[["Courses in Computer Programming","2"],
             ["Courses in Engineering","3"],
             ["Courses in Marketing","4"]],

       "Q2":"What type of Programming Languages are you interested in?",
       "A2":[["Java",None],["Python",None]],

       "Q3":"What type of Engineering are you interested in?",
       "A3":[["Mechanical Engineering",None],["Electrical Engineering",None]],

       "Q4":"What type of Marketing are you interested in?",
       "A4":[["Social Media Marketing",None],["Search Engine
Optimization",None]]
}
```

fx Chat() Function

Finally, the following chat() function is used to process the knowledge base and implement the chatbot. After asking a question, the chatbot reads the user's response.

- If the response is semantically similar to one of the acceptable answer options for this question, then that option is selected and the chatbot proceeds to the next question.
- If the response is not similar to any of the options, then it asks the user to rephrase the response.

The function uses SBERT to evaluate the semantic similarity score between the response and each candidate option. An option is considered similar if this score is higher than a lower bound parameter (sim_lbound).

```
import numpy as np # used for processing numeric data

def chat(QA:dict, # the Question-Answer script of the chatbot
        model_sbert, # a pre-trained SBERT model
        sim_lbound:float): # lower bound on the similarity between the user's response and the
closest candidate answer

    qa_id='1' # the QA id

    while True: # an infinite loop, will break in specific conditions

        print('>>',QA['Q'+qa_id]) # prints the question for this qa_id
        candidates=QA["A"+qa_id] # gets the candidate answers for this qa_id

        print(flush=True) # used only for formatting purposes
        response=input() # reads the user's response

        # embed the response
        response_embeddings = model_sbert.encode([response], convert_to_
tensor=True)
        # embed each candidate answer. x is the text, y is the qa_id. Only embed x.
        candidate_embeddings = model_sbert.encode([x for x,y in candidates],
convert_to_tensor=True)

        # gets the similarity score for each candidate
        similarity_scores = util.cos_sim(response_embeddings, candidate_
embeddings)

        # finds the index of the closest answer.
        # np.argmax(L) finds the index of the highest number in a list L
        winner_index=np.argmax(similarity_scores[0])

        # if the score of the winner is less than the bound, ask again.
        if similarity_scores[0][winner_index]<sim_lbound:
            print('>> Apologies, I could not understand you. Please rephrase
your response.')
            continue

        # gets the winner (best candidate answer)
        winner=candidates[winner_index]

        # prints the winner's text
        print('\n>> You have selected:',winner[0])
        print()

        qa_id=winner[1] # gets the qa_id for this winner

        if qa_id==None: # no more questions to ask, exit the loop
            print('>> Thank you, I just emailed you a list of courses.')
            break
```

Consider the following two interactions between the chatbot and a user:

Interaction 1

```
chat(QA,model_sbert, 0.5)
```

```
>> What type of courses are you interested in?

marketing courses

>> You have selected: Courses on Marketing

>> What type of Marketing are you interested in?

seo

>> You have selected: Search Engine Optimization

>> Thank you, I just emailed you a list of courses.
```

In this first interaction, the chatbot correctly understands that the user is looking for Marketing courses. It is also intelligent enough to understand that the term "SEO" is semantically similar to "Search Engine Optimization", leading to the successful conclusion of the discussion.

Interaction 2

```
chat(QA,model_sbert, 0.5)
```

```
>> What type of courses are you interested in?

cooking classes
>> Apologies, I could not understand you. Please rephrase your response.
>> What type of courses are you interested in?

software courses

>> You have selected: Courses on Computer Programming

>> What type of Programming Languages are you interested in?

C++

>> You have selected: Java

>> Thank you, I just emailed you a list of courses.
```

In this second interaction, the chatbot correctly realizes that "Cooking Classes" is not semantically similar to any of the options in its knowledge base. It is also intelligent enough to understand that "Software courses" should be mapped to the "Courses on Computer Programming" option.

The final part of the interaction highlights a weakness: the chatbot matches the user's "C++" response to "Java". Even though the two programming languages are indeed related (and are arguably more related than Python and C++), the appropriate response would have been to say that the chatbot does not have the knowledge to recommend C++ courses.

One way to address this weakness would be to use lexical rather than semantic similarity to compare responses and options for some questions.

Using Machine Learning to Generate Realistic Text

The methods described in the previous sections use templates, rules, or selection techniques to produce text for different applications. In this section, you will explore the state-of-the-art in machine learning for NLG.

Table 3.5: Advanced machine learning techniques for NLG

Technique	Description
Long short-term memory (LSTM) network	An LSTM network is made up of several "memory cells" that are connected together. When the network is given a sequence of data, it processes each element in the sequence one at a time and for each element, the network updates its memory cells to produce an output. LSTMs are particularly well-suited for NLG tasks because they can retain information from sequences of data (such as speech or handwriting recognition) and handle the complexity of natural language.
Transformer-based models	Transformer-based models are models that can understand and generate human language. They work by using a technique called "self-attention" that helps them understand the relationships between different words in a sentence.

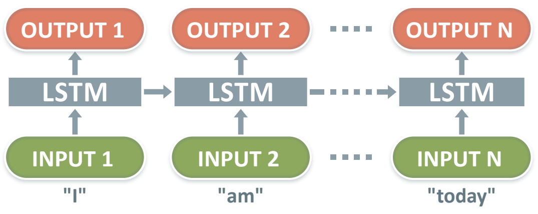


Figure 3.30: LSTM

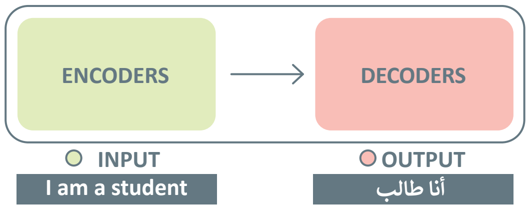


Figure 3.29: Transformer

Transformers

Transformers are particularly well-suited for NLG tasks because they can process sequential input data efficiently. In a transformer model, the input data is first passed through an encoder, which converts the input into a continuous representation. The continuous representation is then passed through a decoder, which generates the output sequence. One of the key features of these models is the use of attention mechanisms that allow the model to focus on the important parts of a sequence while ignoring less informative parts. Transformer models have been shown to produce high-quality text for a variety of NLG tasks, including machine translation, summarization, and question answering.

OpenAI GPT-2 Model

In this section, you will use GPT-2, a powerful language model developed by OpenAI, to generate text based on text prompts that are provided by the user. GPT-2 (Generative Pre-training Transformer 2) was trained on a dataset of over 8 million web pages and has the ability to generate human-like text in a variety of languages and styles. The transformer-based architecture of GPT-2 allows it to capture long-range dependencies and generate coherent text. GPT-2 is trained with the objective of predicting the next word, given all of the previous words within the text. The model can thus be used to produce texts of arbitrary length, by continuously predicting and appending more words.

```
%%capture
!pip install transformers
!pip install torch
import torch # an open-source machine learning library for neural networks, required for GPT2.
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# initialize a tokenizer and a generator based on a pre-trained GPT2 model.

# used to:
# -encode the text provided by the user into tokens
# -translate (decode) the output of the generator back to text
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# used to generate new tokens based on the inputted text
generator = GPT2LMHeadModel.from_pretrained('gpt2')
```

The following text will then be provided as a seed to GPT-2:

```
text='We had dinner at this restaurant yesterday. It is very close to my
house. All my friends were there, we had a great time. The location is
excellent and the steaks were delicious. I will definitely return soon, highly
recommended!'
```

```
# encodes the given text into tokens
encoded_text = tokenizer.encode(text, return_tensors='pt')

# use the generator to generate more tokens.
# do_sample=True prevents GPT-2 from just predicting the most likely word at every step.
generated_tokens = generator.generate(encoded_text,
                                      max_length=200) # max number of new tokens to
generate
#decode the generates tokens to convert them to words
# skip_special_tokens=True is used to avoid special tokens such as '>' or '-' characters.
print(tokenizer.decode(generated_tokens[0], skip_special_tokens=True))
```

```
We had dinner at this restaurant yesterday. It is very close to my house.
All my friends were there, we had a great time. The location is excellent
and the steaks were delicious.I will definitely return soon, highly
recommended!

I've been coming here for a while now and I've been coming here for a while
now and I've been coming here for a while now and I've been coming here for
a while now and I've been coming here for a while now and I've been coming
here for a while now and I've been coming here for a while now and I've
been coming here for a while now and I've been coming here for a while now
and I've been coming here for a while now and I've been coming here for a
while now and I've been coming here for a while now and I've been coming
here for a while now and I've been coming here for a while now and I've
been coming here for a while now and
```

```
# use the generator to generate more tokens.
# do_sample=True prevents GPT-2 from just predicting the most likely word at every step.
generated_tokens = generator.generate(encoded_text,
                                      max_length=200, # max number of new tokens to
generate
                                      do_sample=True)

print(tokenizer.decode(generated_tokens[0],skip_special_tokens=True))
```

```
We had dinner at this restaurant yesterday. It is very close to my house.
All my friends were there, we had a great time. The location is excellent
and the steaks were delicious.I will definitely return soon, highly
recommended!

If you just found this place helpful. If you like to watch videos or
go to the pool while you're there, go for it! Good service - I'm from
Colorado and love to get in and out of this place. The food was amazing!
Also, we were happy to see the waitstaff with their great hands - I went
for dinner. I ordered a small side salad (with garlic on top), and had a
slice of tuna instead. When I was eating, I was able to get up and eat my
salad while waiting for my friend to pick up the plate, so I had a great
time too. Staff was welcoming and accommodating. Parking is cheap in this
neighborhood, and it is in the neighborhood that it needs to
```

This leads to a much more diverse output, while maintaining the authenticity of the generated text. The text uses a rich vocabulary and is syntactically correct.

GPT-2 allows for the further customization of the output. An example is the use of the 'temperature' parameter, which allows the model to take more risks and to sometimes select some lower-probability words. Higher values of this parameter lead to more diverse texts. For example:

```
# Generate tokens with higher diversity
generated_tokens = generator.generate(
    encoded_text, max_length=200, do_sample=True, temperature=2.0)

print(tokenizer.decode(generated_tokens[0], skip_special_tokens=True))
```

```
We had dinner at this restaurant yesterday. It is very close to my house.
All my friends were there, we had a great time. The location is excellent
and the steaks were delicious.I will definitely return soon, highly
recommended!

Worth a 5 I thought a steak at a large butcher was the end story!! We were
lucky. The price was cheap!! That night though as soon as dinner was on
my turn that price cut completely out. At the tail area they only have
french fries or kiwifet - no gravy - they get a hard egg the other day too
they call kawif at 3 PM it will be better this summer if I stay more late
with friends. When asked it takes 2 or 3 weeks so far to cook that in this
house. Once I found a place it was great. Everything I am waiting is just
perfect as usual....great prices especially at one where a single bite
would suffice or make more as this only runs on the regular hours
```

However, if the temperature is set too high, the model departs from the guidance of the original input and leads to less realistic and meaningful output:

```
# Too high temperature leads to divergence in the meaning of the tokens
generated_tokens = generator.generate(
    encoded_text, max_length=200, do_sample=True, temperature=4.0)

print(tokenizer.decode(generated_tokens[0], skip_special_tokens=True))
```

```
We had dinner at this restaurant yesterday. It is very close to my house.
All my friends were there, we had a great time. The location is excellent
and the steaks were delicious.I will definitely return soon, highly
recommended! It has the nicest ambagas of '98 that I like; most Mexican.
And really nice steak house; amazing Mexican atmosphere to this very
particular piece of house I just fell away before its due date, no surprise
my 5yo one fell in right last July so it took forever at any number on
it being 6 (with it taking two or sometimes 3 month), I really have found
comfort/affability on many more restaurants when ordering.If you try at
it they tell ya all about 2 and three places will NOT come out before they
close them/curry. Also at home i would leave everything until 1 hour but
sometimes wait two nights waiting for 2+ then when 2 times you leave you
wait in until 6 in such that it works to
```

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. Machine Learning-based NLG requires large amounts of training data and computational resources.	●	●
2. Verb could be a POS tag.	●	●
3. In template-based NLG syntax, analysis is used separately from POS tags.	●	●
4. Communities are node clusters that represent semantically different words.	●	●
5. The more Q/A levels added to the chatbot's knowledge base, the smarter it gets.	●	●

2 Compare the different approaches of Natural Language Generation (NLG).

3 State three different applications for NLG.

4 Complete the following code so that the build_graph() function accepts a given vocabulary of words and a trained Word2Vec model and returns a graph with one node for each word in the vocabulary. The graph should have an edge between two nodes if their similarity according to Word2Vec is higher than the given similarity_threshold. There should be no weights on the edges.

```
from ______________ import combinations # tool used to create combinations

import networkx as nx # python library for processing graphs

def build_graph(vocab:set, # set of unique words

                model_wv, # Word2Vec model

                similarity_threshold:float

               ):

    pairs=combinations(vocab,________________) # gets all possible pairs of words in the vocabulary

    G=nx.________________ # makes a new graph

    for w1,w2 in pairs: # for every pair of words w1,w2

        sim=model_wv.__________________(w1, w2)# gets the similarity between the two words

        if _______________:

            G.__________________(w1,w2)

    return G
```

5 Complete the following code so that the function get_max_sim() uses a pre-trained SBERT model to compare a given sentence my_sentence with all the sentences in a second given list of sentences L. The function should then return the sentence from L1 with the highest similarity score to my_sentence.

```
from sentence_transformers import ___________________, util

from _______________ import combinations # tool used to create combinations

model_sbert = ___________________ ('all-MiniLM-L6-v2')

def get_max_sim(L1,my_sentence):

    # embeds my_sentence

    my_embedding = model_sbert,________________([my_sentence], convert_to_tensor=True)

    # embeds the sentences from L2

    L_embeddings = model_sbert._______________(L, convert_to_tensor=True)

    similarity_scores = ________________.cos_sim(_________________, ___________)

    winner_index=np.argmax(similarity_scores[0])

    return _______________
```

Project

Text classification is a 2-step process that includes:

Step 1: Using a set of training documents with known labels (classes) to train a classification model.

Step 2: Using the trained model to predict the label for each document in a testing set. The labels in the testing set are either unknown or hidden and used later for verification.

The documents in both the training and testing sets have to be vectorized before they can be used. The CountVectorizer or TfidfVectorizer tools from the sklearn library can be used for vectorization.

The Python sklearn library offers a long list of classification models. Some of them are:

> GradientBoostingClassifier()

> DecisionTreeClassifier()

> RandomForestClassifier()

Your task is to use the IMDB training set that was used in this lesson to train a model that achieves the highest possible accuracy on the IMDB testing set (imdb_data/imdb_test.csv). You can achieve this by:

1. Replace the MultinomialNB classifier with other classification models from sklearn, such as the ones listed above.
2. Re-run your notebook after each replacement, to compute the accuracy of each new model that you try.
3. Create a report that compares the accuracy of all the models that you tried and identifies the one that achieved the best accuracy.

Wrap up

Now you have learned to:

> Classify text with unsupervised learning models.

> Analyze text with supervised learning models.

> Use Machine Learning models for NLG.

> Program a simple chatbot.

KEY TERMS

Black-Box predictors	Natural Language Generation	Supervised Learning
Chatbot	Natural Language Processing	Syntax Analysis
Cluster	Part of Speech (POS)	Tokenization
Dendrogram	Tags	Transfer Learning
Dimensionality Reduction	Sentiment analysis	Unsupervised Learning
Document Clustering		Vectorization

Part 2

وزارة التعليم
Ministry of Education
2023 - 1445

وزارة التعـــليم
Ministry of Education
2023 - 1445

4. Image Recognition

In this unit, you will learn about supervised and unsupervised learning for image recognition by creating and training a model to classify or cluster images of different animal heads, as an example. You will also learn about image generation and how to alter images or complete their missing content while maintaining realism.

Learning Objectives

In this unit, you will learn to:

> Preprocess images and extract their features.
> Train a supervised learning model to classify images.
> Define the structure of a neural network.
> Train an unsupervised learning model to cluster images.
> Generate images based on a text prompt.
> Realistically complete missing parts of an image.

Tools

> Jupyter Notebook
> Google Colab

Lesson 1

Supervised Learning for Image Analysis

Link to digital lesson

www.ien.edu.sa

Supervised Learning for Computer Vision

Computer vision is a subfield of Artificial Intelligence that focuses on teaching computers how to interpret and understand the visual world. It involves using digital images and videos to train machines to recognize and analyze visual information, such as objects, people, and scenery. The ultimate goal of computer vision is to enable machines to "see" the world as humans do and use this information to make decisions or take actions.

Computer vision has a wide range of applications, such as:

- Medical Imaging: Computer vision can help doctors and healthcare professionals in diagnosing diseases by analyzing medical images, such as X-rays, MRIs, and CT scans.
- Autonomous Vehicles: Self-driving cars and drones use computer vision to recognize traffic signals and road patterns, pedestrians, and obstacles in the road and in the air, enabling them to navigate safely and efficiently.
- Quality Control and Inspection: Computer vision is used to inspect products and identify defects in manufacturing processes. This is used in various industries, including automotive, electronics, and textiles.
- Robotics: Computer vision is used to help robots navigate and interact with their environment, including recognizing and manipulating objects.

Supervised and unsupervised learning are two main types of machine learning that are commonly used in computer vision applications. Both approaches involve training algorithms on large datasets of images or videos to enable machines to recognize and interpret visual information. Supervised learning and unsupervised learning were introduced in unit 3 lessons 1 and 2, and were both applied in NLP and NLG. In this lesson, they will be applied for image analysis.

Unsupervised learning involves training algorithms on unlabeled datasets, where no explicit labels or categories are provided. The algorithm then learns to identify similar patterns in the data without any prior knowledge of the labels. For example, an unsupervised learning algorithm might be used to group similar images together based on common features, such as color, texture, or shape. Unsupervised learning will be detailed in lesson 2.

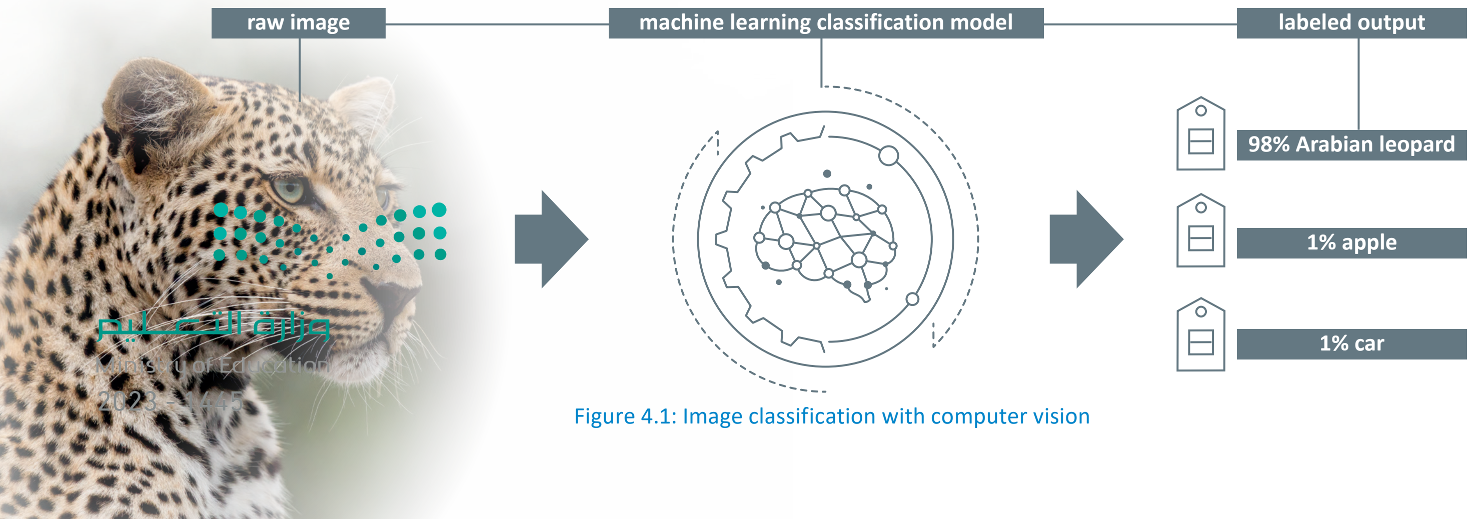


Figure 4.1: Image classification with computer vision

In constrast, supervised learning involves training algorithms on labeled datasets, where each image or video is assigned a specific label or category. The algorithm then learns to recognize patterns and features that are associated with each label, allowing it to accurately classify new images or videos. For example, a supervised learning algorithm might be trained to recognize different breeds of cats based on labeled images of each breed (e.g, see figure 4.1). Supervised learning is the focus of this lesson.

The process of supervised learning typically involves four key steps: data collection, labeling, training, and testing. During data collection and labeling, images or videos are collected and organized into a dataset. Then, each image or video is labeled with a corresponding class or category, such as "eagle" or "cat".

During the training phase, the machine learning algorithm uses this labeled dataset to "learn" the patterns and features that are associated with each class or category. As more training data is presented to the algorithm, it becomes more accurate at recognizing the different classes in the dataset and improves its performance.

Once the model has been trained, it is tested on a separate set of images or videos to evaluate its performance. The testing set is different from the training set to ensure that the model is able to generalize to new data. For example, the data for a Cat has propertirs such as weight, color, breed etc. The accuracy of the model is then evaluated based on how well it performs on the testing set.

The above process is very similar to the one followed for supervised learning tasks on different types of data, such as text. However, visual data is generally considered harder to handle than text due to multiple reasons as mentioned in Table 4.1.

Table 4.1: Challenges of visual data classification

Visual data is high-dimensional	Images contain a large amount of data, which makes them more difficult to process and analyze than textual data. While the basic elements of a text document are words, the elements of an image are pixels. As you will see in this chapter, even a small image can consist of thousands of pixels.
Visual data is noisy and very diverse	Images can be affected by noise, lighting, blurring, and other factors that make it difficult to accurately classify them. In addition, there is a wide variety of visual data, with many different objects, scenes, and contexts that can be difficult to accurately classify.
Visual data does not follow a strict structure	While text tends to follow specific rules for syntax and grammar, visual data does not have such constraints. This makes it harder and more computationally expensive to analyze.

As a result of these complexities, the effective classification of visual data requires specialized techniques. This unit covers techniques that utilize the geometric and color properties of images, besides more advanced machine learning techniques based on neural networks.

Speficially, this first lesson demonstrates how Python can be used for:

• Loading a dataset of labeled images.

• Converting the images to a numeric format that can be used by computer vision algorithms.

• Splitting the numeric data into training and testing datasets.

- Analyzing the data to extract informative patterns and features.
- Using the transformed data to train classification models that can be used to predict the labels of new images.

The dataset you will be using includes 1,730 face images for 16 different types of animals, making it ideal for supervised learning and demonstrating the aforementioned techniques.

Loading and Preprocessing Images

The following code imports a set of libraries that are used to load the images from the LHI-Animal-Faces dataset and convert them to a numeric format.

```
%%capture
import matplotlib.pyplot as plt # used for visualization
from os import listdir # used to list the contents of a directory

!pip install scikit-image # used for image manipulation
from skimage.io import imread # used to read a raw image file (e.g. png or jpg)
from skimage.transform import resize # used to resize images

# used to convert an image to the "unsigned byte" format
from skimage import img_as_ubyte
```

Ensuring that all the images in the dataset have the same dimensions is required by supervised learning algorithms, therefore, the following code reads the images from their input_folder and resizes each of them to the same (width x height) dimensions. :

```
def resize_images(input_folder:str,
                  width:int,
                  height:int
                 ):

    labels = [] # a list with the label for each image
    resized_images = [] # a list of resized images in np array format
    filenames = [] # a list of the original image file names

    for subfolder in listdir(input_folder): # for each sub folder

        print(subfolder)
        path = input_folder + '/' + subfolder

        for file in listdir(path): # for each image file in this subfolder

            image = imread(path + '/' + file) # reads the image
            resized = img_as_ubyte(resize(image, (width, height))) # resizes the image
            labels.append(subfolder[:-4])  # uses subfolder name without "Head" suffix
            resized_images.append(resized) # stores the resized image
            filenames.append(file)         # stores the filename of this image

    return resized_images, labels, filenames
```

```
resized_images, labels, filenames = resize_images("AnimalFace/Image",
width=100, height=100) # retrieves the images with their labels and resizes them to 100 x 100
```

```
BearHead        EagleHead       PigeonHead
CatHead         ElephantHead    RabbitHead
ChickenHead     LionHead        SheepHead
CowHead         MonkeyHead      TigerHead
DeerHead        Natural         WolfHead
DuckHead        PandaHead
```

The names of the folders. Without the "Head" suffix, they serve as the labels of the images contained in them.

The imread() function creates an "RGB" format of the image. This format is widely used because it allows for the representation of a wide range of colors. In the RGB color system, the letters R, G, and B mean that the format contains three major color components, namely red (R = Red), green (G = Green), and blue (B = Blue). Each pixel is represented by three 8-bit channels (one for red, one for green, and one for blue) and can take on a value between 0 and 255. This 0-255 format is also known as the "unsigned byte" format.

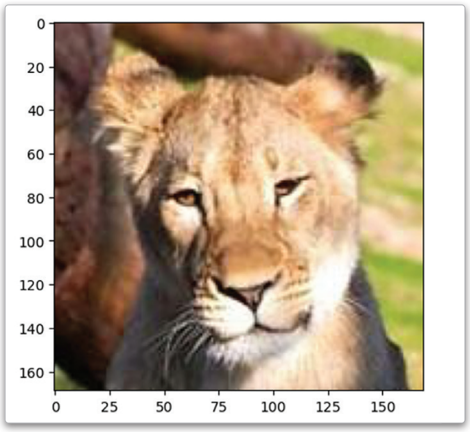


Figure 4.2: Original lion head image

The combination of these three channels allows for the representation of a wide range of colors in the pixel. For example, a pixel with the value (255, 0, 0) would be fully red, a pixel with the value (0, 255, 0) would be fully green, and a pixel with the value (0, 0, 255) would be fully blue. A pixel with the value (255, 255, 255) would be white, and a pixel with the value (0, 0, 0) would be black.

In the RGB system, pixel values are arranged in a two-dimensional grid, with rows and columns representing the x and y coordinates of the pixels in the image. The resulting grid is referred to as the "image matrix."

For example , consider the image in figure 4.2 and the associated code below:

```
# reads an image file, stores it in a variabe and
# shows it to the user in a window
image = imread('AnimalFace/Image/LionHead/lioni78.jpg')
plt.imshow(image)
image.shape
```

```
(169, 169, 3)
```

Printing the image shape reveals a 169x169 matrix, for a total of 28,561 pixels. The "3" in the third column represents the 3 channels (Red/Green/Blue) of the RGB system. For example, the following code would print the RGB value of the first pixel of this image:

```
# the pixel at the first column of the first row
print(image[0][0])
```

```
[102  68  66]
```

Resizing has the effect of converting RGB images to a float-based format:

```
resized = resize(image, (100, 100))
print(resized.shape)
print(resized[0][0])
```

```
(100, 100, 3)
[0.40857161 0.27523827 0.26739514]
```

Even though the image has now indeed been resized to a 100x100 matrix, the 3 RGB values of each pixels have been normalized to a value between 0 and 1. It can be transformed back to the original unsigned byte format via the following code:

```
resized = img_as_ubyte(resized)
print(resized.shape)
print(resized[0][0])
print(image[0][0])
```

```
(100, 100, 3)
[104  70  68]
[102  68  66]
```

The RGB values of the resized pixel are slightly different from those in the original image, which is a common effect of the resizing. Printing the resized image also reveals that it is slightly less clear, as appears in figure 4.3. Again, this is a result of compressing the 169x169 matrix to a 100x100 format.

```
# displays the resized image
plt.imshow(resized);
```

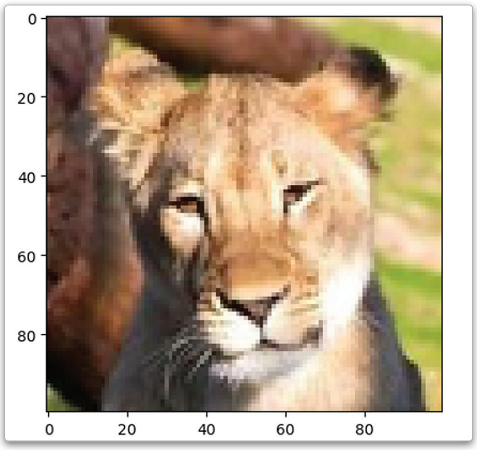


Figure 4.3: Resized lion head image

Before proceeding with the training of supervised learning algorithms, it is good practice to check if any of the images in the dataset violates the (100,100,3) format:

```
violations = [index for index in range(len(resized_images)) if
resized_images[index].shape != (100,100,3)]

violations
```

```
[455, 1587]
```

The code reveals two such images. This is unexpected, given that the resize_images() function was applied to all images in the dataset. The following code snippets print the two images, along with their dimensions and file names:

```
pos1 = violations[0]
pos2 = violations[1]

print(filenames[pos1])
print(resized_images[pos1].shape)
plt.imshow(resized_images[pos1])
plt.title(labels[pos1])
```

```
cow1.gif
(100, 100, 4)
```

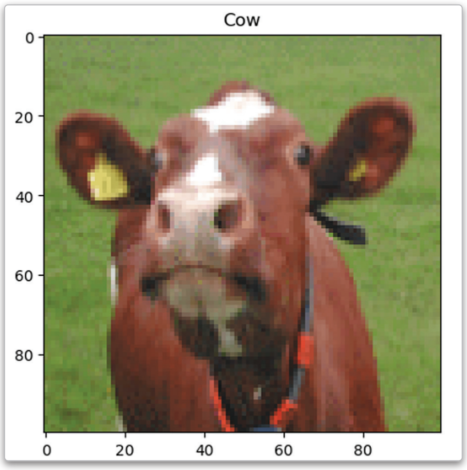


Figure 4.4: RGBA image

```
print(filenames[pos2]);
print(resized_images[pos2].shape);
plt.imshow(resized_images[pos2]);
plt.title(labels[pos2]);
```

```
tiger0000000168.jpg
(100, 100)
```

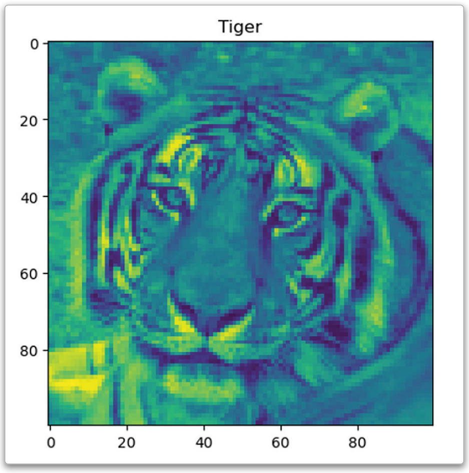


Figure 4.5: Image that shows transparency of each pixel

The first image has a shape of (100, 100, 4). The "4" reveals that the image has an "RGBA" rather than RGB format. This is an extended format and contains a fourth additional channel called the "Alpha" channel that represents the transparency of each pixel.

For example:

```
# prints the first pixel of the RGBA image
# a value of 255 reveals that the pixel is
# not transparent at all.
resized_images[pos1][0][0]
```

```
array([135, 150, 84, 255],
dtype=uint8)
```

The second image has a shape of (100, 100). The lack of the third dimension reveals that the image has a grayscale rather than RGB format. The misleading yellow/blue format shown above is due to a color map that the imshow applies by default to grayscale images. It can be switched off as follows:

```
plt.imshow(resized_images[pos2],
cmap='gray')
```

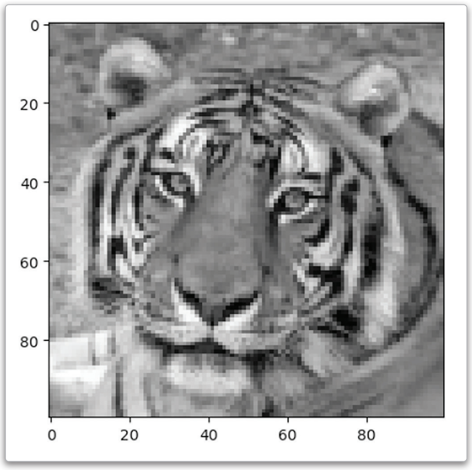


Figure 4.6: Grayscale image

Grayscale images have only one channel (rather than the 3 RGB channels). each pixel value is just a single number ranging from 0 to 255. The pixel value 0 represents black and the pixel value 255 represents white. For example:

```
resized_images[pos2][0][0]
```

```
100
```

As an additional data quality check, the following code counts the frequency of each animal label in the dataset:

```
# used to count the frequency of each element in a list.
from collections import Counter

label_cnt = Counter(labels)
label_cnt
```

```
Counter({'Bear': 101,
         'Cat': 160,
         'Chicken': 100,
         'Cow': 104,
         'Deer': 103,
         'Duck': 103,
         'Eagle': 101,
         'Elephant': 100,
         'Lion': 102,
         'Monkey': 100,
         'Nat': 8,
         'Panda': 119,
         'Pigeon': 115,
         'Rabbit': 100,
         'Sheep': 100,
         'Tiger': 114,
         'Wolf': 100})
```

The outlier in the data can be seen clearly here. The "Nat" (Nature) category has only 8 elements in comparison to the others.

The dataset contains both images of animals and nature to showcase outlier data.

The Counter reveals a very small category "Nat" with only 8 images. A quick inspection reveals that this is an outlier category with images of natural landscapes without any animal faces.

The following code removes the two RGBA and Grayscale images, as well as all the images from the "Nat" category from the resized_images, labels, and filenames lists:

```
N = len(labels)

resized_images = [resized_images[i] for i in range(N) if i not in violations
and labels[i] != "Nat"]
filenames = [filenames[i] for i in range(N) if i not in violations and
labels[i] != "Nat"]
labels = [labels[i] for i in range(N) if i not in violations and labels[i] !=
"Nat"]
```

The next step is to convert the resized_images and labels lists to numpy arrays, which is expected by many computer vision algorithms. The following code also uses the (X,y) names that are typically used to represent data and labels, respectively, in supervised learning tasks:

```
import numpy as np
X = np.array(resized_images)
y = np.array(labels)

X.shape
```

```
(1720, 100, 100, 3)
```

The shape of the final X dataset reveals that it includes 1,720 RGB images, according to the number of channels, all with the same 100x100 dimensions (10,000 pixels). Finally, the train_test_split() function from the sklearn library can be used to split the dataset 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 = 0.20,   # uses 20% of the data for testing
    shuffle = True,     # to randomly shuffle the data.
    random_state = 42,  # to ensure that data is always shuffled in the same way
)
```

Given that the animal folders were loaded one at a time, the images from each folder are packed together in the above lists. This can be misleading for many algorithms, especially in the computer vision domain. Setting shuffle=True in the code above solves this issue. In general, it is good to randomly shuffle the data before proceeding with any analysis.

Prediction without Feature Engineering

Even though the steps followed in the previous section have indeed converted the data into a numeric format, they are not in the standard one-dimensional format that is expected by many machine learning algorithms. For instance, unit 3 described how each document had to be converted to a one-dimensional numeric vector before the data could be used for training and testing machine learning models. Instead, each data point in the dataset has a 3-dimensional format:

```
X_train[0].shape
```

```
(100, 100, 3)
```

The following code can be used to "flatten" each image into a one-dimensional vector. Each image is now represented as a flat numeric vector of 100 x 100 x 3 = 30,000 values:

```
X_train_flat = np.array([img.flatten() for img in X_train])
X_test_flat = np.array([img.flatten() for img in X_test])
X_train_flat[0].shape
```

```
(30000,)
```

This flat format can now be used with any standard classification algorithm, without any additional effort to engineer additional predictive features. An example of feature engineering for image data will be explored in the following section. The following code uses the Naive Bayes (NB) classifier that was also used to classify text data in unit 3:

```
from sklearn.naive_bayes import MultinomialNB # imports the Naive Bayes Classifier

model_MNB = MultinomialNB()
model_MNB.fit(X_train_flat,y_train) # fits the model on the flat training data
```

```
MultinomialNB()
```

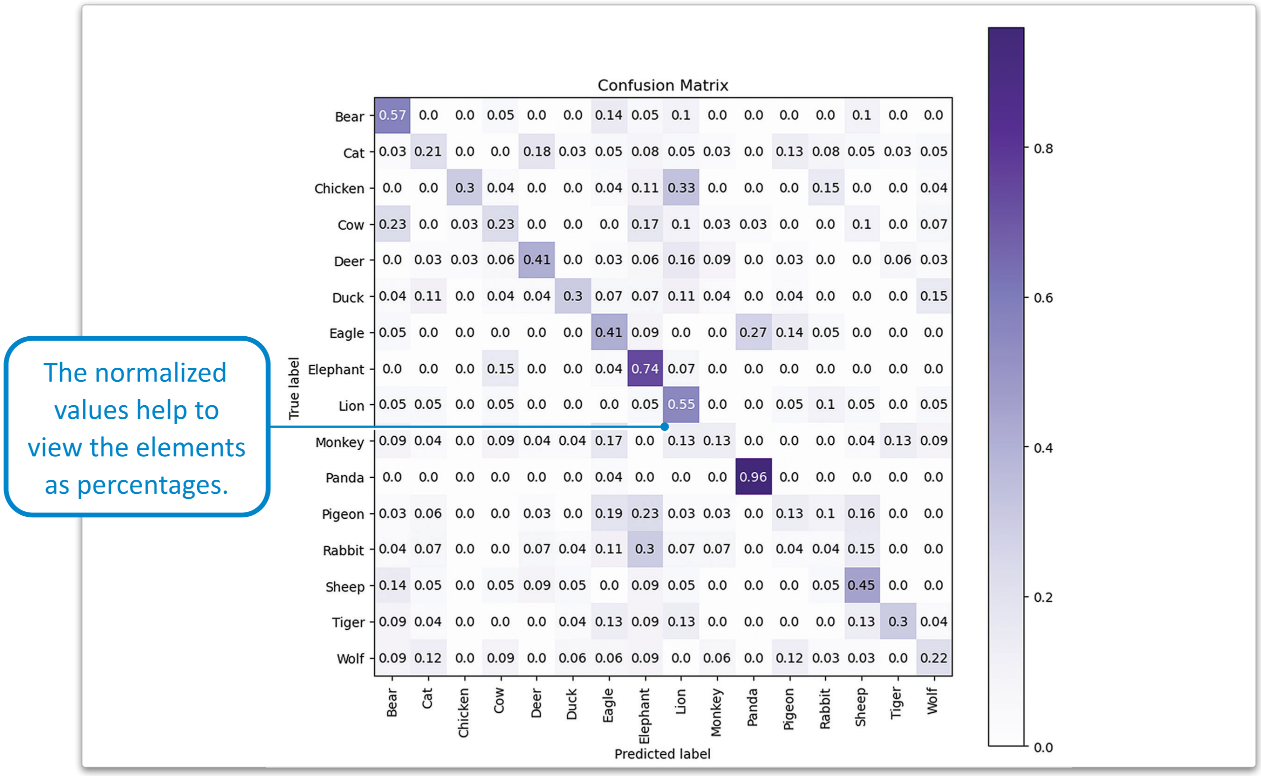
```
from sklearn.metrics import accuracy_score # used to measure the accuracy

pred = model_MNB.predict(X_test_flat) # gets the predictions for the flat test set
accuracy_score(y_test,pred)
```

```
0.36046511627906974
```

The following code prints the confusion matrix of the results, to provide additional insight:

```
%%capture
!pip install scikit-plot
import scikitplot
```

```
scikitplot.metrics.plot_confusion_matrix(y_test, # actual labels
                                    pred, # predicted labels
                                    title = "Confusion Matrix",
                                    cmap = "Purples",
                                    figsize = (10,10),
                                    x_tick_rotation = 90,
                                    normalize = True # to print percentages
                                    )
```



True label \ Predicted label	Bear	Cat	Chicken	Cow	Deer	Duck	Eagle	Elephant	Lion	Monkey	Panda	Pigeon	Rabbit	Sheep	Tiger	Wolf
Bear	0.57	0.0	0.0	0.05	0.0	0.0	0.14	0.05	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.0
Cat	0.03	0.21	0.0	0.0	0.18	0.03	0.05	0.08	0.05	0.03	0.0	0.13	0.08	0.05	0.03	0.05
Chicken	0.0	0.0	0.3	0.04	0.0	0.0	0.04	0.11	0.33	0.0	0.0	0.0	0.15	0.0	0.0	0.04
Cow	0.23	0.0	0.03	0.23	0.0	0.0	0.0	0.17	0.1	0.03	0.03	0.0	0.0	0.1	0.0	0.07
Deer	0.0	0.03	0.03	0.06	0.41	0.0	0.03	0.06	0.16	0.09	0.0	0.03	0.0	0.0	0.06	0.03
Duck	0.04	0.11	0.0	0.04	0.04	0.3	0.07	0.07	0.11	0.04	0.0	0.04	0.0	0.0	0.0	0.15
Eagle	0.05	0.0	0.0	0.0	0.0	0.0	0.41	0.09	0.0	0.0	0.27	0.14	0.05	0.0	0.0	0.0
Elephant	0.0	0.0	0.0	0.15	0.0	0.0	0.04	0.74	0.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Lion	0.05	0.05	0.0	0.05	0.0	0.0	0.0	0.05	0.55	0.0	0.0	0.05	0.1	0.05	0.0	0.05
Monkey	0.09	0.04	0.0	0.09	0.04	0.04	0.17	0.0	0.13	0.13	0.0	0.0	0.0	0.04	0.13	0.09
Panda	0.0	0.0	0.0	0.0	0.0	0.0	0.04	0.0	0.0	0.0	0.96	0.0	0.0	0.0	0.0	0.0
Pigeon	0.03	0.06	0.0	0.0	0.03	0.0	0.19	0.23	0.03	0.03	0.0	0.13	0.1	0.16	0.0	0.0
Rabbit	0.04	0.07	0.0	0.0	0.07	0.04	0.11	0.3	0.07	0.07	0.0	0.04	0.04	0.15	0.0	0.0
Sheep	0.14	0.05	0.0	0.05	0.09	0.05	0.0	0.09	0.05	0.0	0.0	0.0	0.05	0.45	0.0	0.0
Tiger	0.09	0.04	0.0	0.0	0.0	0.04	0.13	0.09	0.13	0.0	0.0	0.0	0.0	0.13	0.3	0.04
Wolf	0.09	0.12	0.0	0.09	0.0	0.06	0.06	0.09	0.0	0.06	0.0	0.12	0.03	0.03	0.0	0.22

Figure 4.7: Confusion matrix of MultinomialNB algorithm performance

The MultinomialNB algorithm achieves an accuracy around 30%. While this might seem low, it has to be considered in the context of the fact that the dataset includes 20 different labels. This means that, assuming a relatively balanced dataset where each label covers 1/20 of the data, a random classifier that randomly assigns a label to each testing point would achieve an accuracy of around 5%. Therefore, a 30% accuracy would be 6 times higher than a random guess!

Still, as shown in the following sections, this accuracy can be improved significantly. The confusion matrix also verifies that there is room for improvement. For example, the Naive Bayes model often mistakes Pigeons for Eagles or Wolves for Cats.

The easiest way to try to improve the results is to leave the data as it is and experiment with different classifiers. One model which has been shown to work well with vectorized image data is the SGDClassifier from the sklearn library. During training, the SGDClassifier adjusts the weights of the model based on the training data. The goal is to find the set of weights that minimizes a "loss" function, which measures the difference between the predicted labels and the true labels in the training data.

The following code uses the SGDClassifier to train a model on the flat dataset:

MultinomialNB

MultinomialNB is a machine learning algorithm used for classifying text or other data into different categories. It is based on the Naive Bayes algorithm, which is a simple and efficient method for solving classification problems.

SGDClassifier

The SGDClassifier is a machine learning algorithm used to classify data into different categories or groups. It is based on a technique called Stochastic Gradient Descent (SGD), which is an efficient method for optimizing and training various types of models, including classifiers.

```
from sklearn.linear_model import SGDClassifier

model_sgd = SGDClassifier()
model_sgd.fit(X_train_flat, y_train)
pred=model_sgd.predict(X_test_flat)
accuracy_score(y_test,pred)
```

```
0.46511627906976744
```

The SGDClassifier achieves signicantly higher accuracy of over 46%, despite the fact that it was trained on the exact same data as the MultinomialNB classifier. This demonstrates the potential benefits of experimenting with various classification algorithms to find the one that best fits each particular dataset. In that effort, it is also important to understand the strengths and weaknesses of each algorithm. For example, the SGDClassifier is known to perform better when the input data is scaled and the features are standardized. That is why you will be using standard scaling in your model.

Standard scaling

A preprocessing technique used in machine learning to scale the features of a dataset so that they have zero mean and unit variance.

The following code uses the StandardScaler tool from the sklearn library to scale the data:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_flat_scaled = scaler.fit_transform(X_train_flat)
X_test_flat_scaled = scaler.fit_transform(X_test_flat)

print(X_train_flat[0]) # the values of the first image pre-scaling
print(X_train_flat_scaled[0]) # the values of the first image post-scaling
```

```
[144 142 151 ...  76  75  80]
[ 0.33463473  0.27468959  0.61190285 ... -0.65170221 -0.62004162
 -0.26774175]
```

A new model can now be trained and tested using the scaled datasets:

```
model_sgd = SGDClassifier()
model_sgd.fit(X_train_flat_scaled, y_train)
pred=model_sgd.predict(X_test_flat_scaled)
accuracy_score(y_test,pred)
```

```
0.49069767441860466
```

The results indeed demonstrate an improvement after scaling. It is likely that further improvement can be achieved by experimenting with other algorithms and tuning their parameters to better fit the dataset.

Prediction with Feature Selection

While the previous section focused on training models by simply flattening the data, this section will describe how the original data can be transformed to engineer smart features that capture key properties of the image data. Specifically, the section demonstrates a popular technique called the Histogram of Oriented Gradients (HOG).

The first step towards engineering HOGs is to convert the RGB images to grayscale. This can be done with the rgb2gray() function from the sckit-image library:

Histogram of Oriented Gradients (HOG)

HOGs divide an image into small sections and analyze the distribution of intensity changes within each section, in order to identify and understand the shape of an object in the image.

```
from skimage.color import rgb2gray # used to convert a multi-color (rgb) image to grayscale
# converts the training data
X_train_gray = np.array([rgb2gray(img) for img in X_train])
# converts the testing data
X_test_gray = np.array([rgb2gray(img) for img in X_test])
```

```
plt.imshow(X_train[0]);
```

```
plt.imshow(X_train_gray[0],cmap='gray');
```

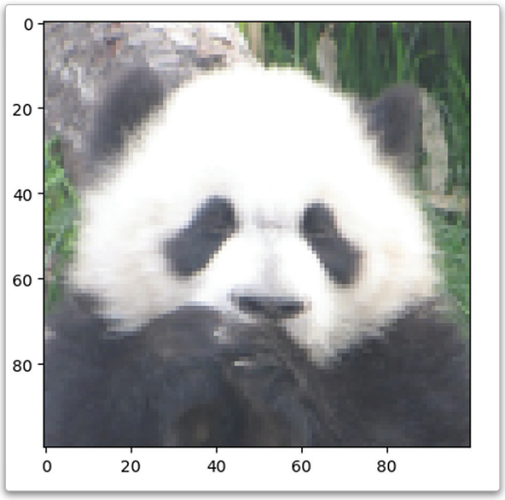


Figure 4.8: RGB image

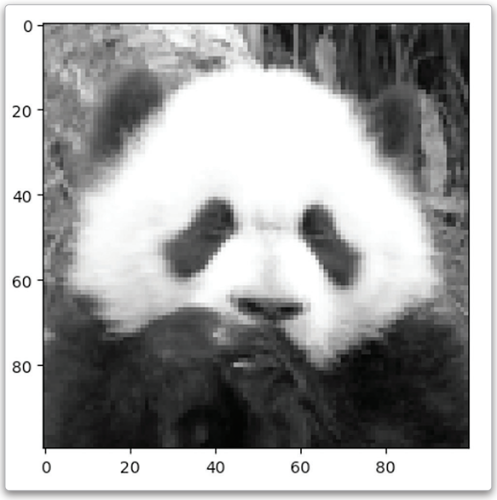


Figure 4.9: Grayscale image

The new shape of each image is now 100x100, rather than the RGB-based 100x100x3 format:

```
print(X_train_gray[0].shape)
print(X_train[0].shape)
```

```
(100, 100)
(100, 100, 3)
```

The next step is to create the HOG features for each image in the data. This can be achieved via the hog() function from the scikit-image library. The following code shows an example for the first image in the training dataset:

```
from skimage.feature import hog

hog_vector, hog_img = hog(
                  X_train_gray[0],
                  visualize = True
                  )
hog_vector.shape
```

```
(8100,)
```

The hog_vector is a one-dimensional vector with 8,100 numeric values that can now be used to represent this image. A visual representation of this vector is shown using:

```
plt.imshow(hog_img);
```

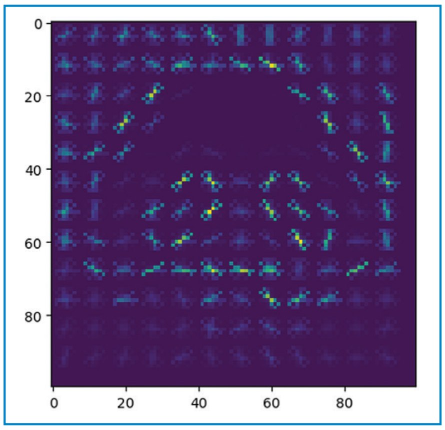


Figure 4.10: HOG of image

This new representation captures the boundaries of the key shapes in the image. It eliminates noise and focuses on the informative parts that can help a classifier to make a prediction. The following code applies this transformation to all images in both training and testing sets:

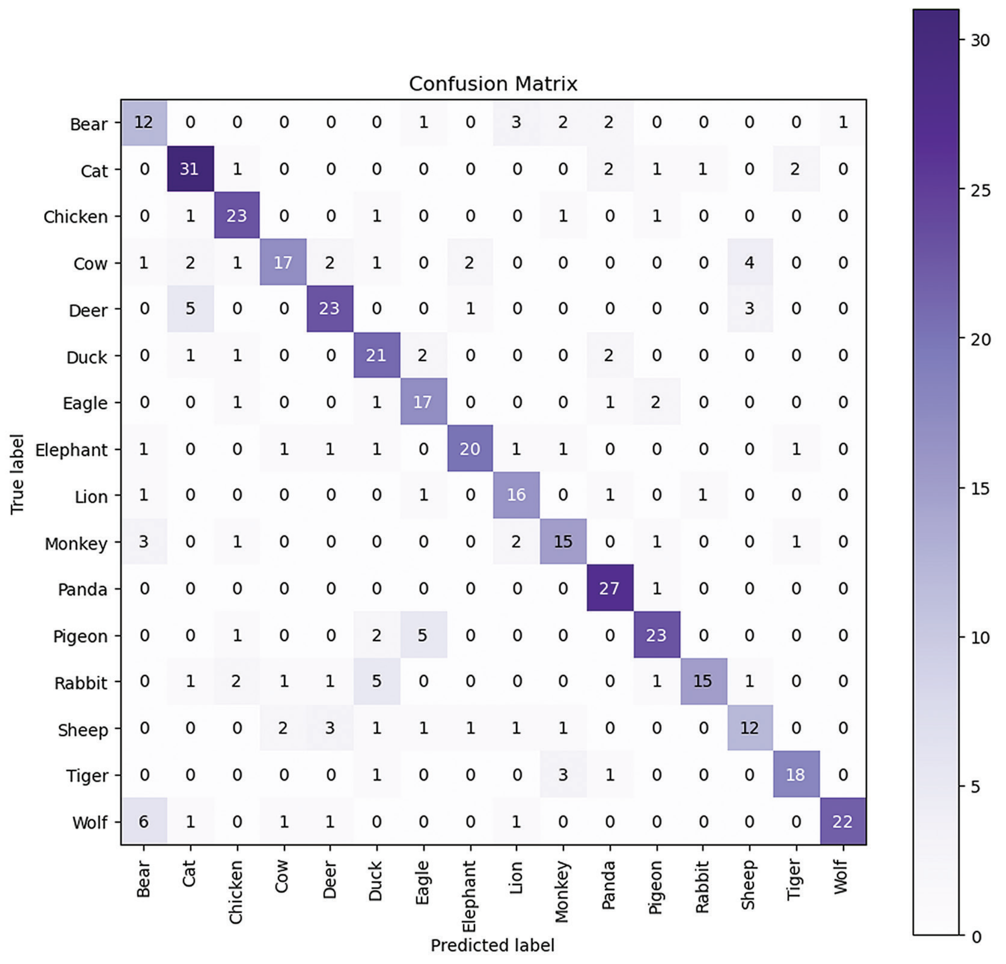
```
X_train_hog = np.array([hog(img) for img in X_train_gray])
X_test_hog = np.array([hog(img) for img in X_test_gray])
```

A new SGDClassifier can now be trained on this new representation:

```
# scales the new data
scaler = StandardScaler()
X_train_hog_scaled = scaler.fit_transform(X_train_hog)
X_test_hog_scaled = scaler.fit_transform(X_test_hog)

# trains a new model
model_sgd = SGDClassifier()
model_sgd.fit(X_train_hog_scaled, y_train)

# tests the model
pred = model_sgd.predict(X_test_hog_scaled)
accuracy_score(y_test,pred)
```

```
0.7418604651162791
```

```
scikitplot.metrics.plot_confusion_matrix(y_test, # actual labels
                                         pred, # predicted labels
                                         title = "Confusion Matrix", # title to use
                                         cmap = "Purples", # color palette to use
                                         figsize = (10,10), # figure size
                                         x_tick_rotation = 90
                                         );
```



True label \ Predicted label	Bear	Cat	Chicken	Cow	Deer	Duck	Eagle	Elephant	Lion	Monkey	Panda	Pigeon	Rabbit	Sheep	Tiger	Wolf
Bear	12	0	0	0	0	0	1	0	3	2	2	0	0	0	0	1
Cat	0	31	1	0	0	0	0	0	0	0	2	1	1	0	2	0
Chicken	0	1	23	0	0	1	0	0	0	1	0	1	0	0	0	0
Cow	1	2	1	17	2	1	0	2	0	0	0	0	0	4	0	0
Deer	0	5	0	0	23	0	0	1	0	0	0	0	0	3	0	0
Duck	0	1	1	0	0	21	2	0	0	0	2	0	0	0	0	0
Eagle	0	0	1	0	0	1	17	0	0	0	1	2	0	0	0	0
Elephant	1	0	0	1	1	1	0	20	1	1	0	0	0	0	1	0
Lion	1	0	0	0	0	0	1	0	16	0	1	0	1	0	0	0
Monkey	3	0	1	0	0	0	0	0	2	15	0	1	0	0	1	0
Panda	0	0	0	0	0	0	0	0	0	0	27	1	0	0	0	0
Pigeon	0	0	1	0	0	2	5	0	0	0	0	23	0	0	0	0
Rabbit	0	1	2	1	1	5	0	0	0	0	0	1	15	1	0	0
Sheep	0	0	0	2	3	1	1	1	1	1	0	0	0	12	0	0
Tiger	0	0	0	0	0	1	0	0	0	3	1	0	0	0	18	0
Wolf	6	1	0	1	1	0	0	0	1	0	0	0	0	0	0	22

Figure 4.11: Confusion matrix of SGDClassifier algorithm performance

The new results reveal a massive improvement in accuracy, which has now jumped to over 70% and has far surpassed the accuracy achieved by the same classifier on the flat data without any feature engineering. The improvement is also apparent in the updated confusion matrix, which now includes far less false positives (mistakes). This demonstrates the value of using computer vision techniques to engineer intelligent features that capture the various visual properties of the data.

Prediction Using Neural Networks

This section demonstrates how neural networks can be used to design classifiers that are customized for image data and can often surpass even highly effective techniques, such as the HOG process that was described in the previous section. The popular Tensorflow and Keras libraries are used for this purpose. TensorFlow is a low-level library that provides a wide range of tools for machine learning and artificial intelligence. It allows users to define and manipulate numerical computations involving tensors, which are multi-dimensional arrays of data.

Keras, on the other hand, is a higher-level library that provides a simpler interface for building and training models. It is built on top of TensorFlow (or other backends) and provides a set of pre-defined layers and models that can be easily assembled to build a deep learning model. Keras is designed to be user-friendly and easy to use, making it a popular choice for practitioners. Activation functions are mathematical functions applied to the output of each neuron in a neural network that have the advantage of adding non-linear properties to the model and allowing the network to learn complex patterns in the data. The choice of activation function is important and can impact the network's performance. Neurons receive input, process it with weights and biases, and produce an output based on an activation function, as shown in figure 4.12. Neural networks are constructed by connecting many neurons together in layers and are trained to adjust the weights and biases and improve their performance over time.

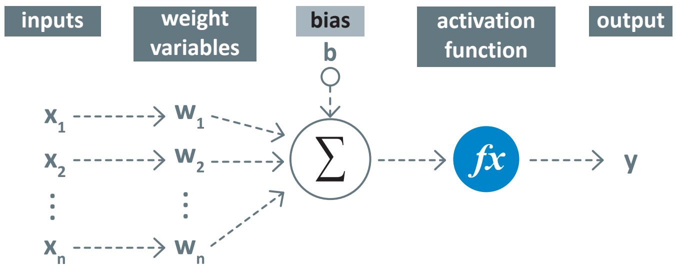


Figure 4.12: Activation function

The following code installs the libraries **tensorflow** and **keras**:

```
%%capture
!pip install tensorflow
!pip install keras
```

In the previous unit you were introduced to artificial neurons and neural network architectures. Specifically, the Word2Vec model, which used a hidden layer and an output layer to predict the context words of a given word in a sentence. Next, Keras is used to create a similar neural architecture for images. First, the labels in y_train are converted to an integer format, as required by Keras:

```
# gets the set of all distinct labels
classes=list(set(y_train))
print(classes)
print()

# replaces each label with an integer (its index in the classes lists) for both the training and testing data
y_train_num = np.array([classes.index(label) for label in y_train])
y_test_num = np.array([classes.index(label) for label in y_test])
print()

# example:
print(y_train[:5]) # first 5 labels
print(y_train_num[:5]) # first 5 labels in integer format
```

The Sequential tool from the Keras library can now be used to build a neural network as a sequence of layers.

```
from keras.models import Sequential # used to build neural networks as sequences of layers
# every neuron in a dense layer is connected to every other neuron in the previous layer.
from keras.layers import Dense

# builds a sequential stack of layers
model = Sequential()
# adds a dense hidden layer with 200 neurons, and the ReLU activation function.
model.add(Dense(200,input_shape = (X_train_hog.shape[1],), activation='relu'))
# adds a dense output layer and the softmax activation function.
model.add(Dense(len(classes), activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 200)               1620200

 dense_1 (Dense)             (None, 16)                3216

=================================================================
Total params: 1,623,416
Trainable params: 1,623,416
Non-trainable params: 0
_________________________________________________________________
```

The number of neurons in the hidden layer is a design choice. The number of neurons in the output layer is dictated by the number of classes.

The model summary reveals the total number of parameters that the model has to learn by fitting on the training data. Since the input has 8,100 entries, which are the the dimensions of the HOG images X_train_hog and the hidden layer has 200 neurons and is a dense layer that is fully connected to the input, this creates a total of 8,100 x 200 = 1,620,000 weighted connections whose weights (parameters) have to be learned. An additional 200 "bias" parameters are added, one for each neuron in the hidden layer. A bias parameter is a value that is added to the input of each neuron in a neural network. It is used to shift the activation function of the neuron to the negative or positive side, allowing the network to model more complex relationships between the input data and the output labels.

Given that the output layer has 16 neurons that are fully connected to the 200 neurons of the hidden layer, this adds an additional 16 x 200 = 3,216 weighted connections. An additional 16 bias parameters are added, one for each neuron in the output layer. The following line is used to "compile" the model:

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

The Keras smart model preparation method known as model.compile() is used to define the basic characteristics of a smart model and prepare it for training, verification, and prediction. It takes three main arguments as illustrated in Table 4.2

Table 4.2: The arguments of the "compile" method

loss	This is the loss function that is used to evaluate the error in the model during training. It measures how well the model's predictions match the true labels for a given set of input data. The goal of training is to minimize the loss function, which typically involves adjusting the model's weights and biases. In this case, the loss function is 'sparse_categorical_crossentropy', which is a loss function suitable for multi-class classification tasks where the labels are integers (as in y_train_num).
metrics	This is a list of metrics that is used to evaluate the model during training and testing. These metrics are computed using the output of the model and the true labels, and they can be used to monitor the performance of the model and identify areas where it can be improved. "Accuracy" is a common metric for classification tasks that measures the fraction of correct predictions made by the model.
optimizer	This is the optimization algorithm that is used to adjust the model's weights and biases during training. The optimizer uses the loss function and the metrics to guide the training process, and it adjusts the model's parameters in an effort to minimize the loss and maximize the performance of the model. In this case, the optimizer is 'adam', which is a popular algorithm for training neural networks.

Finally, the fit() method is used to train the model on the available data:

```
model.fit(X_train_hog, # training data
          y_train_num, # labels in integer format
          batch_size = 80, # number of samples processed per batch
          epochs = 40, # number of iterations over the whole dataset
          )
```

```
Epoch 1/40
17/17 [==============================] - 1s 16ms/step - loss: 2.2260 - accuracy: 0.3333
Epoch 2/40
17/17 [==============================] - 0s 15ms/step - loss: 1.1182 - accuracy: 0.7256
Epoch 3/40
17/17 [==============================] - 0s 15ms/step - loss: 0.7198 - accuracy: 0.8155
Epoch 4/40
17/17 [==============================] - 0s 15ms/step - loss: 0.4978 - accuracy: 0.9031
Epoch 5/40
17/17 [==============================] - 0s 16ms/step - loss: 0.3676 - accuracy: 0.9388
...
Epoch 36/40
17/17 [==============================] - 0s 15ms/step - loss: 0.0085 - accuracy: 1.0000
Epoch 37/40
17/17 [==============================] - 0s 21ms/step - loss: 0.0080 - accuracy: 1.0000
Epoch 38/40
17/17 [==============================] - 0s 15ms/step - loss: 0.0076 - accuracy: 1.0000
Epoch 39/40
17/17 [==============================] - 0s 15ms/step - loss: 0.0073 - accuracy: 1.0000
Epoch 40/40
17/17 [==============================] - 0s 15ms/step - loss: 0.0071 - accuracy: 1.0000
```

The fit() method is used to train a model on a given set of input data and labels. It takes four main arguments, as illustrated in Table 4.3.

Table 4.3: The arguments of the "fit" method

X_train_hog	This is the input data that is used to train the model. It consists of the HOG-transformed data that was also used to train the latest version of the SGDClassifier in the previous section.
y_train_num	This includes the label for each image in integer format.
batch_size	This is the number of samples that is processed in each batch during training. The model updates its weights and biases after each batch, and the batch size can affect the speed and stability of the training process. Larger batch sizes can lead to faster training, but they can also be more computationally expensive and may result in less stable gradients.
epochs	This is the number of times the model iterates over the entire dataset during training. An epoch consists of one pass through the entire dataset, and the model updates its weights and biases after each epoch. The number of epochs can affect the model's ability to learn and generalize to new data. It is an important hyperparameter that should be chosen carefully. In this case, the model is trained for 40 epochs.

The trained model can now be used to predict the labels of the images in the testing set:

```
pred = model.predict(X_test_hog)
pred[0]  # prints the predictions for the first image
```

```
14/14 [==============================] - 0s 2ms/step

array([4.79123509e-03, 9.79321003e-01, 8.39506648e-03, 1.97884417e-03,
       7.83501855e-06, 3.50346789e-04, 3.45465224e-07, 1.19854585e-05,
       4.41945267e-05, 4.11721296e-04, 1.27362555e-05, 9.83431892e-06,
       1.97038025e-04, 2.34744814e-03, 5.49758552e-04, 1.57057808e-03],
      dtype=float32)
```

While the predict() function from the sklearn library returns the most likely label as predicted by the classifier, the Keras predict() function returns the probability of all candidate labels. The np.argmax() function can then be used to return the index of the highest probability:

```
# index of the class with the highest predicted probability.
print(np.argmax(pred[0]))
# name of this class
print(classes[np.argmax(pred[0])])
# uses axis=1 to find the index of the max value per row
accuracy_score(y_test_num,np.argmax(pred, axis=1))
```

```
1
Duck
0.7529021558872305
```

This simple neural network achieves an accuracy around 75%, similar to the one reported by the SGDClassifier. However, the advantage of neural architectures comes from their versatility, which allows you to experiment with different architectures to find the one that best fits your dataset.

This accuracy was achieved with a simple and shallow architecture that included just one hidden layer with 200 neurons. Adding additional layers would make the network deeper, while adding more neurons per layer would make it wider. The choice of the number of layers and number of neurons per layer are important components of neural network design that have a considerable impact on their performance. However, they are not the only way to improve performance and, in some cases, using a different type of neural network architecture may be more effective.

Prediction Using Convolutional Neural Networks

One such type of architecture that is particularly well-suited for image classification is the Convolutional Neural Network (CNN). As the CNN processes the input data, it continually adjusts the parameters of convolved filters to detect patterns based on the data it sees, in order to better detect the desired features. The output of each layer is then passed on to the next layer, where more complex features are detected, until the final output is produced.

Despite the benefits of complex neural networks like CNNs, it is important to note that:

- The power of convolutional neural networks (CNNs) is their ability to automatically extract relevant features from images, without the need for manual feature engineering.
- More complex neural architectures have more parameters that have to be learned from the data during training. This typically requires a larger training dataset, which may not be available in some cases. In such cases, creating an overly complex architecture is unlikely to be effective.
- Even though neural networks have indeed achieved impressive results in image processing and other tasks, they are not guaranteed to always deliver the best performance across problems and datasets.
- Even if a neural network architecture is the best possible solution for a specific task, it may take a lot of time, effort, and computational resources to experiment with different options until this architecture is found. It is therefore best practice to start with simpler (but still effective) models, such as the SGDClassifier and many others from libraries such as sklearn. Once you have built a better prediction for the dataset and have reached the point where such models can no longer be improved, then experimenting with neural architectures is an excellent next step.

Convolutional Neural Network (CNN)

CNNs are deep neural networks that automatically learn a hierarchy of features from raw data, like images, by applying a series of convolved filters to the input data, which are designed to detect specific patterns or features.

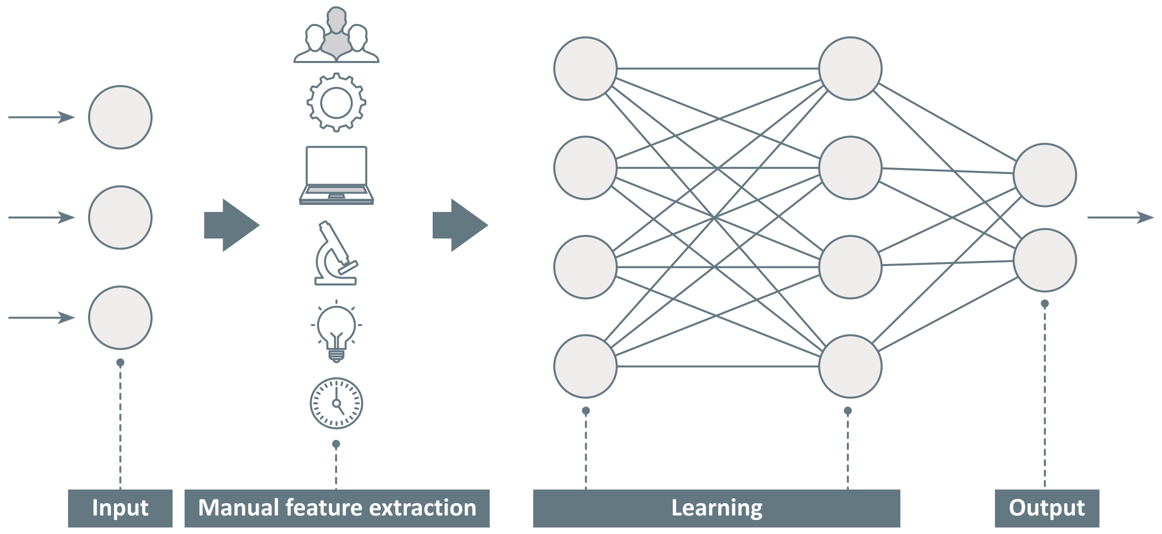


Figure 4.13: Neural network with manual feature engineering

INFORMATION

One of the key advantages of CNNs is that they are very good at learning from large amounts of data, and can often achieve high levels of accuracy on tasks such as image classification without the need for manual feature engineering, such as the HOG process.

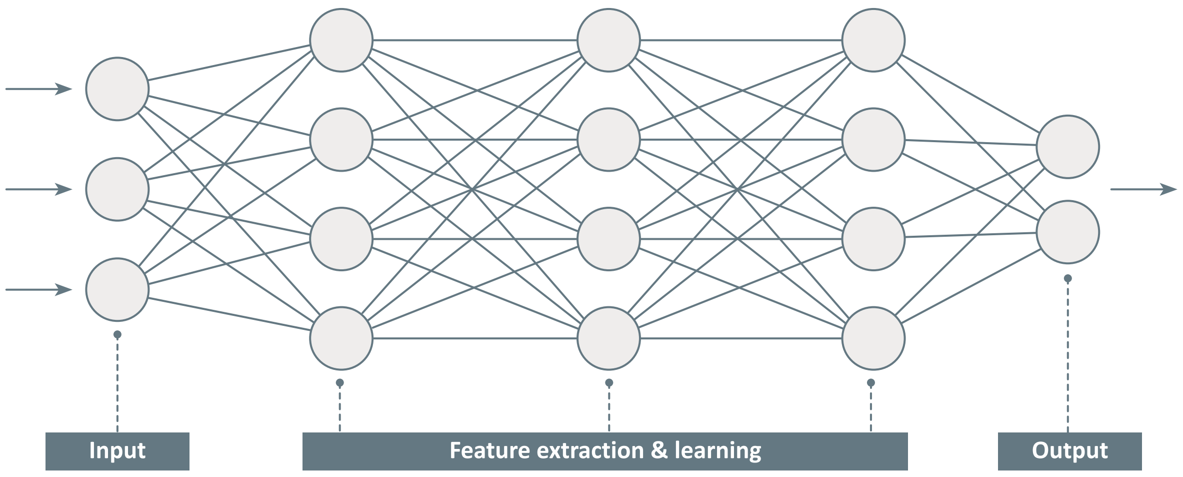


Figure 4.14: Convolutional neural network without manual feature engineering

Transfer Learning

Transfer learning is a process of reusing a pre-trained neural network to solve a new task. In the context of convolutional neural networks (CNN), transfer learning involves taking a pre-trained model, which was trained on a large dataset, and adapting it to a new dataset or task. Instead of starting from scratch, transfer learning allows the use of pre-trained models, which have already learned important features, such as edges, shapes, and textures from the training dataset.

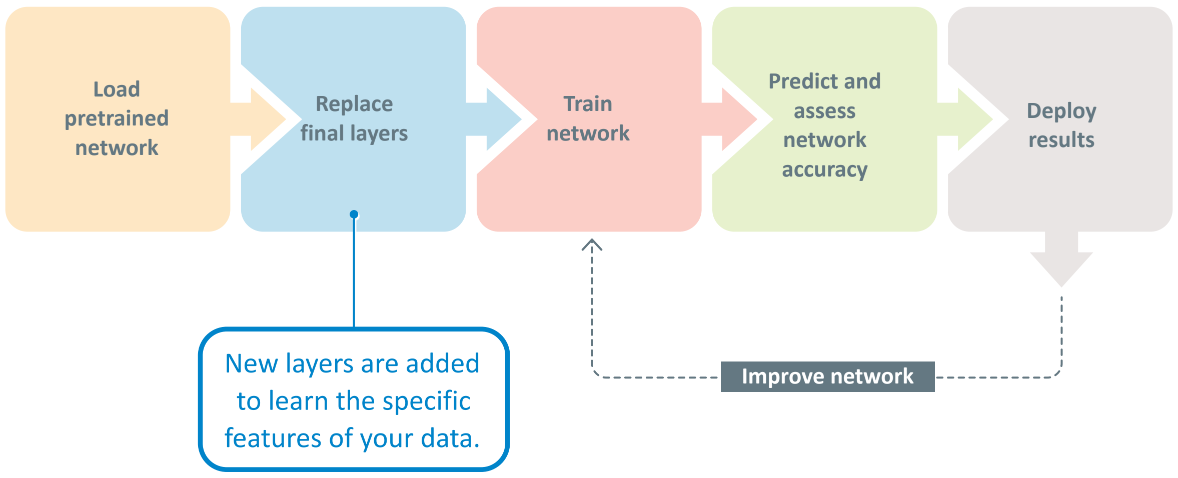

Load pretrained network
Replace final layers
Train network
Predict and assess network accuracy
Deploy results
New layers are added to learn the specific features of your data.
Improve network

Figure 4.15: Reuse of pretrained network

Exercises

1 What are the challenges of visual data classification?

2 You are given two numpy arrays X_train and y_train. Each row in X_train has a shape of (100, 100, 3) and represents a 100 × 100 RGB image. The n_th row in y_train represents the label of the n_th image in X_train. Complete the following code so that it flattens X_train and then trains a MultinomialNB model on this dataset.

```
from sklearn.naive_bayes import MultinomialNB # imports the Naive Bayes Classifier from sklearn

X_train_flat = np.array(_______________)

model_MNB = MultinomialNB() # new Naive Bayes model

model_MNB.fit(___________________, _________________) # fits model on the flat training data
```

3 Descibe briefly how CNNs work and one of their key advantages.

4 You are given two numpy arrays X_train and y_train. Each row in X_train has a shape of (100, 100, 3) and represents a 100 × 100 RGB image. The n_th row in y_train represents the label of the n_th image in X_train. Complete the following code so that it applies the HOG transformation on this dataset and then uses the transformed data to train a MultinomialNB model:

```
from skimage.color import _____________  # used to convert a multi-color (rgb) image to grayscale
from sklearn._____________ import StandardScaler # used to scale the data
from sklearn.naive_bayes import MultinomialNB # imports the Naive Bayes Classifier from sklearn
X_train_gray = np.array([________________(img) for img in X_train]) # converts training data
X_train_hog = ___________________
scaler = StandardScaler()
X_train_hog_scaled = ___________________.fit_transform(X_train_hog)
model_MNB = MultinomialNB()
model_MNB.fit(X_train_flat_scaled, __________________)
```

5 Name some challenges of CNNs.

Lesson 2

Unsupervised Learning for Image Analysis

Link to digital lesson

www.ien.edu.sa

Understanding Image Content

In the context of computer vision, unsupervised learning has been used for a variety of tasks, such as image segmentation, video segmentation, and anomaly detection. Another key application of unsupervised learning is image search, which involves searching a large database of images to find those that are similar to a given query image.

The first step towards building a search engine for image data is defining a similarity function that can evaluate the similarity between two images based on their visual properties, such as their border, texture, or shape. Once the user submits a new image as a query, the search engine goes over all the images in the available database, finds those with the highest similarity score, and returns them to the user.

An alternative approach is to use the similarity function to separate the images into clusters, so that each cluster consists of images that are visually similar to each other. Each cluster is then represented by a centroid: an image that sits at the center of the cluster and has the smallest overall distance (i.e. difference) from the other cluster members. Once the user submits a new image as a query, the search engine will go over all the clusters and select the one whose centroid is the most similar to the query image. The members of the selected cluster then returned to the user. Figure 4.16 shows an example of this approach:

Anomaly Detection

Anomaly detection is a process used to identify abnormal or unexpected patterns, events, or data points within a dataset. Its aim is to uncover unusual cases that stand out from the norm and may warrant further investigation.

Image Segmentation

Image segmentation is a process of dividing an image into multiple segments or regions that share common visual properties. Its aim is to partition an image into meaningful and coherent parts that can be used for further analysis.

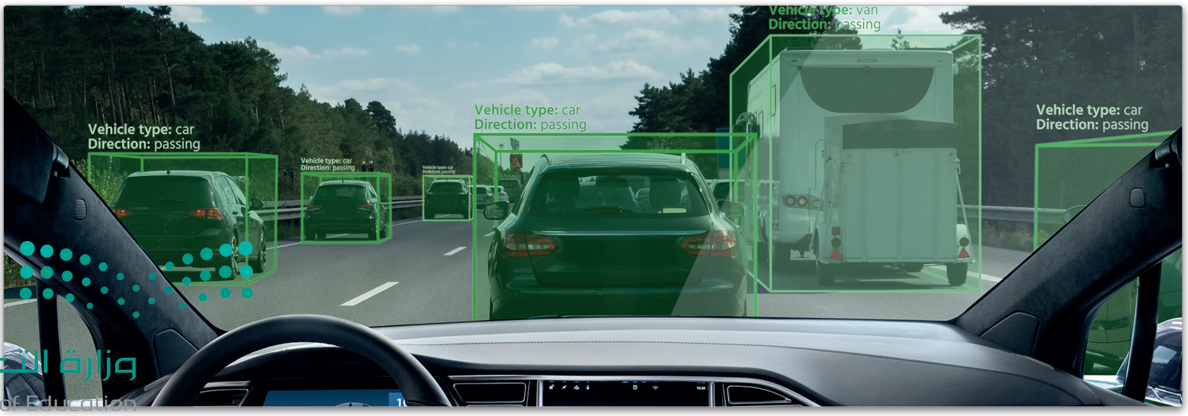


Figure 4.16: Autonomous vehicle vision with image segmentation

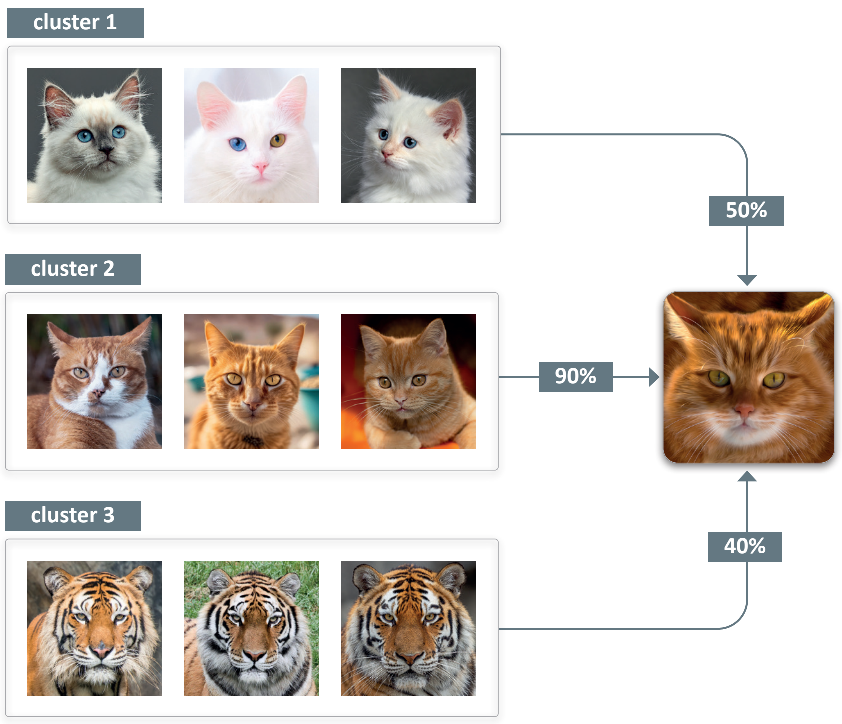


Figure 4.17: Clusters of image recognition analysis

In this example shown in figure 4.17, the query image has a similarity of 40%, 50%, and 90% with the centroids of the three image clusters, respectively. Similarity is assumed to be a percentage between 0% and 100%. Cluster 2 has the highest score, as it includes cats of the same breed and color as the query image. The scores of clusters 1 and 3 are close to each other (40% and 50%), as the two clusters are similar to the query in different ways. Cluster 1 includes cats with a significantly different color pattern. On the other hand, even though cluster 3 represents a different type of animal (tiger), the color pattern is similar to that of the query image.

The process of clustering visual data is similar to that of clustering numeric or textual data. However, the unique nature of visual data requires specialized methods for evaluating visual similarity. Even though early methods relied on hand-crafted features, recent advances in deep learning have led to the development of powerful models that can automatically learn sophisticated features from unlabeled visual data.

This lesson uses an image-clustering task to demonstrate how using more sophisticated features can lead to significantly better results. Specifically, the lesson will cover three different approaches:

- Flattening and clustering the original data, without any feature engineering.
- Transforming the data using the HOG feature descriptor (introduced in the previous lesson) and then clustering the transformed data.
- Using a neural network model to cluster the original data without any feature engineering.

The LHI-Animal-Faces dataset that was used in the previous lesson will also be used to evaluate the various image-clustering techniques. This dataset was originally designed for classification tasks and therefore includes the true label (the actual animal type) for each image. In this lesson, these labels will only be used for validation and will not be used to actually cluster the images. An effective clustering approach should be able to group images with the same label in the same cluster and separate images with different labels into different clusters.

Loading and Preprocessing Images

The following code imports the libraries that will be used to load and preprocess the images:

```
%%capture
import matplotlib.pyplot as plt
from os import listdiry

!pip install scikit-image
from skimage.io import imread
from skimage.transform import resize
from skimage import img_as_ubyte

# a palette of 10 colors that will be used to visualize the clusters.
color_palette = ['blue','green','red','yellow','gray','purple','orange',
'pink','black','brown']
```

The following function reads the images of the LHI-Animal-Faces dataset from their input_folder and resizes each of them to the same width and height dimensions. It extends the resize_images() from the previous lesson by allowing the user to specify a list of animal classes that should be considered. It also uses a single line of Python code to read, resize, and store each image:

```
def resize_images_v2(input_folder:str,
                  width:int,
                  height:int,
                  labels_to_keep:list
                 ):
    labels = []          # a list with the label for each image
    resized_images = [] # a list of resized images in np array format
    filenames = []       # a list of the original image file names

    for subfolder in listdir(input_folder):

        print(subfolder)
        path = input_folder + '/' + subfolder

        for file in listdir(path):

            label=subfolder[:-4] # uses the subfolder name without the "Head" suffix
            if label not in labels_to_keep: continue
            labels.append(label) # appends the label
            #loads, resizes, preprocesses, and stores the image.
            resized_images.append(img_as_ubyte(resize(imread(path+'/'+file),
(width, height))))
            filenames.append(file)

    return resized_images,labels,filenames
```

Unstructured data is diverse and can require a lot of time and computational resources. This is especially true when they are processed via complex deep learning techniques, as will be done later in this lesson. Therefore, in order to reduce computational time, the resize_images_v2() is applied to a subset of images from animal classes:

```
resized_images,labels,filenames=resize_images_v2(
            "AnimalFace/Image",
            width = 224,
            height = 224,
            labels_to_keep=['Lion', 'Chicken', 'Duck', 'Rabbit', 'Deer',
'Cat', 'Wolf', 'Bear', 'Pigeon', 'Eagle']
            )
```

```
BearHead                                MonkeyHead
CatHead                                 Natural
ChickenHead                             PandaHead
CowHead                                 PigeonHead
DeerHead                                RabbitHead
DuckHead                                SheepHead
EagleHead                               TigerHead
ElephantHead                            WolfHead
LionHead
```

These 10 are the labels that are going to be used

You can easily change the "labels_to_keep" parameter to focus on particular classes. You will also notice that the width and height of the images are now set to 224 × 224, rather than the 100 × 100 shape that was used in the previous lesson. This is done because one of the deep-learning clustering methods that is presented in this lesson requires the images to have these dimensions. The 224 × 224 shape is therefore adopted in order to ensure that all methods are given access to the same input.

As also mentioned in the previous lesson, the original lists (resized_images, labels, filenames) include the images from each class packed together. For instance, all the "Lion" images appear together at the beginning of the 'resized' list. This can be misleading for many algorithms, especially in the computer vision domain. While this can be addressed by randomly shuffling each of the three lists, it is important to ensure that the same random order is used for all three of them. Otherwise, it is impossible to find the correct label or filename for a specific image.

In the previous lesson, shuffling was taken care of by the train_test_split() function. However, given that this function is not applicable for clustering tasks, the following code is used for shuffling:

```
import random

#connects the three lists together, so that they are shuffled in the same order
connected = list(zip(resized_images,labels,filenames))
random.shuffle(connected)
# disconnects the three lists
resized_images,labels,filenames= zip(*connected)
```

The next step is to convert the 'resized_images' and 'labels' lists to numpy arrays. Similarly to the previous lesson, the standard (X,y) variable names are used to represent data and labels:

```
import numpy as np # used for numeric computations
X = np.array(resized_images)
y = np.array(labels)

X.shape
```

```
(1085, 224, 224, 3)
```

The shape of the data verifies that it includes 1,085 images, each with dimensions of 224 × 224 and 3 RGB channels.

Clustering without Feature Engineering

The first clustering attempt will focus on simply flattening the images to convert each of them to a one-dimensional vector with 224 × 224 × 3 = 150,528 numbers.

Similar to the classification algorithms that were explored in the previous lesson, most clustering algorithms also require this type of vectorized format.

```
X_flat = np.array([img.flatten() for img in X])
X_flat[0].shape
```

```
(150528,)
```

```
X_flat[0] # prints the first flat image
```

```
array([107, 146, 102, ...,  91,  86, 108], dtype=uint8)
```

Each numeric value in this flat format is an RGB value between 0 and 255. As also seen in the previous lesson, standard scaling and normalization can sometimes improve the results of some machine learning algorithms.

The following code can be used to normalize the values and bring them between 0 and 1.

```
X_norm = X_flat / 255
X_norm[0]
```

```
array([0.41960784, 0.57254902, 0.4       , ..., 0.35686275, 0.3372549 ,
       0.42352941])
```

The data can now be visualized using the familiar TSNEVisualizer tool from the yellowbrick library. This tool was also used in unit 3 lesson 2 to visualize the clusters in text data.

```
%%capture
!pip install yellowbrick
from yellowbrick.text import TSNEVisualizer
```

```
tsne = TSNEVisualizer(colors = color_palette) # initializes the tool
tsne.fit(X_norm, y) # uses TSNE to reduce the data to 2 dimensions
tsne.show();
```

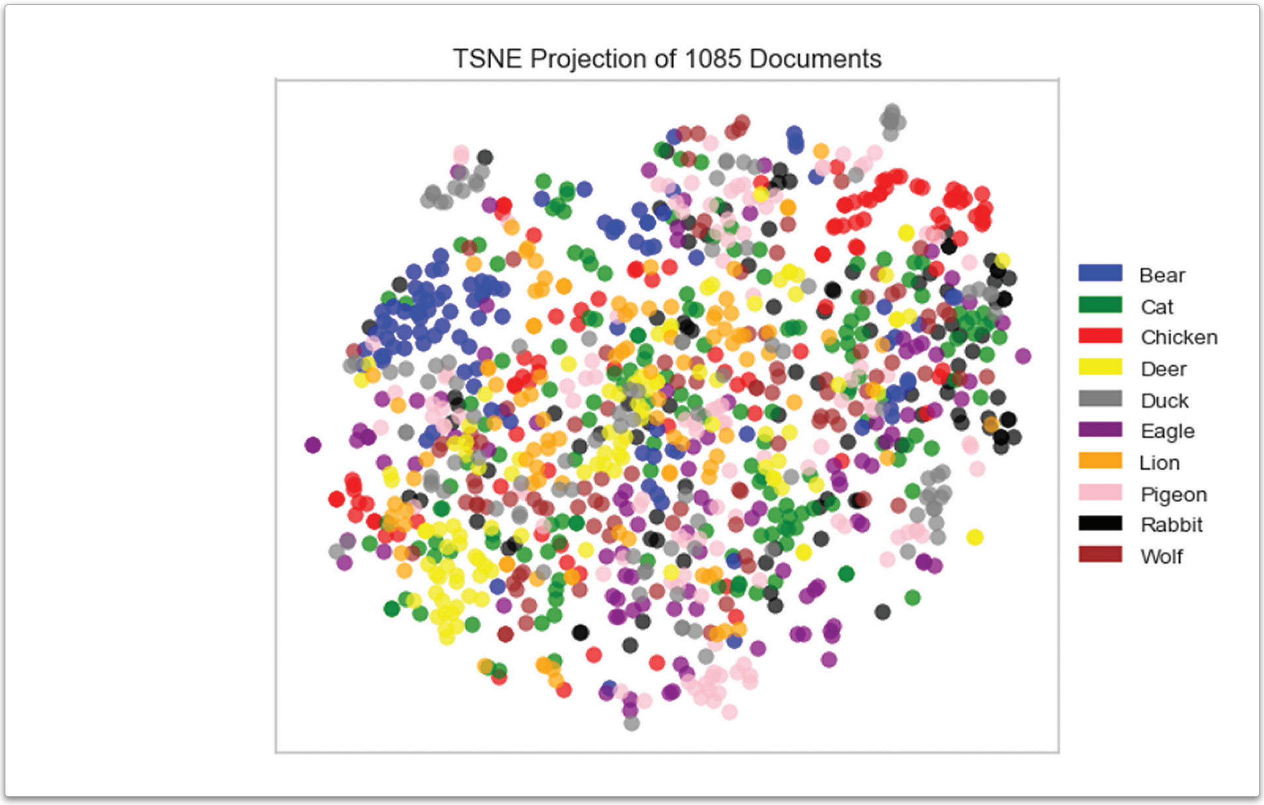


Figure 4.18: Clusters visualization

This preliminary visualization is not promising. The various animal classes seem to be scrambled together, without clear separation and no obvious clusters. This indicates that simply flattening the original image data is unlikely to lead to high quality results.

Next, the same agglomerative clustering algorithm that was used in unit 3 lesson 2 is also used to cluster the data in X_norm. The following code imports the set of required tools and visualizes the dendrogram of the dataset:

```
from sklearn.cluster import AgglomerativeClustering # used for agglomerative clustering
import scipy.cluster.hierarchy as hierarchy

hierarchy.set_link_color_palette(color_palette) # sets the color palette
plt.figure()

# iteratively merges points and clusters until all points belong to a single cluster
linkage_flat = hierarchy.linkage(X_norm, method = 'ward')
hierarchy.dendrogram(linkage_flat)
plt.show()
```

'ward' is a linkage method used in hierarchical agglomerative clustering.

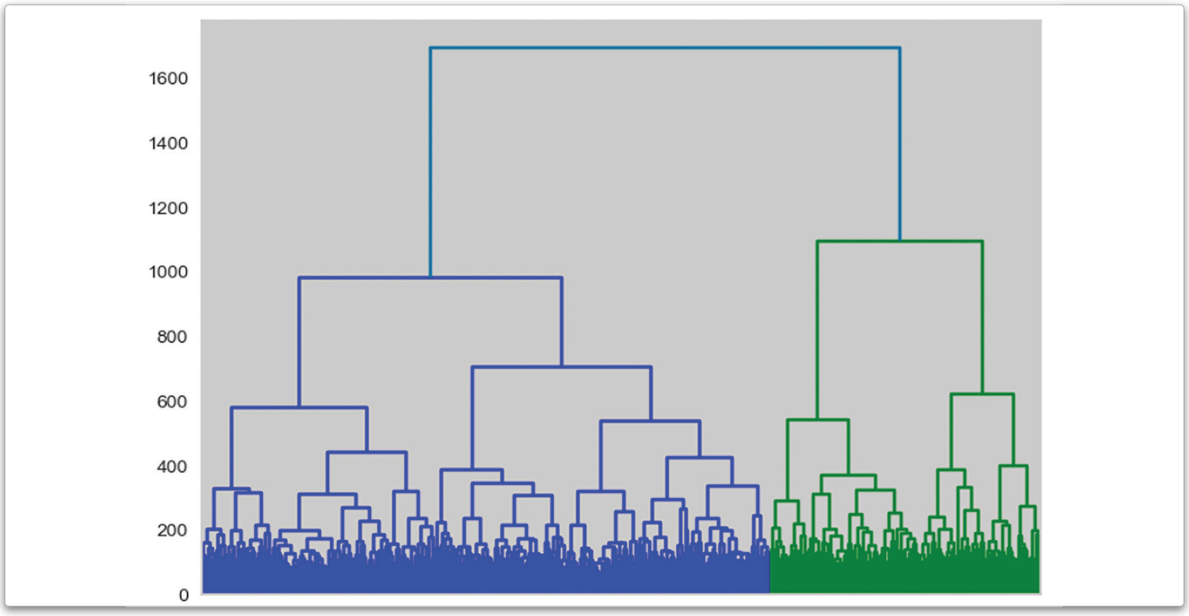


Figure 4.19: Dendrogram categorizing data into two clusters

The dendrogram reveals two large clusters that can be further broken down into smaller ones. The following code uses the AgglomerativeClustering tool to create 10 clusters, which is the actual number of clusters in the data:

```
AC = AgglomerativeClustering(linkage = 'ward',n_clusters = 10)
AC.fit(X_norm) # applies the tool to the data

pred = AC.labels_ # gets the cluster labels

pred
```

```
array([9, 6, 3, ..., 4, 4, 3], dtype=int64)
```

Finally, the homogeneity, completeness, and adjusted Rand metrics (all introduced in unit 3 lesson 2) are used to evaluate the quality of the produced clusters:

```
from sklearn.metrics import homogeneity_score, adjusted_rand_score,
completeness_score

print('\nHomogeneity score:', homogeneity_score(y, pred))
print('\nAdjusted Rand score:', adjusted_rand_score(y, pred))
print('\nCompleteness score:', completeness_score(y, pred))
```

```
Homogeneity score: 0.09868725008128477

Adjusted Rand score: 0.038254515908926826

Completeness score: 0.101897123096584
```

As described in detail in unit 3 lesson 2, the homogeneity and completeness scores take values between 0 and 1. The first is maximized when all the points of each cluster have the same ground truth label. The second one is maximized when all the data points with the same ground truth label also belong to the same cluster. Finally, the adjusted Rand score takes values between –0.5 and 1.0 and is maximized when all the data points with the same label are in the same cluster and all points with different labels are in different clusters. As expected following the visualization of the data, the algorithm fails to find high-quality clusters that match the actual animal classes. The values for all three metrics are very low. This demonstrates that, even though simply flattening the data was sufficient to get reasonable results for image classification, image clustering is a significantly harder problem.

Clustering with Feature Selection

The previous lesson demonstrated how the HOG transformation can be used to convert image data into a more informative format that led to significantly higher performance for image classification. Next, the same transformation is applied to test whether it can also improve the results of image clustering tasks.

```
from skimage.color import rgb2gray
from skimage.feature import hog
# converts the list of resized images to an array of grayscale images
X_gray = np.array([rgb2gray(img) for img in resized_images])
# computes the HOG features for each grayscale image in the array
X_hog = np.array([hog(img) for img in X_gray])
X_hog.shape
```

```
(1085, 54756)
```

The shape of the transformed data reveals that each image is now represented as a vector of 54,756 numeric values.

The following code uses the TSNEVisualizer tool to visualize this new format:

```
tsne = TSNEVisualizer(colors = color_palette)
tsne.fit(X_hog, y)
tsne.show();
```

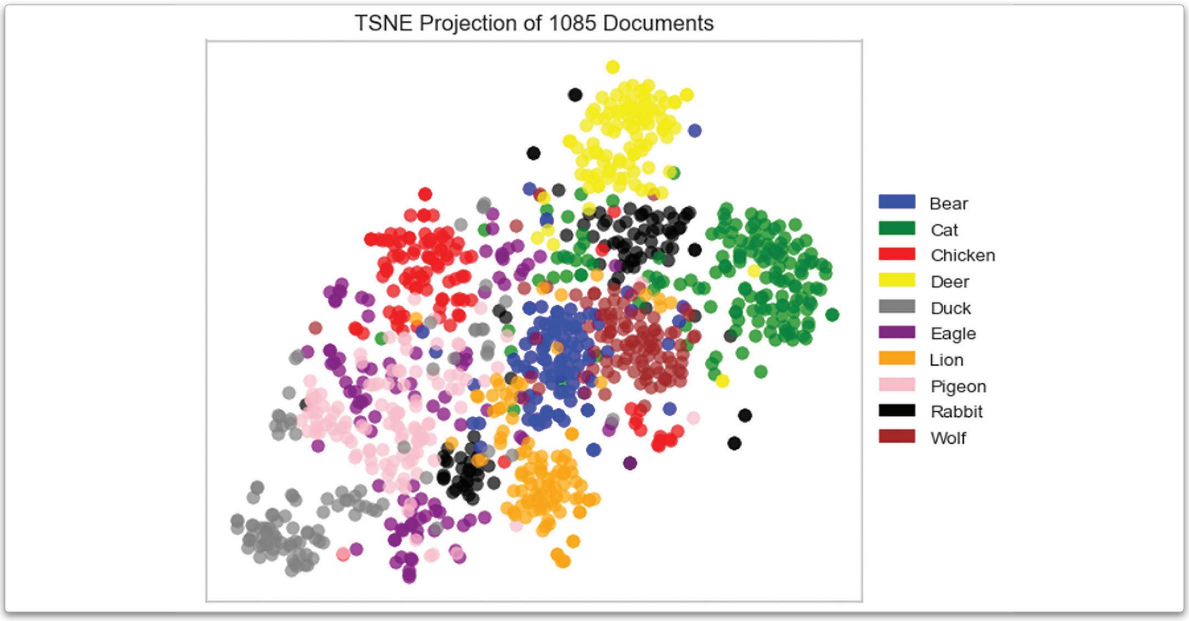


Figure 4.20: Clusters visualization

The visualization is much more promising than the one produced for the non-transformed data. Even though some impurities exist, the figure reveals clear and generally well-separated clusters. The dendrogram of this more promising dataset can now be computed:

```
plt.figure()
linkage_2 = hierarchy.linkage(X_hog,method = 'ward')
hierarchy.dendrogram(linkage_2)
plt.show()
```

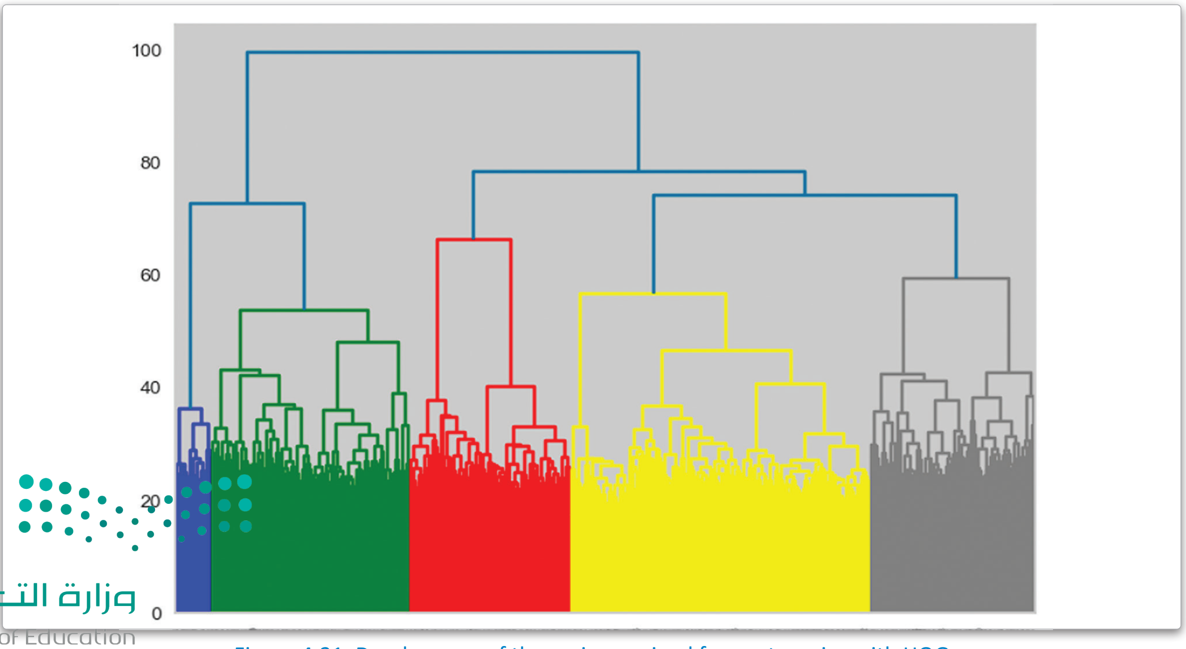


Figure 4.21: Dendrogram of the various animal face categories with HOG

The dendrogram suggests 5 clusters, exactly half of the correct number of 10. The following code adopts this suggestion, applies the AgglomerativeClustering tool, and reports the results for the three metrics:

```
AC = AgglomerativeClustering(linkage = 'ward', n_clusters = 5)
AC.fit(X_hog)
pred = AC.labels_

print('\nHomogeneity score:', homogeneity_score(y, pred))
print('\nAdjusted Rand score:', adjusted_rand_score(y, pred))
print('\nCompleteness score:', completeness_score(y, pred))
```

```
Homogeneity score: 0.4046340612330986

Adjusted Rand score: 0.29990205334627734

Completeness score: 0.6306921317302154
```

The results reveal that, even though the number of clusters that was used was significantly lower than the correct one, the results are far superior to those delivered when using the correct number on the non-transformed data.

This demonstrates the intelligence of the HOG transformation and validates that it can lead to impressive performance improvements for both supervised and unsupervised learning tasks in computer vision. To complete the analysis, the following code re-clusters the transformed data with the correct number of clusters:

```
AC = AgglomerativeClustering(linkage = 'ward', n_clusters = 10)
AC.fit(X_hog)
pred = AC.labels_

print('\nHomogeneity score:', homogeneity_score(y, pred))
print('\nAdjusted Rand score:', adjusted_rand_score(y, pred))
print('\nCompleteness score:', completeness_score(y, pred))
```

```
Homogeneity score: 0.5720932612704411

Adjusted Rand score: 0.41243540297103065

Completeness score: 0.617016965322667
```

As expected, the scores have increased overall. For instance, both homogeneity and completeness are now above 0.55, indicating that the algorithm does a better job both of placing animals from the same class in the same cluster and of creating "pure" clusters that mostly consist of the same animal class.

Clustering Using Neural Networks

The use of deep learning models (deep neural networks with multiple layers) has revolutionized the field of image clustering by providing powerful and highly accurate algorithms that can automatically group similar images together without the need for feature engineering. Many traditional image clustering methods rely on feature extractors to extract meaningful information from an image and use this information to group similar images together. This process can be time-consuming and requires domain expertise to design effective feature extractors. In addition, as seen in the previous lesson, even though feature descriptors such as the HOG transformation can indeed improve the results, they are far from perfect and there is certainly room for improvement.

Deep learning, on the other hand, has the ability to learn feature representations from the raw data automatically. This allows deep learning methods to learn highly discriminative features that capture the underlying patterns in the data, resulting in more accurate and robust clustering. To achieve this, several different layers are used in a neural network including:

- Dense layers
- Pooling layers
- Dropout layers

In the neural network of unit 3 lesson 1, a 300-neuron hidden layer of the Word2Vec model was used to represent each word. In that case, the Word2Vec model was pre-trained on a very large dataset with millions of stories from Google News. Pre-trained neural network models are also popular in the computer vision domain. A chracteristic example is the VGG16 model, which is commonly used for image recognition tasks.

VGG16 follows a deep CNN-based architecture with 16 layers. VGG16 is a supervised model that was trained on a large dataset of labeled images, called ImageNet. However, the training dataset for the VGG16 consists of millions of images and hundreds of different labels. This significantly improves the model's ability to understand the different parts of an image.

Similar to the simple CNN shown in the figure 4.22, VGG16 also uses a final dense layer with 4,096 neurons to represent each image, before feeding it to the output layer. This section demonstrates how VGG16 can be adapted for image clustering, even though it was originally designed for image classification:

Dense layer

A layer in neural networks Where the signals are passed from the nodes in the previous layer in the network to the nodes in the current layer by means of a specific weight, and an activation function is applied to the signals sent to the dense layer to generate the final output results.

Pooling layer

A layer in neural networks used to reduce the spatial dimensions of the input data.

Dropout layer

A regularization technique used to prevent overspecialization of a model to a dataset in neural networks by randomly dropping out nodes in the layer during each training iteration.

1. Load the pre-trained VGG16 model.
2. Remove the output layer of the model. This leaves the final dense layer as the new output layer.
3. Use the truncated model to map each of the images in the Animal Faces dataset to a numeric vector with 4,096 values.
4. Use Agglomerative Clustering to cluster the produced vectors.

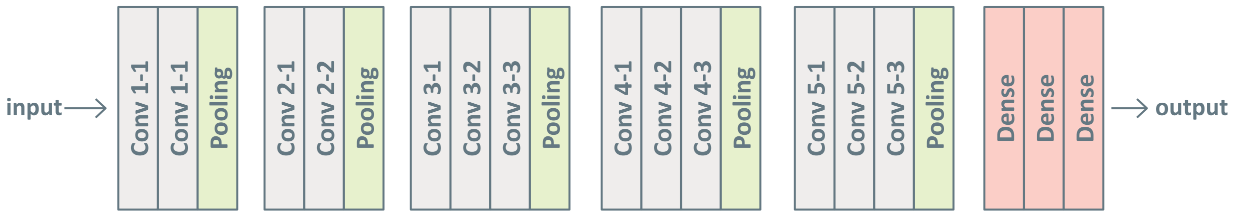


Figure 4.22: VGG16 architecture

The TensorFlow and Keras libraries that were introduced in the previous lesson can be used to access and truncate the VGG16 model. The first step is to import all the required tools:

```
from keras.applications.vgg16 import VGG16 # used to access the pre-trained VGG16 model
from keras.models import Model

model = VGG16() # loads the pretrained VGG16 model
# removes the output layer
model = Model(inputs = model.inputs, outputs = model.layers[-2].output)
```

Remove the last layer from the output.

The following code applies some basic preprocessing required by VGG16, such as scaling the RGB values to be between 0 and 1:

```
from keras.applications.vgg16 import preprocess_input
X_prep = preprocess_input(X)
X_prep.shape
```

```
(1085, 224, 224, 3)
```

Note that the shape of the data remains the same: 1,085 images, each with dimensions of 224 × 224 and 3 RGB channels. Next, the truncated model can be used to map each image to a vector of 4,096 numbers:

```
X_VGG16 = model.predict(X_prep, use_multiprocessing = True)
X_VGG16.shape
```

```
34/34 [==============================] - 57s 2s/step

(1085, 4096)
```

The multiprocessing=True parameter is set to speed up the process by computing the vectors for multiple images in parallel. Before proceeding with the clustering step, the following code is used to visualize the vectorized data:

```
tsne = TSNEVisualizer(colors = color_palette)
tsne.fit(X_VGG16, labels)
tsne.show();
```

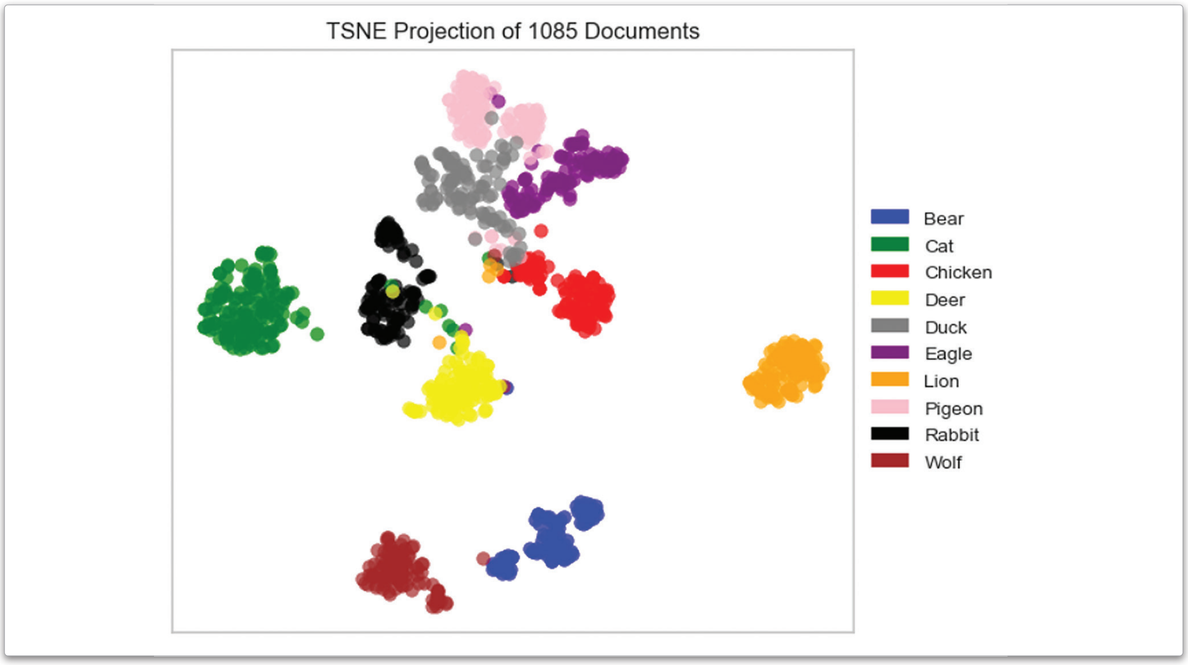


Figure 4.23: Clusters visualization

The results are impressive. The new visualization reveals clearly separated, near perfect clusters. The separation is also significnatly better than that in the HOG-transformed data.

```
linkage_3 = hierarchy.linkage(X_VGG16, method = 'ward')
plt.figure()
hierarchy.dendrogram(linkage_3)
plt.show()
```

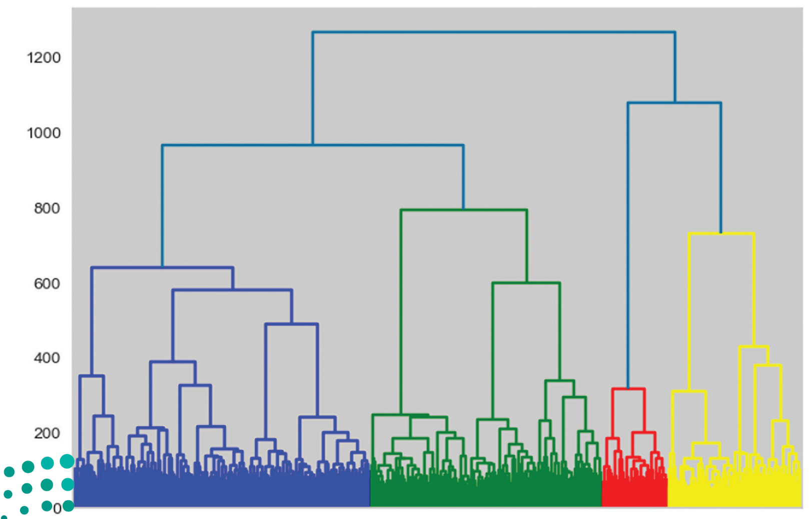


Figure 4.24: Dendrogram of the various animal face categories with VGG16

The dendrogram suggests 4 clusters. In this case, the practitioner can easily ignore this suggestion and instead follow the visualization above which clearly indicates the existence of 10 clusters.

The following code uses Agglomerative Clustering and reports the metric scores for both 4 and 10 clusters:

```
AC = AgglomerativeClustering(linkage = 'ward',n_clusters = 4)
AC.fit(X_VGG16)
pred=AC.labels_

print('\nHomogeneity score:', homogeneity_score(y, pred))
print('\nAdjusted Rand score:', adjusted_rand_score(y, pred))
print('\nCompleteness score:', completeness_score(y, pred))
```

```
Homogeneity score: 0.504687456015823

Adjusted Rand score: 0.37265351562538257

Completeness score: 0.9193141240200559
```

```
AC = AgglomerativeClustering(linkage='ward',n_clusters = 10)
AC.fit(X_VGG16)
pred=AC.labels_

print('\nHomogeneity score:', homogeneity_score(y, pred))
print('\nAdjusted Rand score:', adjusted_rand_score(y, pred))
print('\nCompleteness score:', completeness_score(y, pred))
```

```
Homogeneity score: 0.8403973102506642

Adjusted Rand score: 0.766734821176714

Completeness score: 0.8509145102288217
```

The results validate the evidence provided by the visualization. The transformations produced by VGG16 lead to vastly superior results for both 4 and 10 clusters. In fact, near-perfect scores for all three metrics were reported when using 10 clusters, verifying that the produced results are almost perfectly aligned with the animal classes in the dataset.

VGG16 is one of the earliest highly intelligent pre-trained CNN models for computer vision applications. However, many other intelligent pre-trained CNN models have been published and surpassed the performance of the VGG16 model.

Exercises

1 Mention an advantage that unsupervised vision techniques have over supervised techniques.

____

____

____

____

2 You are given a numpy array X_flat that includes flattened images. Each row in the array represents a different flattened image as a sequence of integers between 0 and 255. Complete the following code so that it uses Agglomerative Clustering to group the images from X_flat into 5 different clusters.

```
from ______________ import AgglomerativeClustering # used for agglomerative clustering

AC = AgglomerativeClustering(linkage='ward', ______________)

X_norm = ________________ # normalizes the data

AC.fit(X_norm) # applies the tool to the data

pred = AC.________________ # gets the cluster labels
```

3 List some advantages of using Deep Learning over other traditional image clustering methods?

____

____

____

____

____

4 You are given a numpy array X_flat that includes flattened images. Each row in the array represents a different flattened image as a sequence of integers between 0 and 255. Complete the following code so that it uses the ward method to create and visualize the dendrogram of the images in this array:

```
import scipy.cluster.hierarchy as hierarchy # visualizes and supports hierarchical clustering tasks

import ___________________ as plt

X_norm = _________________ # normalizes the data

plt.figure() # creates a new empty figure

linkage_flat=hierarchy.linkage(_______________, method='________________')

hierarchy.________________(linkage_flat)

plt.show() #shows the figure
```

5 Describe how clustering with neural networks is applied in image analysis.

Lesson 3

Generating Visual Data

Link to digital lesson

www.ien.edu.sa

Using AI to Generate Images

While the computer vision algorithms described in the previous two lessons of this unit focused on understanding the different aspects of a given image, the field of image generation in this lesson focuses on creating new images. The field of image generation has a long history, dating back to the 1950s & 1960s, when researchers first began experimenting with mathematical equations to create images. Today, the field has grown to encompass a wide range of techniques.

One of the earliest and most well-known techniques for image generation is the use of fractals. A fractal is a geometric shape or pattern that is self-similar, meaning that it looks the same at different zoom scales. The most famous fractal is the Mandelbrot set, which can be seen in figure 4.25.

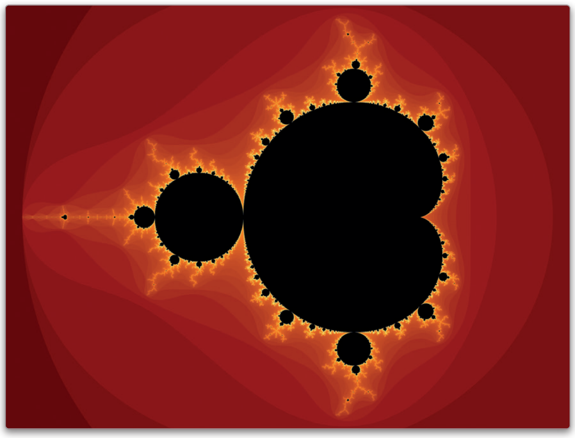


Figure 4.25: Mandelbrot fractal

In the late 20th century, researchers began to explore more advanced techniques for image generation, such as neural networks. One of the most popular techniques for image generation with neural networks is text-to-image synthesis. This technique involves training a neural network to generate images from textual descriptions. The neural network is trained on a dataset of images and their associated text descriptions. The network learns to associate certain words or phrases with specific features of an image, such as the shape or color of an object. Once trained, the network can be used to generate new images from text descriptions. This technique has been used to generate a wide range of images, from simple objects to complex scenes.

Another technique for image generation is image-to-image synthesis. This technique involves training a neural network on a dataset of images to learn to recognize the unique features of an image, in order to generate new images that are similar to the existing one, but with variations. Recently, researchers have been exploring text-guided image-to-image synthesis, which combines the strengths of text-to-image and image-to-image synthesis methods by allowing the user to guide the synthesis process using text prompts. This technique has been used to generate high-quality images that are consistent with a given text prompt while also being visually similar to an initial image.

Finally, another state-of-the-art technique is text-guided image-inpainting, which focuses on filling in missing or corrupted parts of an image based on a given text description. The text description provides information about what the missing or corrupted parts of the image should look like, and the goal of the inpainting algorithm is to use this information to generate a realistic and coherent image. This lesson provides practical examples for text-to-image, text-guided image-to-image, and text-guided image-inpainting generation.

Image Generation and Computational Resources

Image generation is a computationally intensive task, as it involves the use of complex algorithms that require large amounts of processing power. These algorithms typically involve the treating of large amounts of data, such as 3D models, textures, and lighting information, which can also contribute to the computational demands of the task.

One of the key technologies that is used to accelerate image generation is the use of Graphics Processing Units (GPUs). Unlike a traditional Central Processing Unit (CPU), which is designed to handle a wide range of tasks, a GPU is optimized for the types of mathematical operations required for image rendering and other graphics-related tasks. This makes them much more efficient at handling large amounts of data and performing complex calculations, which is why they are often used in image generation and other computationally intensive tasks.

Graphics Processing Unit (GPU)

A GPU is a specialized type of processor that is designed to handle the large number of mathematical operations required for rendering images and video.

This lesson demonstrates how you can utilize the popular Google Colab platform to get access to a powerful GPU-based infrastructure at no cost, using only a standard google account. Google Colab is a free cloud-based platform that allows users to write and execute code, run experiments, and train models in a Jupyter Notebook environment.

To access Google Colab:

> Go to **https://colab.research.google.com** 1
> Sign in with your **Google** account. 2
> Click on **Edit > Notebook settings.** 3
> Choose **GPU** 4 and click **Save.** 5

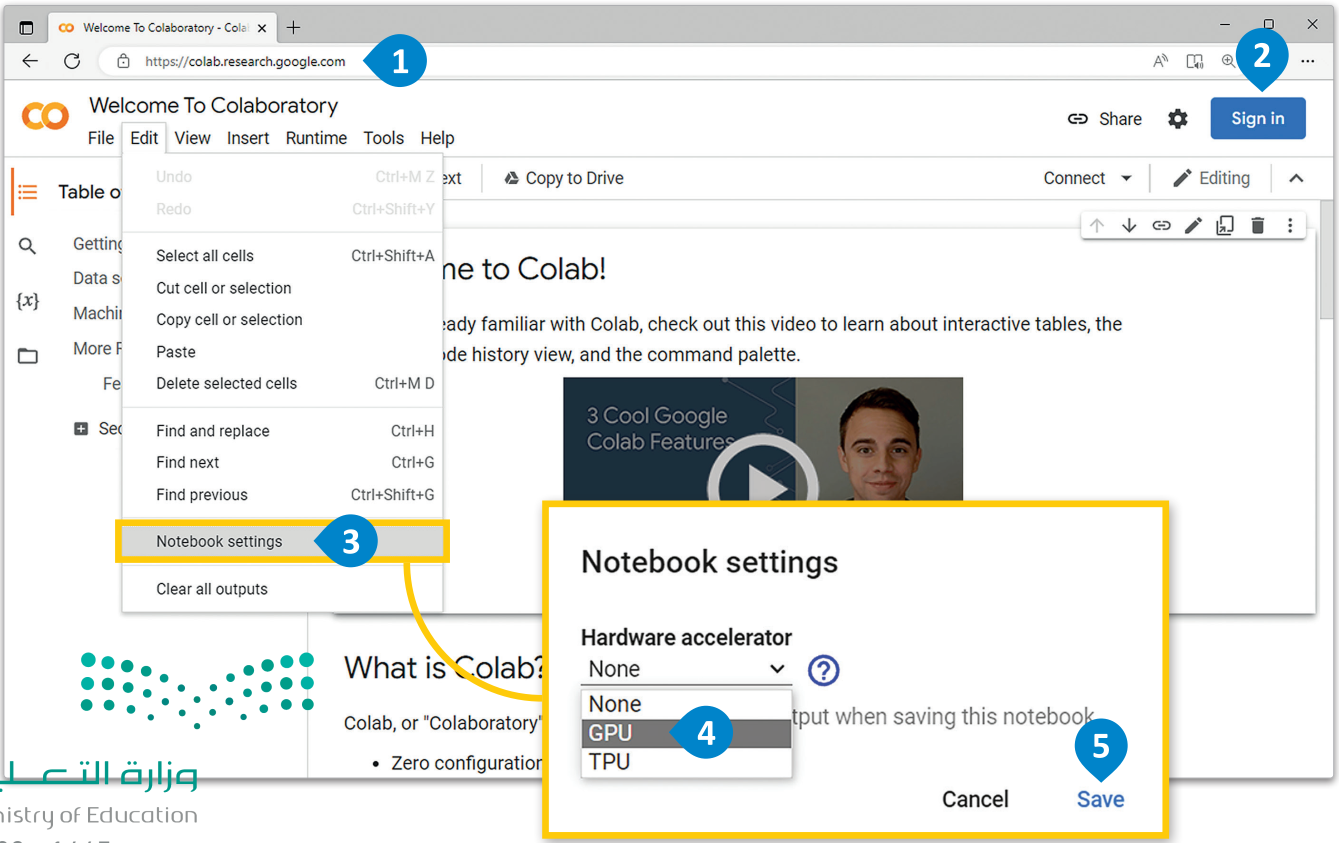


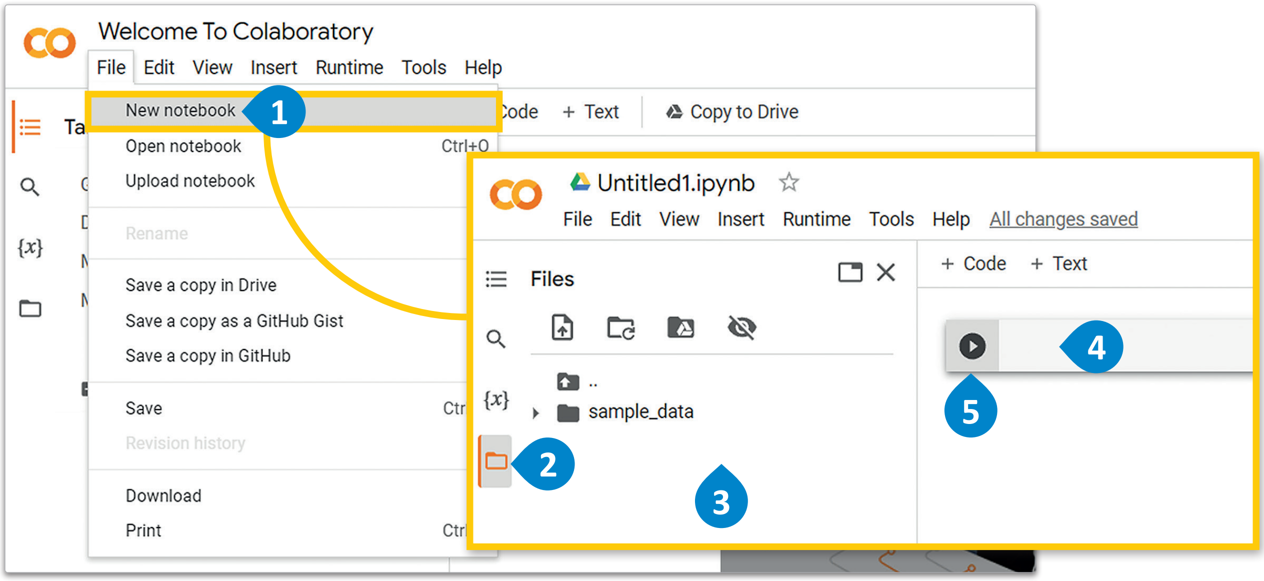
Figure 4.26: Accessing Google Colab

To use Python Notebook:

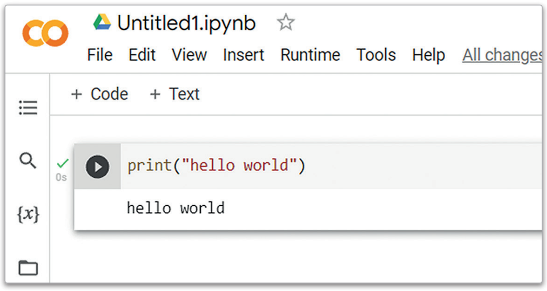
> Click on **File > New notebook.** 1

> Click **Files** 2 and inside the adjacent area that unfolds **drag and drop** the **images** you will be using in the lesson. 3

> You can now type your python code inside the code cell 4 and run it by clicking the button beside. 5



The Google Colab environment works similarly to Jupyter Notebook. Below is the classic "Hello World" example:



```
print("hello world")
```

```
hello world
```

Figure 4.27: Use a Python Notebook.

The image generation algorithms described in this chapter are designed to be creative, and are thus not determistic. This means that they are not guaranteed to always generate the exact same image for the same input. The generated images included in this chapter are thus just examples of the possible images that can be generated by the code.

Diffusion Models and Generative Adversarial Networks

In recent years, the field of image generation has seen significant progress, with the development of various techniques and models that can generate realistic and high-quality images from different sources of information. Two of the most popular and widely used techniques for image generation are Generative Adversarial Networks (GANs) and Stable Diffusion.

In this section, you will be introduced to the main concepts and techniques behind GANs and Stable Diffusion and provide an overview of their applications in image generation. Furthermore their similarities and differences will be discussed and the pros and cons of each approach.

Generating Images with Generative Adversarial Networks (GANs)

GANs are a class of generative models that consist of two main components: a generator and a discriminator. The generator generates fake images, while the discriminator tries to distinguish the generated images from real images. The two components are trained in an adversarial way, where the generator tries to "trick" the discriminator, and the discriminator tries to become better at detecting fake images.

One of the main advantages of GANs is that they can generate high-quality and realistic images that are difficult to distinguish from real images. However, GANs also have some limitations, such as non-convergence which means generator and discriminator networks do not improve over time, and mode collapse in outputs, which means often repeating the same or similar outputs, regardless of the input noise or data.

The generator and the discriminator in GANs are typically implemented using Convolutional Neural Networks (CNNs) or a similar architecture.

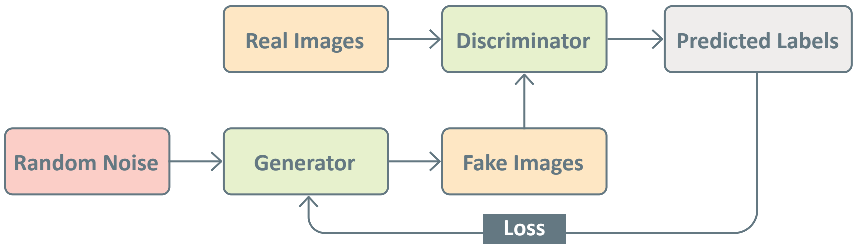


Figure 4.28: GAN architecture

Generating Images with Stable Diffusion

Stable Diffusion is a deep learning model for text-to-image generation. The method consists of two main components: a text encoder and a visual decoder. The text encoder and visual decoder are trained together on a dataset of paired text and image data, where each text input is associated with one or more corresponding images.

The text encoder is a neural network that takes in text input (such as a sentence or a paragraph) and maps it to an embedding: a numeric vector with a fixed number of values. This embedding representation captures the meaning of the input text. A similar approach is used by the Word2Vec and SBERT models that were covered in unit 3 and generate embeddings for individual words and sentences, respectively.

The text embedding created by the encoder is then passed through the visual decoder to generate an image. The visual decoder is also a type of neural network and is typically implemented using a CNN or a similar architecture. The generated image is compared with the corresponding real image from the dataset, and the difference between them is used to compute the loss. The loss is then used to update the parameters of the text encoder and visual decoder to minimize the difference between the generated images and the real images.

Table 4.4: Stable Diffusion training process

1.	Pass the text input through the text encoder to get the text embedding.
2.	Pass the text embedding through the visual decoder to generate an image.
3.	Compute the loss (difference) between the generated image and the corresponding real image.
4.	Use the loss to update the parameters of the text encoder and visual decoder. At a high level, this includes rewarding the neurons that helped reduce the loss and "punishing" the neurons that contributed to its increase.
5.	Repeat the above steps for multiple text-image pairs in the dataset.

Both GANs and Stable Diffusion models have delivered impressive results in the field of image generation. The remainder of this lesson focuses on providing practical Python examples for the diffusion-based approach, which is currently considered the state-of-the-art.

As described before, image generation is a computationally intensive task. It is therefore strongly encouraged that you run all Python examples on the Google Colab platform or a different GPU-powered infrastructure that you may have access to.

This chapter utilizes the "diffusers" library, which is currently considered the best open-source library for diffusion-based models. The following code installs the library, as well as some additional required libraries:

```
%%capture
!pip install diffusers
!pip install transformers
!pip install accelerate

import matplotlib.pyplot as plt
from PIL import Image # used to represent images
```

Text-to-Image Generation

This section demonstrates how the diffusers library can be used to generate images based on text prompt provided by the user. The examples in this section utilizes "stable-diffusion-v1-4", a popular pretrained model for text-to-image generation.

```
# a tool used to generate images using stable diffusion
from diffusers import DiffusionPipeline
generator = DiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4")
# specifies what GPUs should be used for this generation
generator.to("cuda")

image = generator("A photo of a white lion in the jungle.").images[0]
plt.imshow(image);
```

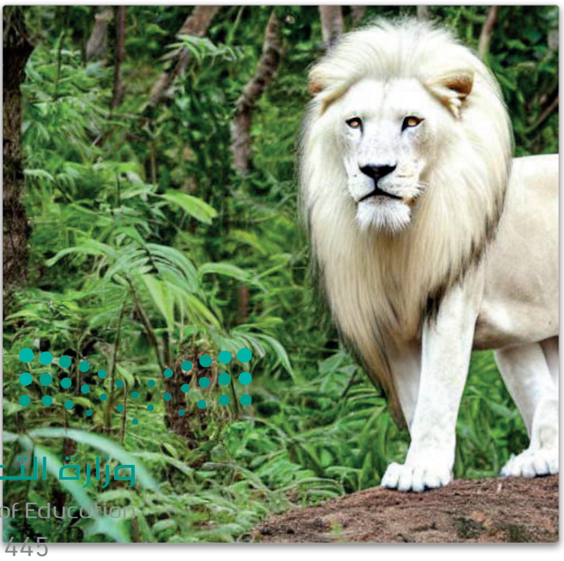


Figure 4.29: Generated image of a white lion in the jungle

The model responds to the "A photo of a white lion in the jungle" prompt with an impressive and very realistic image as shown in figure 4.29. Experimenting with creative prompts is the best way to gain experience and understand the capabilities and limitations of this approach.

INFORMATION

CUDA (Compute Unified Device Architecture) is a parallel computing platform that enables the use of GPUs.

The following prompt adds an additional dimension to the generation process, by asking for a white lion painted in the specific style of Pablo Picasso, one of the most famous artists of the twentieth-century.

```
image = generator("A painting of a white lion in the style of Picasso.").
images[0]
plt.imshow(image);
```

Again, the results are impressive and demonstrate the creativity of the stable diffusion process. The produced image is indeed that of a white lion. However, contrary to the previous prompt, the new prompt leads to painting-like rather than photo-like images. In addition, the painting's style is indeed remarkably similar to that followed by Pablo Picasso.

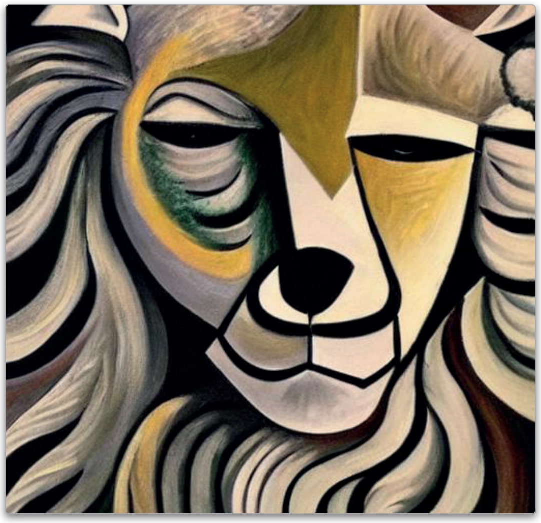


Figure 4.30: Generated image of lion in Picasso style

Image-to-Image Generation with Text Guidance

The next example uses the diffusers library to generate an image based on two inputs: an existing image, which serves as the basis for the new image that will be generated and a text prompt that describes what the produced image should look like. While the text-to-image task demonstrated in the previous section was only limited by a text prompt, this new task has to ensure that the new image is both similar to the original and an accurate visual of the description given in the text prompt.

```
# pipeline used for image to image generation with stable diffusion
from diffusers import StableDiffusionImg2ImgPipeline
# loads a pretrained generator model
generator = StableDiffusionImg2ImgPipeline.from_pretrained("runwayml/stable-
diffusion-v1-5")
# moves the generator model to the GPU (CUDA) for faster processing
generator.to("cuda")

init_image = Image.open("landscape.jpg")
init_image.thumbnail((768, 768)) # resizes the image to prepare it as input of the model
plt.imshow(init_image);
```

The example on figure 4.31 uses the pretrained model "stable-diffusion-v1-5", which is appropriate for image-to-image generation with text guidance.

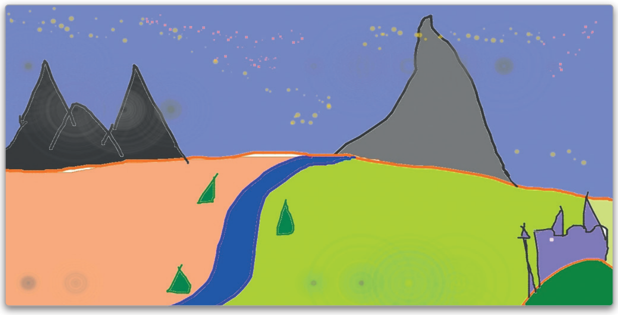


Figure 4.31: Original landscape image

```
# a detailed prompt describing the desired visual
# for the produced image
prompt = "A realistic mountain
landscape with a large castle."
image = generator(prompt=prompt,
image = init_image, strength=0.75).
images[0]
plt.imshow(image);
```

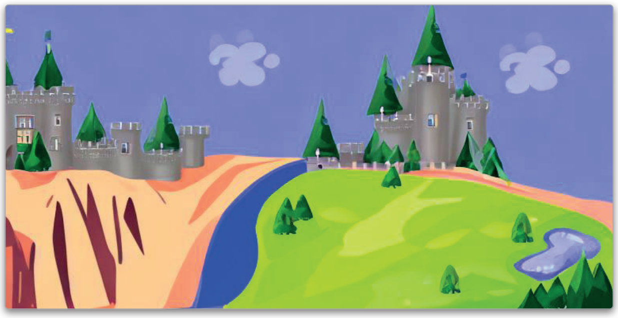


Figure 4.32: Generated landscape image with strength=0.75

The model indeed generates an image that is both faithful to the text prompt and visually similar to the original image. The "strength" parameter is used to control the visual difference between the original and new images. The parameter takes values between 0 and 1, with higher values allowing the model to be more flexible and less constrained by the original image. For example, the following code uses the exact same prompt with a strength=1.

```
# generate a new image based on the prompt and the
# initial image using the generator model
image = generator(prompt=prompt,
image = init_image, strength=1).images[0]
plt.imshow(image);
```

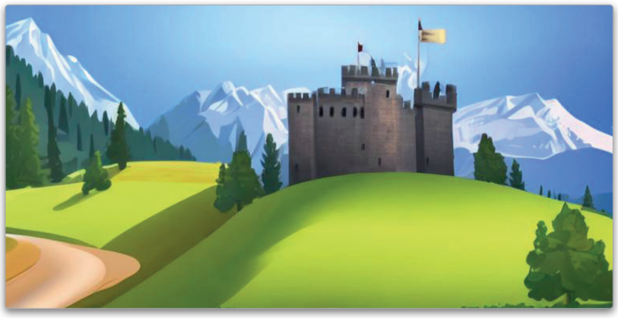


Figure 4.33: Generated landscape image with strength=1

The resulting image in figure 4.33 verifies that increasing the value of the strength parameter leads to a visual that fits even better with the guidance offered by the text prompt, but is also significantly less similar to the input image.

Another characteristic example is shown below. Its output is shown on figure 4.34.

```
init_image = Image.open("cat_1.jpg")
init_image.thumbnail((768, 768))
plt.imshow(init_image);
```

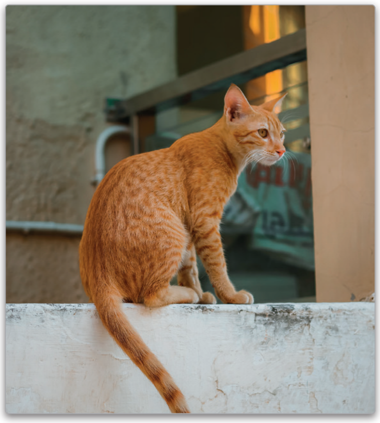


Figure 4.34: Original cat image

The following code will now be used to convert this to a photo of a tiger:

```
prompt = "A photo of a tiger"
image = generator(prompt=prompt, image=init_image, strength=0.5).images[0]
plt.imshow(image);
```

The first attempt is bound by the value of the strength parameter, leading to a picture that appears to be a mix between a tiger and the cat from the original photo as shown in figure 4.35. The new picture indicates that the algorithm did not have enough "strength" to properly convert the face of the cat to that of a tiger. The background remains highly similar to that of the original image.

Next, the strength parameter is increased to allow the model to move further away from the original image and closer to the text prompt:

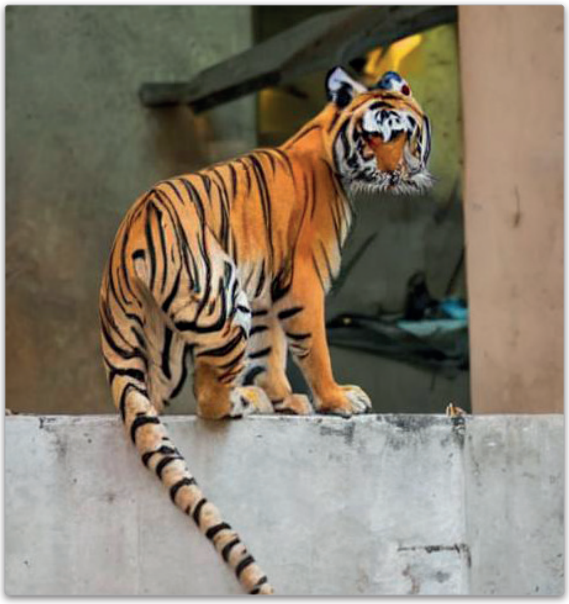


Figure 4.35: Generated tiger image with strength=0.5

```
image = generator(prompt=prompt,
image = init_image, strength=0.75).
images[0]
plt.imshow(image);
```

Indeed, the new image displayed is a tiger. However, notice how the surroundings, posture and angles of the animal remains very similar to the original. This demonstrates that the model is still aware of the original image and tried to maintain elements that did not have to be changed to get closer to the text prompt.

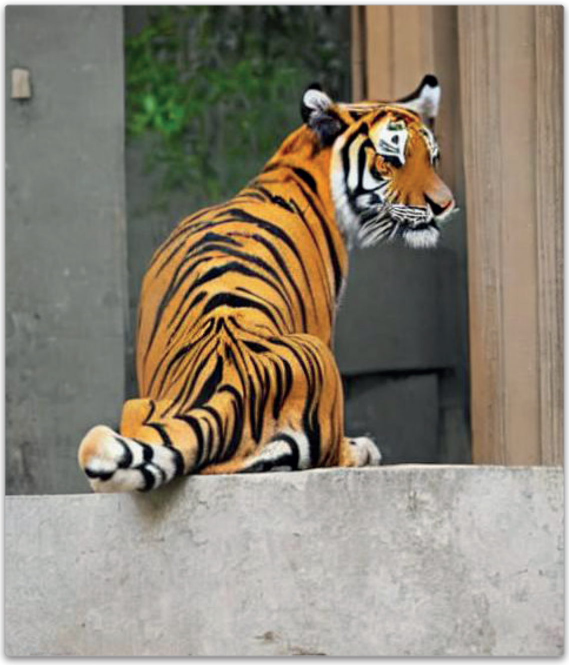


Figure 4.36: Generated tiger image with strength=0.75

Text-Guided Image-Inpainting

The next example focuses on using stable diffusion to replace specific parts of a given image with a new visual described by a text prompt. The "stable-diffusion-inpainting" pretrained model is used for this purpose. The following code loads the image of a cat on a bench and a "mask" isolates the specific parts of the image that are covered by the cat

```python
# tool used for text-guided image in-painting
from diffusers import StableDiffusionInpaintPipeline
init_image = Image.open("cat_on_bench.png").resize((512, 512))
plt.imshow(init_image);
mask_image = Image.open("cat_mask.jpg").resize((512, 512))
plt.imshow(mask_image);
```

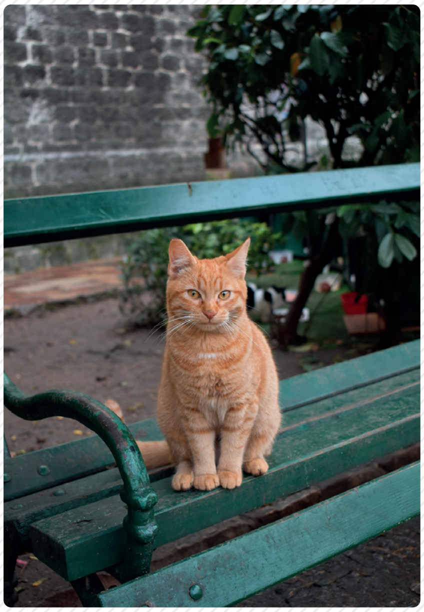


Figure 4.37: Original cat image

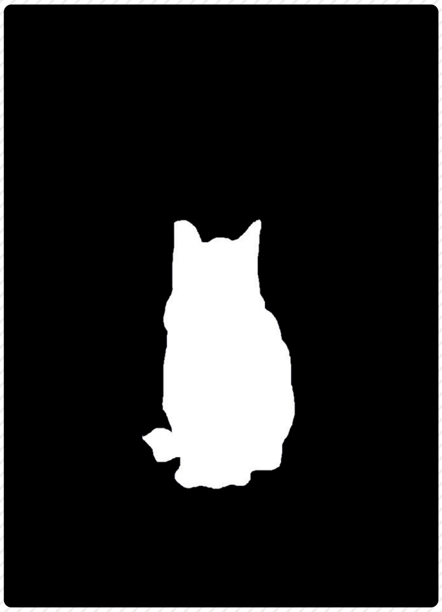


Figure 4.38: Cat image mask

The mask is a simple black and white image that has the exact same dimensions as the original. The parts that are replaced in the new image are highlighted in white, while every other part of the mask is black. Next, the pretrained model is loaded and a prompt is created to replace the cat in the original picture with an astronaut as you can see in figure 4.39.

```python
generator = StableDiffusionInpaintPipeline.from_pretrained("runwayml/stable-
diffusion-inpainting")
generator = generator.to("cuda")

prompt = "A photo of an astronaut"
image = generator(prompt=prompt, image=init_image, mask_image=mask_image).
images[0]
plt.imshow(image);
```

The new image successfully replaces the cat from the original image with a very realistic visual of an astronaut. In addition, this visual blends smoothly with the background elements and lighting of the image.

In fact, even a simpler, less accurate mask is sufficient to produce a realistic replacement. Consider the following input image and mask:

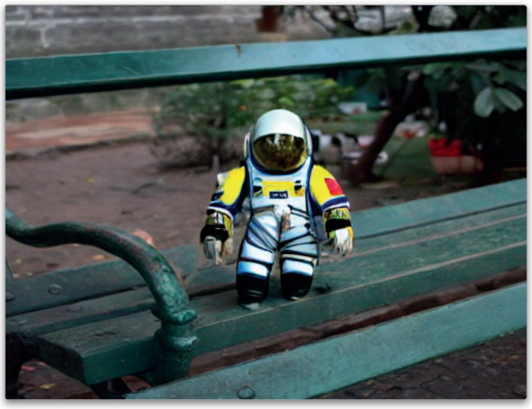


Figure 4.39: Generated astronaut image

```
init_image = Image.open("desk.jpg").resize((512, 512))
plt.imshow(init_image);
mask_image = Image.open("desk_mask.jpg").resize((512, 512))
plt.imshow(mask_image);
```

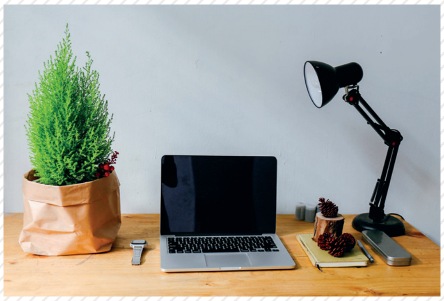


Figure 4.40: Original desk image

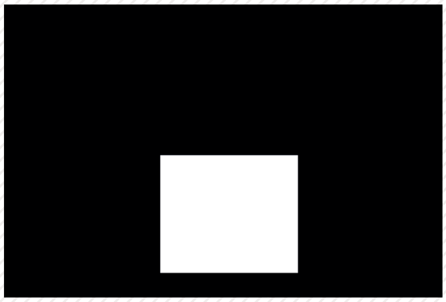


Figure 4.41: Desk image mask

In this example, the mask covers the laptop at the middle of the image. The following prompt and code are then used to replace the laptop with a photo of book:

```
prompt = "A photo of a book"
image = generator(prompt=prompt, image=init_image, mask_image=mask_image).
images[0]
plt.imshow(image);
```

Despite the fact that the prompt asked for the introduction of an object (book) that was significantly different from the one that was being replaced (laptop), the model did a good job of blending shapes and colors to create an accurate visual. With the continued advancement of machine learning and computer graphics technologies, it is likely that even more impressive and realistic images will be generated in the future .

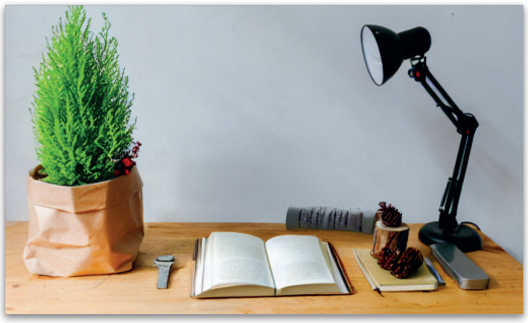


Figure 4.42: Generated desk image with book

Exercises

1 Give a brief description of text-guided image inpainting.

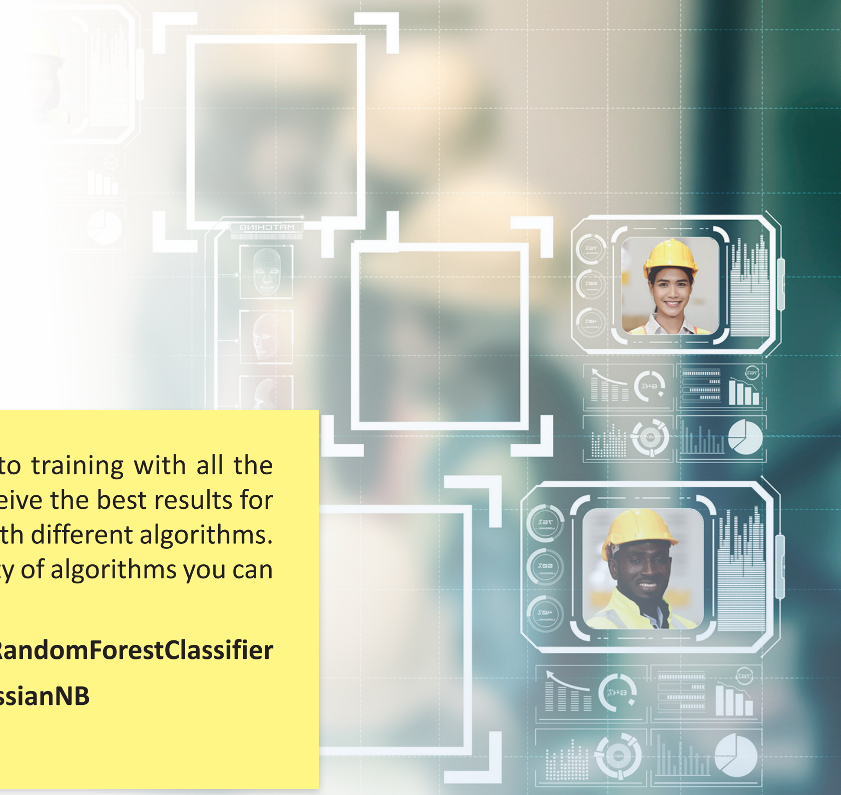
2 Describe the training process for Stable Diffusion models.

3 Describe the generator and discriminator components in Generative Adversarial Networks.

4 Use the DiffusionPipeline tool from the diffusers library to create a photo of your favorite animal eating your favorite food. Use the Google Colab platform for this task.

5 Use the StableDiffusionImg2ImgPipeline tool from the diffusers library to transform the animal in the photo from the previous exercise to a different animal of your choice. Use the Google Colab platform for this task.

Project



Not every dataset responds the same to training with all the classification algorithms. In order to receive the best results for your dataset, you have to experiment with different algorithms. The Python Sklearn library offers a variety of algorithms you can try, including the ones below:

> from sklearn.ensemble.forest import **RandomForestClassifier**

> from sklearn.naive_bayes import **GaussianNB**

> from sklearn.svm import **SVC**

1. Use the training set of the animal faces to train a model that achieves the highest possible accuracy on the testing set.

2. Replace the SGDClassifier library with each of the algorithms mentioned above (RandomForestClassifier, GaussianNB, SVC) and try to find the best one.

3. Re-run your notebook after each replacement to compute the accuracy of each new model that you try.

4. Create a report that compares the accuracy of all the models that you tried and identifies the one that achieved the best accuracy.

Wrap up

Now you have learned to:

> Prepare images for recognition.

> Use libraries and functions to create supervised learning models to classify images.

> Describe how a neural network is structured.

> Use libraries and functions to create unsupervised learning models to cluster images.

> Create images by providing a text prompt.

> Fill missing fragments of an image with realistic data.

KEY TERMS

Computer vision	Generative Adversarial Network (GAN)	Image Recognition
Convolutional Neural Network (CNN)	Histogram of Oriented Gradients (HOG)	Network Layer
Diffusion Model	Image Generation	Stable Diffusion
Feature Engineering	Image Preprocessing	Standard Scaling
Feature Selection		Visual Data

5. Optimization & Decision-making Algorithms

In this unit, you will study various algorithms and techniques that help to find the most efficient solutions to complex optimization problems. You will learn how optimization and decision-making algorithms work and how they can be applied to solve real-world problems related to resource allocation, scheduling, and route optimization.

Learning Objectives

In this unit, you will learn to:

> classify optimization approaches to address complex problems.

> describe different decision-making algorithms.

> use Python to solve team-based resource allocation problems.

> solve scheduling problems by using optimization algorithms.

> use Python to solve scheduling problems.

> use mathematical programming to solve optimization problems.

> define the Knapsack problem.

> define the Traveling Salesman problem.

Tools

> Jupyter Notebook

Lesson 1

Resource Allocation Problem

Link to digital lesson

www.ien.edu.sa

Optimization Algorithms in AI

AI is being used in various industries to make decisions that are efficient and accurate. One way AI is used to make decisions is through the use of machine learning algorithms. As you learned in the previous unit, machine learning algorithms enable AI to learn from data and make predictions or recommendations. For example, in health care, AI can be used to predict patient outcomes and recommend treatment plans based on data collected from similar cases. In finance, AI can be used to make investment decisions by analyzing large sets of financial data and identifying patterns that indicate potential risks or opportunities.

Even though machine learning algorithms are increasingly popular, they are not the only type of AI algorithm that can be used to make decisions. Another approach is to use optimization algorithms, which are generally used to find the best solution to a problem based on certain constraints and objectives.

Constraints

Constraints are restrictions on the solution, such as a maximum weight limit for a package being shipped.

Objective functions

Objective functions are measures of how well the solution meets desired outcomes, such as minimizing the travel distance for a delivery truck.

The purpose of optimization is to achieve the "best" design relative to a set of prioritized criteria or constraints. These include maximizing factors such as productivity, reliability, longevity and efficiency, and in the same time, minimizing other factors such as costs, waste, downtime, and errors.

Allocation Problems

Allocation problems are common optimization problems in which a set of resources, such as workers, machines, or funds, need to be assigned to a set of tasks or projects in the most efficient way possible. They arise in a wide range of fields, including manufacturing, logistics, project management, and finance, and can be formulated in various ways depending on their constraints and objectives. In this lesson, you will learn about allocation problems and the optimization algorithms used to solve them.

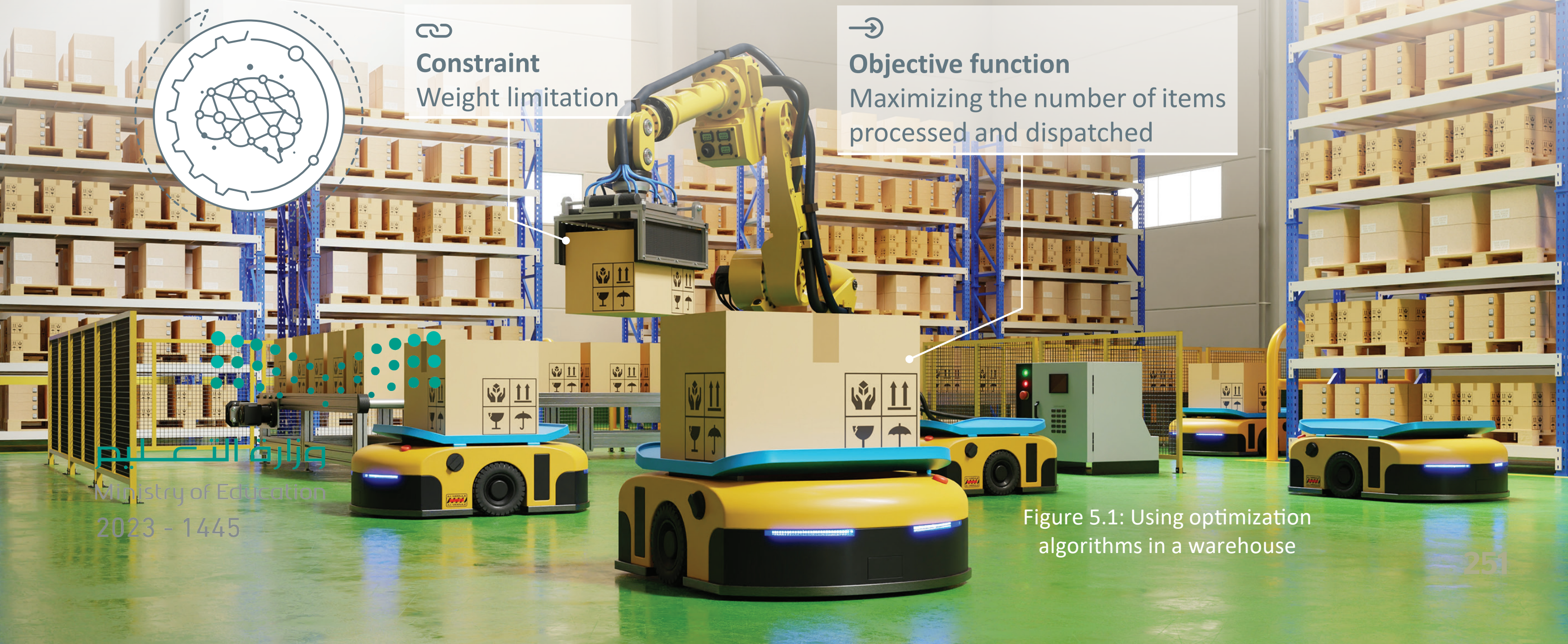


Figure 5.1: Using optimization algorithms in a warehouse

Next, you will look at a number of examples each with their own domain specific constraints and objectives.

	Constraints	Objective functions
Trasportation companies	- Time windows for deliveries, to ensure that packages are delivered within a specific time frame. - The availability and capacity of delivery vehicles, to make sure that the right vehicle is used for each delivery and that it can carry the necessary amount of packages. - The availability and shift patterns of drivers and other employees, to ensure that they work efficiently and are not overworked.	- **Minimizing** delivery time and distance traveled to reduce costs and improve efficiency. - **Maximizing** the number of packages delivered per vehicle to reduce the number of trips needed. - **Maximizing** customer satisfaction by delivering packages on time and within a specific time frame.
Airline scheduling	- Aircraft availability and maintenance schedules, to ensure that all airplanes are well-maintained and available for flights. - Air traffic control restrictions, to avoid delays and reduce fuel consumption. - Passenger demand and preferences, to schedule flights that are most convenient for passengers.	- **Minimizing** flight delays and cancellations to improve customer satisfaction. - **Maximizing** aircraft utilization to reduce costs and improve efficiency. - **Maximizing** revenue by offering flights that are in high demand and adjusting ticket prices based on demand.
Manufacturers	- Production capacity and lead times, to ensure that products are produced on time. - Material availability and storage capacity, to avoid stockouts or overstocking. - Demand fluctuations, to adjust production schedules based on changes in customer demand.	- **Minimizing** production costs by optimizing the use of resources and reducing waste. - **Maximizing** production efficiency by scheduling production runs to minimize setup times and changeovers. - **Maximizing** customer satisfaction by ensuring products are available when needed.
Inventory management in companies	- Limited storage capacity, which requires careful management of inventory levels. - Delivery lead times and variability, which affect how much inventory needs to be held at any given time. - Budget availability for purchasing inventory.	- **Maximing** profit by securing sufficient inventory levels for high-margin items. - **Minimizing** storage costs by optimizing inventory levels based on demand forecasts. - **Maximizing** customer satisfaction by ensuring that the right products are available at the right time and at the right location, and by minimizing stockouts, delays, and other disruptions that can impact the customer experience.
Power companies	- Electricity demand and fluctuations. - The availability of necessary raw materials and energy sources. - Transmission and distribution constraints, such as grid capacity and distance between power plants and consumers.	- **Minimizing** the cost of generating and distributing electricity by optimizing the use of resources. - **Minimizing** power losses and service failures.

All of the above applications can be modeled as complex problems with a vast number of possible solutions. For instance, consider a classic resource-allocation problem focused on team formation. This problem arises when we have:

- a large pool of workers with different skill sets, and
- a task that requires a specific subset of skills in order to be completed.

The objective is to create the smallest possible team of workers, while satisfying the constraint that the members of the team should be able to collectively cover all the skills required by the task.

For instance, assume a simple scenario with five workers:

Worker 1	**Worker 2**	**Worker 3**	**Worker 4**	**Worker 5**
Skills: **s1**, **s3**, **s6**	Skills: **s2**, **s3**	Skills: **s1**, **s2**, **s3**	Skills: **s2**, **s4**	Skills: **s5**

The task to be completed requires all skills s1, s2, s3, s4, s5, and s6. A brute-force solution would be to consider all possible teams of workers, focus on the teams that cover all required skills, and choose the team with the smallest size. Assuming that each team must have at least one person, you can create a total of 31 different teams with 5 workers.

Brute-force

Brute-force is a method of problem-solving that involves systematically trying every possible solution to the problem in order to find the optimal solution, regardless of computational cost.

- For a team of size 1, there are 5 ways to choose 1 out of 5 workers.
- For a team of size 2, there are 10 ways to choose 2 out of 5 workers.
- For a team of size 3, there are 10 ways to choose 3 out of 5 workers.
- For a team of size 4, there are 5 ways to choose 4 out of 5 workers.
- For a team of size 5, there is only 1 way to choose all 5 workers.

The total number of possible different teams you can create is: 5 + 10 + 10 + 5 + 1 = 31 The number can also be computed as 2^5 - 1.

Evaluating all 31 possible teams would reveal that the best possible solution is creating a team that includes workers 1, 4 and 5. That team would cover all six required skills and would include three workers. It is not possible to cover all the skills with a smaller team, making this the optimal solution.

Worker 1	**Worker 4**	**Worker 5**
Skills: **s1**, **s3**, **s6**	Skills: **s2**, **s4**	Skills: **s5**

Another solution would be a team that includes workers 1, 2, 3, and 5. While this team indeed covers all six skills, it also requires more workers. This makes it a feasible but suboptimal solution.

Worker 1	**Worker 2**	**Worker 3**	**Worker 5**
Skills: **s1**, **s3**, **s6**	Skills: **s2**, **s3**	Skills: **s1**, **s2**, **s3**	Skills: **s5**

The exhaustive nature of the brute-force approach guarantees that it will always find the optimal solution, as long as one exists. However, examining all possible teams comes at a high computational cost:

- If we have 6 workers, the number of possible teams is $2^6 - 1 = 63$
- If we have 10 workers, the number of possible teams is $2^{10} - 1 = 1,023$
- If we have 15 workers, the number of possible teams is $2^{15} - 1 = 32,767$
- If we have 20 workers, the number of possible teams is $2^{20} - 1 = 1,048,575$
- If we have 50 workers, the number of possible teams is $2^{50} - 1 = 1,125,899,906,842,623$

Clearly, in such settings, exhaustive enumeration of all possible solutions is not a practical option. Various optimization approaches have been proposed to address complex problems by searching the space of possible solutions in ways that are much more efficient than the brute-force approach. These approaches can be broadly classified into three categories:

Even for a modest number of 50 workers, the number of possible teams explodes to over one quadrillion!

- **heuristic methods**
- **constraint programming**
- **and mathematical programming.**

Optimal Solution

It is possible for multiple optimal solutions to exist. In this example, this would mean multiple teams that include three members and can cover all required skills. It is also possible that no feasible solution exists for some problems. For example, if the given task required a skill s7 that none of the workers possessed, then there would be no feasible solution.

Heuristic Methods

Heuristic methods are typically based on experience, rules of thumb, or common sense, rather than on a rigorous mathematical analysis. They can be used to find good solutions quickly, but do not guarantee an optimal (best possible) solution. Examples of heuristics include greedy algorithms, simulated annealing, genetic algorithms, and ant colony optimization. These methods are typically used for complex problems in which the computation time is too high and finding exact solutions is not feasible. You will learn more about these algorithms in the following lessons.

+ Pros

Heuristics are computationally efficient, can handle complex problems, and can find good quality solutions if a reasonable heuristic is used.

- Cons

They do not guarantee an optimal solution and some heuristics require significant tuning to deliver good results.

Constraint Programming

Constraint Programming (CP) solves optimization problems by modeling the constraints and finding a solution that satisfies all the constraints. This approach is particularly useful for problems that have a large number of constraints or that require the optimization of several objectives.

+ Pros

CP can handle complex constraints and can find optimal solutions.

- Cons

These methods can also be computationally expensive for large problems.

Mathematical Programming

Mathematical Programming (MP) is a family of techniques that uses mathematical models to solve optimization problems. MP includes Linear Programming, Quadratic Programming, Nonlinear Programming, and Mixed-Integer Programming. These techniques are widely used in many areas, including economics, engineering, and operations research. MP techniques also play a crucial role in deep learning. Deep learning models typically have a large number of parameters that need to be learned from data. Optimization algorithms are used to adjust the parameters of the model in order to minimize a cost function that measures the difference between the predicted output from the model and the true output. Several optimization algorithms, such as Adam, AdaGrad, and RMSprop have been developed specifically for deep learning models.

+ Pros

MP can handle a wide range of optimization problems and can often guarantee an optimal solution.

- Cons

The computational cost for large problems and the complexity of creating an appropriate mathematical formulation are high for complex real-world problems.

A Working Example: Optimization for the Team-Formation Problem

This lesson will initially demonstrate the use of a brute-force algorithm and a greedy heuristic algorithm for solving a decision problem focused on the team-based resource allocation problem described above. Then, the results of the two algorithms will be compared.

The following function can be used to create randomized instances of the team formation problem. It allows the user to specify four parameters: the total number of skills to be considered, the total number of available workers, the number of skills that the members of a team have to collectively cover in order to complete a task, and the maximum number of skills that each worker can have.

The function then creates and returns a set of workers with different skillsets, as well as the set of required skills. The function uses the popular "random" library, which can be used to sample random numbers from a given interval or random elements from a given list.

Greedy Heuristic Algorithm

A greedy algorithm is a heuristic approach to problem-solving, where the algorithm constructs the solution step-by-step, selects the locally optimal choice at each stage, hoping to eventually reach a global optimum.

```
import random

def create_problem_instance(skill_number, # total number of skills
                            worker_number, # total number of workers
                            required_skill_number, # number of skills the team has to cover
                            max_skills_per_worker # max number of skills per worker
                              ):

    # creates the global list of skills s1, s2, s3, ...
    skills = ['s' + str(i) for i in range(1, skill_number+1)]

    worker_skills = dict() # dictionary that maps each worker to their set of skills
    for i in range(1, worker_number+1): # for each worker

        # makes a worker id (w1, w2, w3, ...)
        worker_id = 'w' + str(i)

        # randomly decides the number of skills that this worker should have (at least 1)
        my_skill_number = random.randint(1, max_skills_per_worker)

        # samples the decided number of skills
        my_skills = set(random.sample(skills, my_skill_number))

        # remembers the skill sampled for this worker
        worker_skills[worker_id] = my_skills

    # randomly samples the set of required skills that the team has to cover
    required_skills = set(random.sample(skills, required_skill_number))

    # returns the worker and required skills
    return {'worker_skills':worker_skills, 'required_skills':required_skills}
```

Now, you will test the above function by creating a problem instance with 10 total skills, 6 workers, 5 required skills, and at most 5 skills per worker.

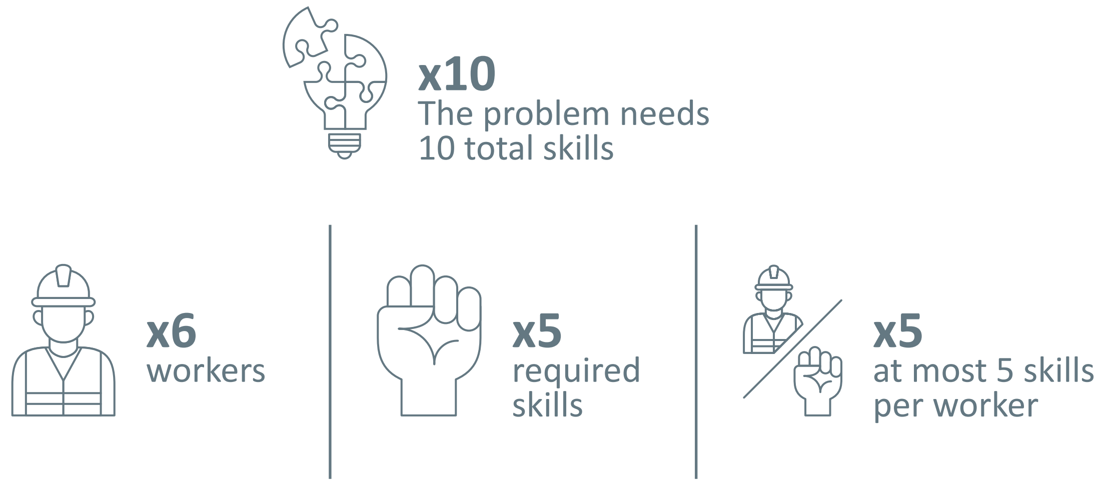

x10
The problem needs 10 total skills
x6
workers
x5
required skills
x5
at most 5 skills per worker

Figure 5.2: Graphic representaion of a problem instance

Given the randomized nature of the function, you will get a different problem instance every time you run this code.

```python
# the following code represents the above test
sample_problem = create_problem_instance(10, 6, 5, 5)

# prints the skills for each worker
for worker_id in sample_problem['worker_skills']:
    print(worker_id, sample_problem['worker_skills'][worker_id])

print()

# prints the required skills that the team has to cover
print('Required Skills:', sample_problem['required_skills'])
```

```
w1 {'s10'}
w2 {'s2', 's8', 's5', 's6'}
w3 {'s7', 's2', 's4', 's5', 's1'}
w4 {'s9', 's4'}
w5 {'s7', 's4'}
w6 {'s7', 's10'}

Required Skills: {'s6', 's8', 's7', 's5', 's9'}
```

The next step is to create a solver which is an optimization algorithm that can determine the smallest possible team of workers that can be used to cover all the required skills.

Decision Making with a Brute-Force Algorithm

The first solver will implement the brute-force approach that relies on exhaustively enumerating and considering all possible teams. This solver will use the "combinations" tools from the "itertools" module to generate all possible teams of a specific size.

The tool is demonstrated via a simple example below:

```python
# used to generate all possible combinations in a given list of elements
from itertools import combinations

L = ['w1', 'w2', 'w3', 'w4']

print('pairs', list(combinations(L, 2))) # all possible pairs
print('triplets', list(combinations(L, 3))) # all possible triplets
```

```
pairs [('w1', 'w2'), ('w1', 'w3'), ('w1', 'w4'), ('w2', 'w3'), ('w2',
'w4'), ('w3', 'w4')]
triplets [('w1', 'w2', 'w3'), ('w1', 'w2', 'w4'), ('w1', 'w3', 'w4'),
('w2', 'w3', 'w4')]
```

The following function can then be created to solve an instance of the team formation problem via the brute-force approach. This brute-force solver considers all possible team sizes and creates all possible teams for each size. It then identifies teams that cover all required skills and reports the smallest one.

```python
def brute_force_solver(problem):

    worker_skills = problem['worker_skills']
    required_skills = problem['required_skills']

    worker_ids = list(worker_skills.keys()) # gets the ids of all the workers
    worker_num = len(worker_ids) # total number of workers
    all_possible_teams = [] # remembers all possible teams
    best_team = None # remembers the best (smallest) team found so far

    #for each possible team size (singles, pairs, triplets, ...)
    for team_size in range(1, worker_num+1):

        # creates all possible teams of this size
        teams = combinations(worker_ids, team_size)
        for team in teams: # for each team of this size

            skill_union = set() # union of skills covered by all members of this team
            for worker_id in team: # for each team member
                # adds their skills to the union
                skill_union.update(worker_skills[worker_id])

            # if all the required skills are included in the union
            if required_skills.issubset(skill_union):

                # if this is the first team that covers all required skills
                # or this team is smaller than the best one or
                if best_team == None or len(team) < len(best_team):
                    best_team = team # makes this team the best one

    return best_team # returns the best solution
```

It is possible for a problem instance to not have a feasible solution. For example, if the set of required skills includes a skill that none of the available workers possesses, then there is no way to create a team that covers all skills. In such cases, the above solver will simply return a None value.

The following code can now be used to test the brute-force solver on the sample problem that was created above.

```python
# uses the brute-force solver to find the best team for the sample problem
best_team = brute_force_solver(sample_problem)
print(best_team)
```

```
('w2', 'w3', 'w4')
```

The brute-force solver is guaranteed to always find the best possible solution (the smallest possible team), as long as a solution exists. However, as discussed in the beginning of this Lesson, its exhaustive nature also leads to an explosion of computational cost as the size of the problem gets bigger.

This can be demonstrated by creating multiple problems with an increasing number of workers. The following code can be used to generate instances of the team formation problem. The number of workers is varied to be equal to 5, 10, 15, and 20. A total of 100 instances are generated for each worker number. All instances include 10 total skills, 8 required skills and at most 5 skills per worker.

```
problems_with_5_workers = [] # 5 workers
problems_with_10_workers = [] # 10 workers
problems_with_15_workers = [] # 15 workers
problems_with_20_workers = [] # 20 workers

for i in range(100): # repeat 100 times

    problems_with_5_workers.append(create_problem_instance(10, 5, 8, 5))
    problems_with_10_workers.append(create_problem_instance(10, 10, 8, 5))
    problems_with_15_workers.append(create_problem_instance(10, 15, 8, 5))
    problems_with_20_workers.append(create_problem_instance(10, 20, 8, 5))
```

The following function accepts a list of problem instances and a solver. It then uses the solver to compute and returns the solution for all instances. It also prints the total time (in seconds) required to compute the solutions, as well as the total number of instances for which a solution could be found.

```
import time

def gets_solutions(problems,solver):

    total_seconds = 0 # total seconds required to solve all problems with this solver
    total_solved = 0 # total number of problems for which the solver found a solution
    solutions = [] # solutions returned by the solver

    for problem in problems:

        start_time = time.time() # starts the timer
        best_team = solver(problem) # computes the solution
        end_time = time.time() # stops the timer
        solutions.append(best_team) # rememberσ the solution
        total_seconds += end_time-start_time # computes total elapsed time

        if best_team != None: # if the best team is a valid team
            total_solved += 1
    print("Solved {} problems in {} seconds".format(total_solved,
                                                     total_seconds))

    return solutions
```

The following code uses this function and the brute-force solver to compute the solutions for 5-worker, 10-worker, 15-worker, and 20-worker datasets that were created above.

```
brute_solutions_5 = gets_solutions(problems_with_5_workers,
           solver = brute_force_solver)

brute_solutions_10 = gets_solutions(problems_with_10_workers,
           solver = brute_force_solver)

brute_solutions_15 = gets_solutions(problems_with_15_workers,
           solver = brute_force_solver)

brute_solutions_20 = gets_solutions(problems_with_20_workers,
           solver = brute_force_solver)
```

```
Solved 23 problems in 0.0019948482513427734 seconds
Solved 80 problems in 0.06984829902648926 seconds
Solved 94 problems in 2.754629373550415 seconds
Solved 99 problems in 109.11902689933777 seconds
```

While the exact numbers printed by the gets_solutions() function will vary due to the randomized nature of the datasets, two patterns will always be consistent:

- Increasing the number of workers leads to a higher number of problem instances for which a solution could be found. This is reasonable, as having more workers increases the probability that the worker pool includes at least one worker that possesses each required skill.
- Increasing the number of workers leads to a significant (exponential) increase in computational time. This was anticipated given the analysis conducted in the beginning of this lesson. For worker populations of size 5, 10, 15, and 20, the number of possible teams is equal to 31, 1023, 32767, and 1048575, respectively.

In general, given N workers, the number of possible teams is equal to 2^N-1. This number becomes far too large to evaluate even for modest values of N. This demonstrates that, even for a simple problem with 1 constraint (covering all required skills) and 1 objective (minimizing the size of the team), brute force is only applicable for very small datasets and it is certainly not practical for any of the complex real-world optimization problems described at the beginning of this lesson.

Decision Making with a Greedy Heuristic Algorithm

The following function addresses this constraint by implementing an optimization algorithm based on a "greedy" heuristic. The heuristic gradually populates a team by adding one member at a time. The next member that is added is always the one that covers the most of the previously uncovered skills. The process continues until all required skills have been covered.

The "greedy heuristic" in this case is the criterion of choosing a worker that covers the most of the previously uncovered skills. A different heuristic function could have been adding first the worker having the largest number of skills.

```
def greedy_solver(problem):

    worker_skills = problem['worker_skills']
    required_skills = problem['required_skills']

    # skills that still have not been covered
    uncovered_required_skills = required_skills.copy()
    best_team = []
    # remembers only the skills of each worker that are required but haven't been covered yet
    uncovered_worker_skills = {}

    for worker_id in worker_skills:

        # remembers only the required uncovered skills that this worker has
        uncovered_worker_skills[worker_id] = worker_skills[worker_id].
intersection(uncovered_required_skills)

    # while there are still required skills to cover
    while len(uncovered_required_skills) > 0:

        best_worker_id = None  # the best worker to add next
        # number of uncovered skills required for the best worker to cover
        best_new_coverage = 0

        for worker_id in uncovered_worker_skills:

            # uncovered required skills that this worker can cover
            my_uncovered_skills = uncovered_worker_skills[worker_id]

            # if this worker can cover more uncovered required skills than the best worker so far
            if  len(my_uncovered_skills) > best_new_coverage:
                best_worker_id=worker_id  # makes this worker the best worker
                best_new_coverage=len(my_uncovered_skills)

        if best_worker_id != None:  # if a best worker was found
            best_team.append(best_worker_id)  # adds the worker to the solution

            #removes the best worker's skills from the skills to be covered
            uncovered_required_skills = uncovered_required_skills -
                                        uncovered_worker_skills[best_worker_id]

            for worker_id in uncovered_worker_skills:

                # remembers only the required uncovered skills that this worker has
                uncovered_worker_skills[worker_id] =
uncovered_worker_skills[worker_id].intersection(uncovered_required_skills)

        else:  # no best worker has been found and some required skills are still uncovered
            return None  # no solution could be found

    return best_team
```

intersection() returns a new set containing only the skills that are common to worker's worker_skills and uncovered_required_skills.

The greedy solver does not consider all possible teams and does not guarantee finding the optimal solution. However, as demonstrated below it is much faster than the brute-force solver and can still produce good (and often optimal) solutions. The method is also guaranteed to find a solution if one exists.

The following code uses the greedy solver to compute the solutions for the same 5-worker, 10-worker, 15-worker, and 20-worker datasets that was used to evaluate the brute-force solver.

```
greedy_solutions_5 = gets_solutions(problems_with_5_workers,
           solver = greedy_solver)

greedy_solutions_10 = gets_solutions(problems_with_10_workers,
           solver = greedy_solver)

greedy_solutions_15 = gets_solutions(problems_with_15_workers,
           solver = greedy_solver)

greedy_solutions_20 = gets_solutions(problems_with_20_workers,
           solver = greedy_solver)
```

```
Solved 23 problems in 0.0009970664978027344 seconds
Solved 80 problems in 0.000997304916381836 seconds
Solved 94 problems in 0.001995086669921875 seconds
Solved 99 problems in 0.0019943714141845703 seconds
```

The difference in speed between the two solvers is evident. In fact, the greedy solver can be applied even on much larger problem instances. For example:

```
# creates 100 problem instances of a team formation problem with 1000 workers
problems_with_1000_workers = []

for i in range(100): # repeats 100 times
    problems_with_1000_workers.append(create_problem_instance(10, 1000, 8, 5))

# solves the 100-worker problems using the greedy solver
greedy_solutions_1000 = gets_solutions(problems_with_1000_workers,
           solver = greedy_solver)
```

```
Solved 100 problems in 0.09574556350708008 seconds
```

Comparing the Algorithms

Having demonstrated the speed advantage of the greedy heuristic, the next step is to also validate the quality of the solutions that it produces. The following function accepts the solutions produced by the greedy and brute-force solvers on the same collection of problem instances. It then reports the percentage of instances for which both solvers report the optimal solution (the smallest possible team).

```
def compare(brute_solutions,greedy_solutions):
    total_solved = 0
    same_size = 0

    for i in range(len(brute_solutions)):

        if brute_solutions[i] != None: # if a solution was found
            total_solved += 1

            # if the solvers reported a solution of the same size
            if len(brute_solutions[i]) == len(greedy_solutions[i]):
                same_size += 1

    return round(same_size / total_solved, 2)
```

The compare() function can now be used to compare the effectiveness of the two solvers applied to the datasets with 5, 10, 15, and 20 workers.

```
print(compare(brute_solutions_5,greedy_solutions_5))
print(compare(brute_solutions_10,greedy_solutions_10))
print(compare(brute_solutions_15,greedy_solutions_15))
print(compare(brute_solutions_20,greedy_solutions_20))
```

```
1.0
0.82
0.88
0.85
```

The results demonstrate that the greedy heuristic can consistently find the optimal solution for about 80%, or higher, of all solvable problem instances. In fact, one can easily verify that, even for cases when it fails to find the optimal solution, the size of the team that it returns is very close to that of the best possible team.

Combined with the overwhelming speed advantage, this makes the heuristic a far more practical choice for realistic applications.

The next lesson will explore even more intelligent optimization techniques and their applications to different problems.

Exercises

1. What are the advantages and disadvantages of the brute-force and greedy algorithms for solving optimization problems?

2. Analyze how greedy heuristic algorithms are used to find optimal solutions in optimization problems.

3 You want to create a greedy solver for the optimization problem of team formation. Complete the following code so that the function utilizes a greedy heuristic for the assignment of team members to a job.

```
def greedy_solver(problem):
    worker_skills=problem['worker_skills'] # worker skills for this problem
    required_skills=problem['required_skills'] # required skills for this problem

    uncovered_required_skills = required_skills.____________() # skills not covered
    best_team=[] # best solution
    uncovered_worker_skills={}
    for worker_id in worker_skills:

        uncovered_worker_skills[worker_id]=worker_skills[worker_id]._____
(uncovered_required_skills)
    while len(uncovered_required_skills) > 0:

        best_worker_id=__________ # the best worker to add next
        best_new_coverage=0 # number of uncovered required skills covered by the best worker
        for worker_id in uncovered_worker_skills: # for each worker
            my_uncovered_skills=uncovered_worker_skills[worker_id]
            # if this worker can cover more uncovered required skills than the best worker so far
            if  len(my_uncovered_skills)>best_new_coverage:
                best_worker_id=worker_id # makes this worker the best worker

                best_new_coverage=____________(my_uncovered_skills)

        if best_worker_id!=_____________: # if a best worker was found

            best_team.___________(best_worker_id) # adds the worker to the solution
            #removes the best worker's skills from the skills to be covered
            uncovered_required_skills=uncovered_required_skills - uncovered_
worker_skills[best_worker_id]
            # for each worker
            for worker_id in uncovered_worker_skills:

                # remembers only the required uncovered skills that this worker has
                uncovered_worker_skills[worker_id]=uncovered_worker_

skills[worker_id].________(uncovered_required_skills)
        else: # no best worker has been found and some required skills are still uncovered
            return _______ # no solution could be found
    return best_team
```

4 List three different real-world optimization problem. For each problem:

- Give an example of an objective function.
- Give two examples of constraints, if any.

5 In the brute-force solver, if you increase the number of workers, how does it affect the problem in terms of number of solutions and computational time?

Resource Scheduling Problem

Link to digital lesson

www.ien.edu.sa

Scheduling Problems

Scheduling problems are common in the optimization field as they require the allocation of limited resources to multiple tasks in a way that optimizes some objective function. Scheduling problems often have additional constraints, such as requiring tasks to happen in a specific order or to be completed by a certain deadline. These problems are essential in various real-world applications, including manufacturing, transportation, healthcare, and project management. In this lesson, you will go deeper into optimization algorithms by introducing additional techniques to solve scheduling problems.

Table 5.1: Real-world applications need scheduling solutions

Application	Function
Project scheduling	Allocate resources and tasks to project activities to minimize project duration and costs.
Production planning	Determine the optimal production plan to meet demand while minimizing inventory and costs.
Airline scheduling	Schedule aircraft departures and crew shifts to optimize flight schedules while minimizing costs and delays.
Call center scheduling	Allocate agents to shifts to ensure adequate coverage for working periods while minimizing costs and meeting service level agreements.
Job shop scheduling	Allocate resources in manufacturing to minimize production time and costs.
Media scheduling	Schedule the timing of advertisements on television or radio to maximize audience reach and revenue, while respecting budget constraints.
Nurse scheduling	Assign nurses to shifts in a hospital to ensure adequate coverage during working periods while minimizing labor costs.

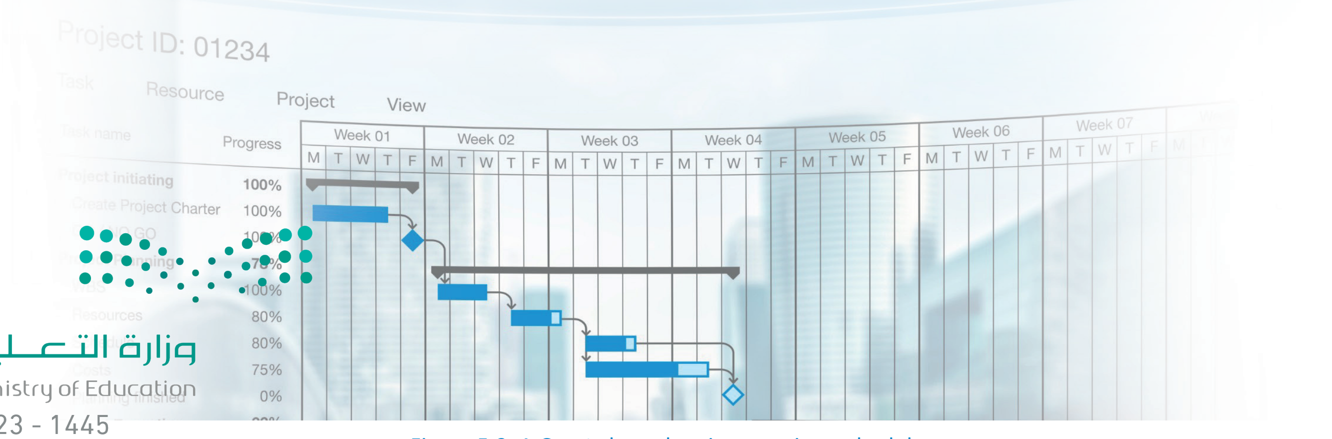


Figure 5.3: A Gantt chart showing a project schedule

In this lesson, the Single-Machine Weighted Tardiness (SMWT) problem will be used as a working example to demonstrate how optimization algorithms can solve scheduling problems.

Single-Machine Weighted Tardiness (SMWT) Problem

To illustrate the problem, consider a factory that needs to schedule the production tasks of several items on a single machine.

- Each job has a specific processing time and a due date by which it needs to be completed.
- Each job is also associated with a weight, that represents the job's importance.

If it is impossible to complete all tasks by the deadline, it will be less expensive to miss the deadline for low-weight tasks than for high-weight tasks.

Goal

The goal is to schedule the jobs in a way that minimizes the weighted sum of the lateness (tardiness) of each job. The weight tardiness sum thus serves as the objective function for optimization algorithms designed to solve this problem.

Lateness calculation

The lateness of a job is calculated as the difference between its completion time and its due time. Job weights are then used as multipliers to complete the final weighted sum. For example, consider a schedule with three jobs J1, J2, and J3 with weights equal to 2, 1, and 2, respectively. According to this schedule, J1 will finish on time, J2 will be 3 hours late, and J3 will finish 1 hours late. This means that the weighted tardiness sum is equal to 3×1+1×2=5.

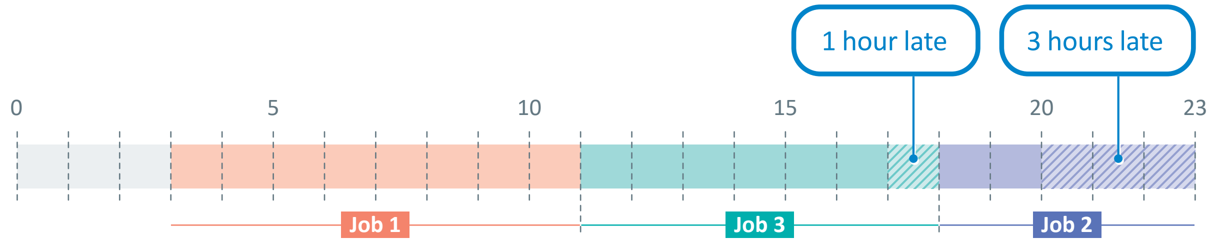


Figure 5.5: Visual representation of the sequence of jobs

Job	Due Time	Completion Time	Lateness	Weighted Tardiness
----------------	--------------------------	--------------------------	------------------	--------------------------
J1	14	11	0	0
J2	20	23	3	3
J3	17	18	1	2

Figure 5.4: Calculate the weighted tardiness

The SMWT problem is challenging to solve as its complexity grows exponentially with the number of jobs. Thus making it computationally expensive, and often impossible, to find the best possible solution for large input sizes.

Optimization algorithms are used to obtain near-optimal solutions for a problem in a reasonable amount of time.

Job Shop Scheduling (JSS) Problem

The Job Shop Scheduling (JSS) problem, is another classic scheduling problem that has been widely studied in the optimization field. The JSS problem involves scheduling a set of jobs on multiple machines, where each job has to be processed in a specific order and time on each machine in relation to the other jobs.

Goal

To minimize the total completion time (makespan) of all jobs.

Variants of the problem

Other variants of this problem introduce multiple additional constraints, such as:

- Each job has a "release" date before which it cannot be started, in addition to each deadline date.
- Some jobs have to be scheduled before others due to precedence constraints between them.
- Each machine has to undergo periodic maintenance according to a strict maintaince schedule. Machines cannot service jobs during maintenance and a job cannot stop once it has started.

Each machine needs to have some downtime after completing a job. The length of the downtime might be fixed or it may vary across machines. It might also depend on the time that it took to complete the previous job.

The above are only a subset of the various complex constraints and problem variants that occur in real-world scheduling problems. Each variant has its own unique characteristics and practical applications, and different optimization algorithms may be more suitable for solving each variant.

Using Python and Optimization to Solve the SMWT Problem

The following code can be used to create randomized instances of the SMWT problem:

```python
import random

# creates an instance of the Single-Machine Weighted Tardiness problem.

def create_problem_instance(job_num, # number of jobs to create
                            duration_range, # job duration range
                            deadline_range, # deadline range
                            weight_range):# importance weight range

    # generates a random duration, deadline, and weight for each job
    durations = [random.randint(*duration_range) for i in range(job_num)]
    deadlines = [random.randint(*deadline_range) for i in range(job_num)]
    weights = [random.randint(*weight_range) for i in range(job_num)]

    # returns the problem instance as a dictionary
    return {'durations':durations,
            'deadlines':deadlines,
            'weights':weights}
```

The random.randint(x,y) function is used to generate a random integer between x and y. A different way to use this function is to provide a list [x,y] or a tuple (x,y). In that case, the * symbol needs to be typed before the list, exactly as done in the function above. For example:

```
for i in range(5):# prints 5 random integers between 1 and 10
    print(random.randint(*[1, 10]))
```

```
6
5
5
10
1
```

The following code uses the create_problem_instance() function to generate a sample problem instance such that:

- The instance includes 10 jobs.
- Each job can last between 5 and 20 time units. We will assume hours as time units for the remainder of this lesson.
- Each job has a deadline that can range between 5 and 50 hours. The deadline clock starts from the moment the first job starts using the machine. For example, if the deadline for a job is equal to 10, then this means that the job has to be completed within 10 hours from the beginning of the job that was scheduled first.
- The weight of each job is an integer between 1 and 3.

number of jobs to create

job duration range

```
create_problem_instance(10, [5, 20], [5, 50], [1, 3])
```

deadline range

importance weight range

```
{'durations': [18, 17, 17, 6, 9, 6, 20, 12, 9, 19],
 'deadlines': [39, 31, 6, 42, 48, 10, 39, 16, 34, 35],
 'weights': [2, 2, 3, 2, 1, 3, 2, 1, 3, 1]}
```

The following function can be used to evaluate the quality of a schedule that has been produced by any algorithm for a specific problem instance. The function accepts a problem instance and a schedule. It then goes over all the jobs in their scheduled order to compute their finish times, as well as the total weighted tardiness of the entire schedule. The latter is computed by computing the tardiness of each job (with respect to its deadline), multiplying it by the job's weight, and adding the result to the sum.

```
# computes the total weighted tardiness of a given schedule for a given problem instance

def compute_schedule_tardiness(problem, schedule):

    # gets the information for this problem
    durations, weights, deadlines=problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(schedule) # gets the number of jobs
    finish_times = [0] * job_num # stores the finish time for each job
    schedule_tardiness = 0 # initializes the weighted tardiness of the overall schedule to 0
    for pos in range(job_num): # goes over the jobs in scheduled order
        job_id=schedule[pos] #  schedule[pos] is the id in the 'pos' position of the schedule

        if pos == 0: # if this is the job that was scheduled first (position 0)

            # the finish time of the job that starts first is equal to its run time
            finish_times[pos] = durations[job_id]

        else: # for all jobs except the one that was scheduled first

            # the finish time is equal to the finish time of the previous time plus the job's run time
            finish_times[pos] = finish_times[pos-1] + durations[job_id]

        # computes the weighted tardiness of this job and adds it to the schedule's overall tardiness
        schedule_tardiness += weights[job_id] * max(finish_times[pos] - 
deadlines[job_id], 0)

    return schedule_tardiness,finish_times
```

The compute_schedule_tardiness() function will be used to evaluate schedules and will serve as a useful tool for all the algorithms that will be presented in this Lesson for the solution of the SMWT problem.

Itertools.Permutations() Function

The brute-force solver will be using the itertools.permutations() function to create all possible schedules (job combinations). It will then be computing the tardiness of each possible schedule and reporting the best one (the one with the lowest total tardiness).

The itertools.permutations() function accepts a single iterable (e.g. list) and creates each possible permutation of input values. The following simple example demonstrates the use of the permutations() function and shows the permutations of all given job ids:

```
job_ids = [0,1,2] # the ids of 3 jobs
for schedule in itertools.permutations(job_ids):
    print(schedule)
```

```
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)
```

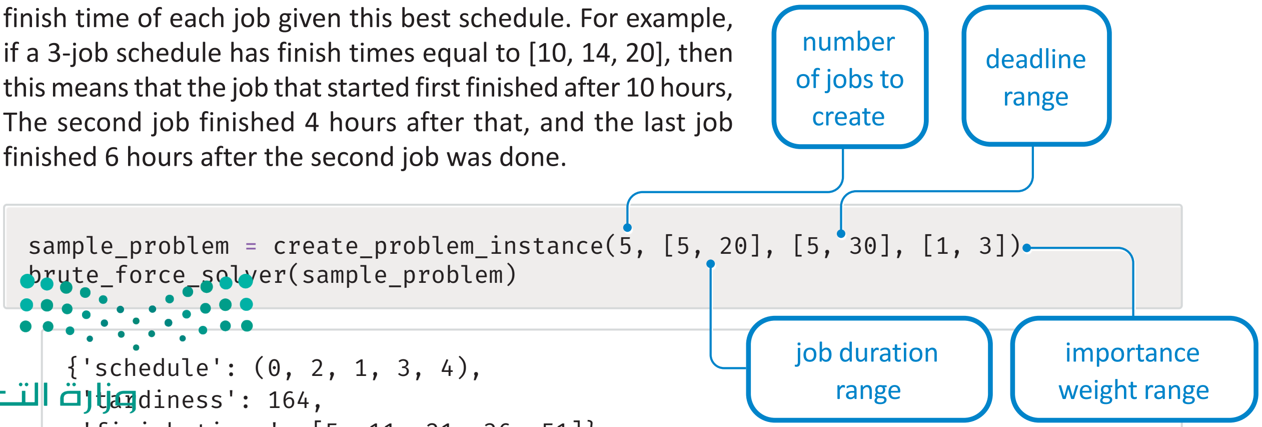
Brute-force solvers are better used for small problems. An instance of the SMWT problem with N jobs has N! possible schedules. For N = 5, this creates only 5! = 120 schedules. However, the number skyrockets for N = 10 to 10! = 3,628,800 and for N = 11 to 11! = 39,916,800.

Brute-Force Solver

In the previous lesson, you learnt how to implement a brute-force solver for the team formation problem. Even though the solver was shown to be too slow for larger problems, its ability to always find the optimal (best possible) solution for small instances was useful for evaluating the quality of the solutions produced by faster optimization algorithms that do not guarantee optimality. Similarly, the following brute-force solver can be used to solve an instance of the SMWT problem.

```python
import itertools

def brute_force_solver(problem):

    # gets the information for this problem
    durations, weights, deadlines=problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(durations) # number of jobs

    # Generates all possible schedules
    all_schedules = itertools.permutations(range(job_num))

    # Initializes the best solution and its total weighted tardiness
    best_schedule = None # initialized to None

    # 'inf' stands for 'infnity'. Python will evaluate all numbers as smaller than this value.
    best_tardiness = float('inf')

    # stores the finish time of each job in the best schedule
    best_finish_times = None # initalized to None

    for schedule in all_schedules: # for every possible schedule

        #evalutes the schedule
        tardiness,finish_times=compute_schedule_tardiness(problem, schedule)

        if tardiness < best_tardiness: # this schedule is better than the best so far
            best_tardiness = tardiness
            best_schedule = schedule
            best_finish_times = finish_times

    # returns the results as a dictionary
    return  {'schedule':best_schedule,
             'tardiness':best_tardiness,
             'finish_times':best_finish_times}
```

The solver returns the best schedule, its tardiness, and the finish time of each job given this best schedule. For example, if a 3-job schedule has finish times equal to [10, 14, 20], then this means that the job that started first finished after 10 hours, The second job finished 4 hours after that, and the last job finished 6 hours after the second job was done.



```python
sample_problem = create_problem_instance(5, [5, 20], [5, 30], [1, 3])
brute_force_solver(sample_problem)
```

```
{'schedule': (0, 2, 1, 3, 4),
 'tardiness': 164,
 'finish_times': [5, 11, 21, 36, 51]}
```

Greedy Heuristic Solver

This greedy solver uses a simple heuristic to sort the jobs and decide the order in which they should be scheduled. It then goes over the jobs in this order to compute the finish time of each job and the total weighted tardiness of the entire schedule. For this particular example, the greedy solver returns exactly the same type of output as the brute-force solver.

The greedy solver accepts two parameters: the problem to be solved and the heuristic function (job-sorting criterion) to be used. This allows the user to implement any heuristic function of their choosing as a Python function and pass it to the greedy solver as a parameter.

The following function implements an optimization algorithm that uses a greedy heuristic function to solve the problem:

```
def greedy_solver(problem, heuristic):

    # gets the information for this problem
    durations, weights, deadlines = problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(durations)# gets the number of jobs

    # Creates a list of job indices sorted by their deadline in non-decreasing order
    schedule = sorted(range(job_num), key = lambda j: heuristic(j, problem))


    # evaluates the schedule
    tardiness, finish_times = compute_schedule_tardiness(problem, schedule)

    # returns the results as a dictionary
    return  {'schedule':schedule,
             'tardiness':tardiness,
             'finish_times':finish_times}
```

In this example, the greedy heuristic function used to select the next job to be scheduled is choosing the job having the closest deadline.

The use of the 'lambda' syntax is used with Python's sorted() function when the goal is to sort a list of elements based on a value that is computed separately for each element.

The following function returns the deadline of a specific job in a given problem instance:

```
# returns the deadline of a given job
def deadline_heuristic(job,problem):

    # accesses the deadlines for this problem and returns the deadline for the job
    return problem['deadlines'][job]
```

Passing this deadline_heuristic() function as a parameter to the greedy solver means that the solver will schedule (sort) the jobs in ascending deadline order. This means that the jobs with the earliest deadlines will be scheduled first.

```
greedy_sol = greedy_solver(sample_problem, deadline_heuristic)
greedy_sol
```

```
{'schedule': [3, 1, 4, 0, 2],
 'tardiness': 124,
 'finish_times': [15, 26, 32, 48, 57]}
```

The following function implements an alternative heuristic that also takes into account the weights of the jobs when deciding their order in the schedule:

```
# returns the weighted deadline of a given job
def weighted_deadline_heuristic(job,problem):

    # accesses the deadlines for this problem and returns the deadline for the job
    return problem['deadlines'][job] / problem['weights'][job]
weighted_greedy_sol=greedy_solver(sample_problem, weighted_deadline_heuristic)
weighted_greedy_sol
```

```
{'schedule': [3, 2, 1, 4, 0],
 'tardiness': 89,
 'finish_times': [15, 24, 35, 41, 57]}
```

Local Search

Even though the greedy solver is much faster than the brute-force approach, it also tends to produce lower quality solutions with a significantly higher tardiness. A way to improve a solution computed by a greedy algorithm or by any other approach is Local Search.

In Local Search, a starting solution is iteratively refined by examining its neighboring solutions, which are obtained by applying small modifications to the current solution. For many optimization problems, a common approach for modifying a solution is by iteratively swapping elements. For instance, in the team-formation problem that was covered in the previous lesson, a local search approach would try to create a better team by swapping team members with workers who are currently not a part of the team.

Local search

Local search is a heuristic optimization method that focuses on exploring the neighborhood of a given solution to improve it.

The greedy heuristic solver constructed the solution step-by-step until eventually a complete and final solution was obtained. On the contrary, local search methods start with a complete solution (that may be of moderate or even bad quality), and work iteratively to improve the quality of the solution. Each step, a small change is made to the current solution, and the quality of the resulting solution (known as the neighbor) is evaluated. If the neighbor solution has better quality, then it replaces the current solution, and the search continues. Otherwise, the neighbor solution is discarded and the process is repeated to generate another neighbor. The search terminates when no neighbor solution can be found having quality better than the current solution. The best solution found is returned.

Local_search_solver() Function

The following local_search_solver() function implements a swap-based local search solver for the SMWT problem. The function accepts four parameters:

- A problem instance.
- A greedy heuristic that will be used by the greedy_solver() function to compute an initial solution.
- A swap_selector function that will be used to select two jobs that will swap their positions in their schedule. For example, if the current solution of a 4-job problem is [0,2,3,1], and the swap selector decided to swap the first and last jobs, the new candidate solution would be [1,2,3,0].
- A max_iterations integer that determines how many swaps should be attempted before the solver returns the best solution found so far.

In each iteration, the solver selects two jobs to swap. It then creates a new schedule that swaps the two jobs but is otherwise identical to the original. If the new schedule has a lower weighted tardiness than best schedule found so far, then it becomes the best in its place. The solver has the exact same output as the greedy and brute-force solvers.

The behavior of local search optimization algorithms is heavily influenced by the strategy that is used to iteratively modify the solution.

```
def local_search_solver(problem, greedy_heuristic, swap_selector, max_
iterations):

    # gets the information for this problem
    durations, weights, deadlines=problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(durations) # gets the number of jobs

    # uses the greedy solver to get a first schedule
    # this schedule will be then iteratively refined through local search
    greedy_sol = greedy_solver(problem, greedy_heuristic) # the best schedule so far

    best_schedule, best_tardiness, best_finish_times = greedy_sol['schedule'],
greedy_sol['tardiness'], greedy_sol['finish_times']

    # local search
    for i in range(max_iterations): # for each of the given iterations

        # chooses which two positions to swap
        pos1, pos2 = swap_selector(best_schedule)

        new_schedule = best_schedule.copy() # create a copy of the schedule

        # swaps jobs at positions pos1 and pos2
        new_schedule[pos1], new_schedule[pos2] = best_schedule[pos2],
                                                 best_schedule[pos1]
        # computes the new tardiness after the swap
        new_tardiness, new_finish_times = compute_schedule_tardiness(problem,
new_schedule)

        # if the new schedule is better than the best one so far
        if new_tardiness < best_tardiness:


            # the new_schedule becomes the best one
            best_schedule = new_schedule
            best_tardiness = new_tardiness
            best_finish_times = new_finish_times

    # returns the best solution
    return {'schedule':best_schedule,
             'tardiness':best_tardiness,
             'finish_times':best_finish_times}
```

The neighbors of a solution in this example are all solutions that are obtained by selecting two jobs within the solution and swapping their positions in their schedule.

The following function implements a random swap by simply selecting two random jobs in the given schedule that should exchange places.

```
def random_swap(schedule):

    job_num = len(schedule) # gets the number of scheduled jobs

    pos1 = random.randint(0, job_num - 1) # samples a random position

    pos2 = pos1
    while pos2 == pos1: # keeps sampling until it finds a position other than pos1
        pos2 = random.randint(0, job_num - 1) # samples another random position

    return pos1, pos2 # returns the two positions that should be swapped
```

The following function then adopts a different strategy by always choosing to swap a random pair of jobs that are adjacent in the schedule. For example, if the current schedule for a 4-job problem instance was [0,3,1,2], then the only candidate swaps would be 0<>3, 3<>1, and 1<>2.

```
def adjacent_swap(schedule):

    job_num = len(schedule) # gets the number of scheduled jobs

    pos1 = random.randint(0, job_num - 2) # samples a random position (excluding the last
one)
    pos2 = pos1 + 1 # gets the position after the sampled one

    return pos1,pos2 # returns the two positions that should be swapped
```

The following code uses both swap strategies with the local search solver to solve the sample problem generated in the beginning of this lesson.

```
print(local_search_solver(sample_problem, weighted_deadline_heuristic, random_
swap, 1000))

print(local_search_solver(sample_problem, weighted_deadline_heuristic,
adjacent_swap, 1000))
```

```
{'schedule': [3, 4, 2, 1, 0], 'tardiness': 83, 'finish_times': [15, 21, 30,
41, 57]}
{'schedule': [3, 4, 2, 1, 0], 'tardiness': 83, 'finish_times': [15, 21, 30,
41, 57]}
```

The results show the best schedule [3, 4, 2, 1, 0] for this example and also the overall tardiness (83) and finish times (job 3 will finish on the 15th unit of time, job 4 on the 21st and so on).

Comparing Solvers

The following code uses the create_problem_instance() function to generate two datasets:

- A dataset of 100 SMWT problem instances with 7 jobs each.
- A dataset of 100 SMWT problem instances with 30 jobs each.

The first dataset will be used to compare the performance of all solvers that were described in this lesson:

1. The brute-force solver.
2. The greedy solver with the deadline heuristic.
3. The greedy solver with the weighted deadline heuristic.
4. The local search solver with random swaps and a greedy solver with the deadline heuristic to find the initial solution.
5. The local search solver with random swaps and a greedy solver with the weighted deadline heuristic.
6. The local search solver with adjacent swaps and a greedy solver with the deadline heuristic.
7. The local search solver with adjacent swaps and a greedy solver with the weighted deadline heuristic.

The second dataset will be used to compare all solvers except for the brute-force one, which is far too slow for 30-job problems.

```
#Dataset 1
problems_7 = []
for i in range(100):
    problems_7.append(create_problem_instance(7, [5, 20], [5, 50], [1, 3]))

#Dataset 2
problems_30 = []
for i in range(100):
    problems_30.append(create_problem_instance(30, [5,20], [5, 50], [1, 3]))
```

Compare() Function

The following compare() function uses all the solvers to solve all problems in the given dataset. It then returns the average tardiness value achieved by each solver over all the problems in the dataset. The function also accepts a boolean use_brute parameter to determine if the brute-force solver should be used or not.

```
from collections import defaultdict
import numpy

def compare(problems,use_brute):
    # comparison on Dataset 1
    # maps each solver to a list of all tardiness values it achieves for the problems in the given dataset
    results = defaultdict(list)
    for problem in problems: # for each problem in this datset

        #uses each of the solvers on this problem
        if use_brute == True:
            results['brute-force'].append(brute_force_solver(problem)
['tardiness'])
        results['greedy-deadline'].append(greedy_solver(problem,deadline_
heuristic)['tardiness'])
        results['greedy-weighted_deadline'].append(greedy_
solver(problem,weighted_deadline_heuristic)['tardiness'])
        results['ls-random-wdeadline'].append(local_search_solver(problem,
weighted_deadline_heuristic, random_swap, 1000)['tardiness'])
        results['ls-random-deadline'].append(local_search_solver(problem,
deadline_heuristic, random_swap, 1000)['tardiness'])
        results['ls-adjacent-wdeadline'].append(local_search_solver(problem,
weighted_deadline_heuristic, adjacent_swap, 1000)['tardiness'])
        results['ls-adjacent-deadline'].append(local_search_solver(problem,
deadline_heuristic, adjacent_swap, 1000)['tardiness'])

    for solver in results: # for each solver
        # prints the solver's mean tardiness values
        print(solver,numpy.mean(results[solver]))
```

The compare() function can now be used with both the problems_7 and problems_30 datasets:

```
compare(problems_7,True)
```

```
brute-force 211.49
greedy-deadline 308.14
greedy-weighted_deadline 255.61
ls-random-wdeadline 212.35
ls-random-deadline 212.43
ls-adjacent-wdeadline 220.62
ls-adjacent-deadline 224.36
```

```
compare(problems_30,False)
```

```
greedy-deadline 10126.18
greedy-weighted_deadline 8527.61
ls-random-wdeadline 6647.73
ls-random-deadline 6650.99
ls-adjacent-wdeadline 6666.47
ls-adjacent-deadline 6664.67
```

Exercises

1. Describe two different strategies (swapping, inversion, shifting etc.) for the local search approach of solving the SMWT problem.

2. How many possible schedules (solutions) are there for an instance of the SMWT problem with 9 jobs?

3 You want to create a brute-force solver for the SMWT problem. Complete the following code so that the function utilizes brute force to find the optimal scheduling permutation.

```
def brute_force_solver(problem):
    # gets the information for this problem
    durations, weights, deadlines=problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(_______________) # number of jobs
    # generates all possible schedules

    all_schedules = itertools._________________(range(job_num))
    # initializes the best solution and its total weighted tardiness

    best_schedule = ____________________ # initialized to None
    # 'inf' stands for 'infnity'. Python will evaluate all numbers as smaller than this value.

    best_tardiness = float('___________________')
    # stores the finish time of each job in the best schedule

    best_finish_times=__________________ # initalized to None

    for schedule in all_schedules: # for every possible schedule
        #evalute the schedule
        tardiness,finish_times=compute_schedule_tardiness(problem, schedule)
        if tardiness<best_tardiness: # this schedule is better than the best so far

            best_tardiness=____________________

            best_schedule=______________

            best_finish_times=_________________

    # return the results as a dictionary
    return {'schedule':best_schedule,
            'tardiness':best_tardiness,
            'finish_times':best_finish_times}
```

4 You want to create a local search solver for the SMWT problem. Complete the following code so that the function utilizes local search to find the optimal scheduling permutation.

```
def local_search_solver(problem, greedy_heuristic, swap_selector, max_
iterations):
    # gets the information for this problem
    durations, weights, deadlines=problem['durations'], problem['weights'],
problem['deadlines']

    job_num = len(_______________)# gets the number of jobs
    # uses the greedy solver to get a first schedule.
    # this schedule will be then iteratively refined through local search

    greedy_sol = ________________(problem, greedy_heuristic) # remembers the best
schedule so far
    best_schedule, best_tardiness, best_finish_times=greedy_
sol['schedule'],greedy_sol['tardiness'],greedy_sol['finish_times']

    # local search

    for i in range(______________): # for each of the given iterations
        # chooses which two positions to swap

        pos1,pos2=_______________(best_schedule)

        new_schedule = best_schedule.________________()# creates a copy of the
schedule
        # swaps jobs at positions pos1 and pos2
        new_schedule[pos1], new_schedule[pos2] = best_schedule[pos2], best_
schedule[pos1]
        # computes the new tardiness after the swap
        new_tardiness, new_finish_times = compute_schedule_tardiness(problem,
new_schedule)
        # if the new schedule is better than the best one so far
        if new_tardiness < best_tardiness:
            # the new_schedule becomes the best one

            best_schedule = __________________

            best_tardiness = _________________

            best_finish_times=________________

    # returns the best solution
    return {'schedule':best_schedule,
            'tardiness':best_tardiness,
            'finish_times':best_finish_times}
```

5 Describe how local search works.

6 Write your observations about the results of the the greedy solvers compared to the local search solvers, in the 30 job problem. Why do you think in the 30 job problem the brute-force solver was not used?

Lesson 3

Route Optimization Problem

Link to digital lesson

www.ien.edu.sa

Mathematical Programming In Optimization Problems

The previous two lessons demonstrated how heuristic algorithms could be used to solve different types of optimization problems. While heuristics can be very fast and often produce good solutions, they do not always guarantee the optimal solution and may not be suitable for all types of problems. In this lesson you will focus on a different optimization approach: mathematical programming.

Mathematical programming

Mathematical programming is a technique to solve optimization problems by formulating them as mathematical models.

Mathematical programming can solve many optimization problems, such as resource allocation, production planning, logistics, and scheduling. The technique has the advantage of providing a guaranteed optimal solution and can handle complex problems with multiple constraints.

A mathematical programming solution starts with formulating the given optimization problem as a mathematical model using variables. These variables represent the values that have to be optimized. They are used to define the objective function and constraints, which together describe the problem and enable the use of mathematical programming algorithms.

Mathematical programming utilizes decision variables which can be controlled and tuned by the decision-maker to find the solution, or state variables which the decision-maker has no control over and are imposed by the external environment. State variables cannot be tuned. The following lists provide examples of decision and state variables for some popular optimization problems:

Table 5.2: Examples of decision and state variables

	Decision Variables	State Variables
Production Planning	The quantity of each product that has to be produced.	The availability of raw materials, production machines' capacity, and production labor availability.
Resource Transportation	The number of goods to be transported from one location to another.	The distance between the locations that must be visited and the capacity of the vehicles.
Job Scheduling	The order and time duration of each job to be performed.	The availability of workers and machines, the deadlines, and the importance weights of each job.
Personel Rostering	The assignment and scheduling of workers to different tasks at different times.	The skills, preferences, and availability of each worker. The skills required to complete each task.

The objective function is formulated as a mathematical expression to be optimized (maximized or minimized) based on the relevant variables. This function represents the goal of the optimization problem, such as maximizing profit or minimizing costs. It is usually defined in terms of the decision variables and sometimes the state variables. Similarly, the constraints can be formulated using variables and mathematical inequalities.

There are several types of mathematical programming, including Linear Programming (LP), Quadratic Programming (QP), and Mixed Integer Programming (MIP). This lesson focuses on MIP, which is used for problems where the decision variables are restricted to integers, such as scheduling or routing problems.

The Knapsack Problem

A simple example of using MIP to formulate the objective function and constraints is the **0/1 Knapsack Problem**. The problem is defined as follows: you are given a knapsack with a maximum weight capacity of C and a set of items I. Each item i has two state variables: a weight w_i and a value v_i. The requirement is to fill the knapsack with the maximum possible value within the knapsack's capacity. A decision variable x_i is also used to keep track of the combinations of items to be packed in a knapsack, where x_i = 1 if the item i is selected to be added to the knapsack and x_i = 0 otherwise. The goal is to select a subset of the items from I such that:

- **Constraint**: The sum of the weights of the selected items is not greater than the maximum capacity C.
- **Objective function**: The sum of the values of the selected items is as high as possible (maximized).

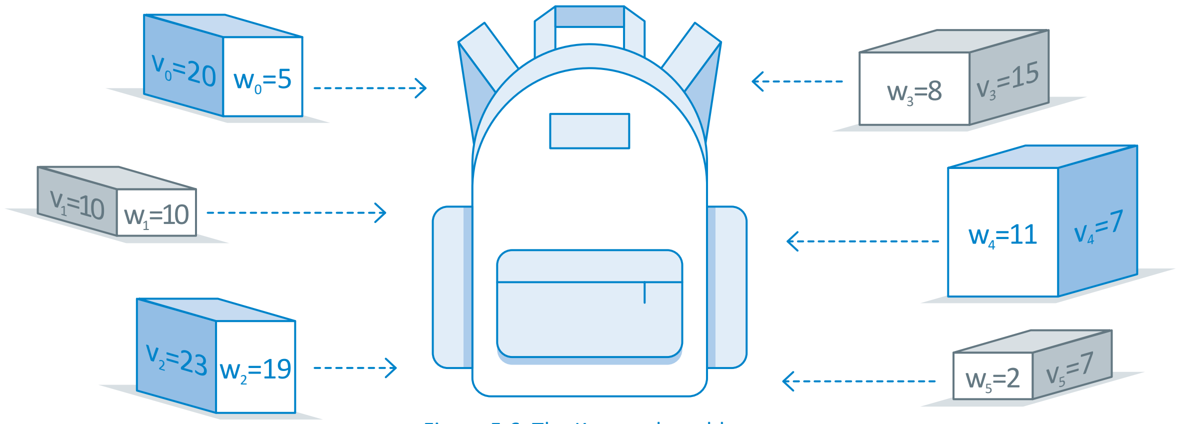


Figure 5.6: The Knapsack problem

A knapsack instance is illustrated in figure 5.6 with six items having specific weights and values and a maximum knapsack capacity of 40. The following code installs and uses the open-source Python library mip (mixed integer programming) to solve an instance of the 0/1 Knapsack problem and imports the necessary modules.

```
!pip install mip # install the mip library
```

```
# imports useful tools from the mip library
from mip import Model, xsum, maximize, BINARY
values = [20, 10, 23, 15, 7, 7] # values of available items
weights = [5, 10, 19, 8, 11, 2] # weights of available items
```

```
C = 40 # knapsack capacity

I = range(len(values)) # creates an index for each item: 0,1,2,3,...

solver = Model("knapsack") # creates a knapsack solver
solver.verbose = 0 # setting this to 1 will print more information on the progress of the solver

x = [] # represents the binary decision variables for each item.

# for each items creates and appends a binary decision variable
for i in I:
    x.append(solver.add_var(var_type = BINARY))

# creates the objective function
solver.objective = maximize(xsum(values[i] * x[i] for i in I))

# adds the capacity constraint to the solver
solver += xsum(weights[i] * x[i] for i in I) <= C

# solves the problem
solver.optimize()
```

```
<OptimizationStatus.OPTIMAL: 0>
```

The code creates a list x to store the binary decision variables for the items. The mip library provides:

- the add_var(var_type=BINARY)) tool for creating binary variables and adding them to the solver.
- the maximize() tool for optimization problems that need to maximize an objective function, whereas the minimize() tool is used for optimization problems that need to minimize an objective.
- the xsum() tool for creating mathematical expressions that include sums. In the above example, the tool is used to compute the total weight of the items in a solution and create the capacity constraint.
- the optimize() tool for finding a solution that optimizes the objective function while respecting the constraints. The tool uses MIP to efficiently consider different combinations of values for the decision variables and find the one that optimizes the objective.
- the += operator to add additional constraints to an existing solver.

In the implementation below, the list x holds one binary variable for each item. After the solution has been computed, each variable will be equal to 1 if the item was included in the solution and equal to 0 otherwise. The mip library uses the x[i].x syntax to return the binary value for the item with index i. The solver computes the decision variable x, then finds the total value and weight of the selected items by iterating over the decision variable x, accumulating the weights and values for each selected item i, based on x[i], and displaying them as shown in the following code.

```
total_weight = 0 # stores the total weight of the items in the solution
total_value = 0 # stores the total value of the items in the solution
```

```
for i in I: # for each item
    if x[i].x == 1: # if the item was selected
        print('item', i, 'was selected')
        # updates the total weight and value of the solution
        total_weight += weights[i]
        total_value += values[i]

print('total weight', total_weight)
print('total value', total_value)
```

```
item 0 was selected
item 2 was selected
item 3 was selected
item 5 was selected
total weight 34
total value 65
```

Traveling Salesman Problem

Another problem that can be solved via MIP is the Traveling Salesman Problem (TSP). It is a classic problem that seeks to determine the shortest possible route a salesman must take to visit a set of cities exactly once and then return to his starting point, without visiting any city twice. The figure 5.7 visualizes an instance of this problem.

TSP graph instances are fully connected; there is an edge connecting every pair of nodes.

Each circle (node) represents a city or location that has to be visited. There is an edge connecting two locations if it is possible to travel between them. The nu mber on the edge represents the cost (distance) between the two locations. In this example, the locations have been numbered according to their order in the optimal solution to the problem. The total cost of the route $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ is 10 + 25 + 30 + 15 = 80, which is the shortest possible route that visits every city exactly once and returns to the starting point. TSP has practical applications in logistics, transportation, supply chain management, and telecommunications. It belongs to a broader family of routing problems that also includes other famous problems which are presented below:

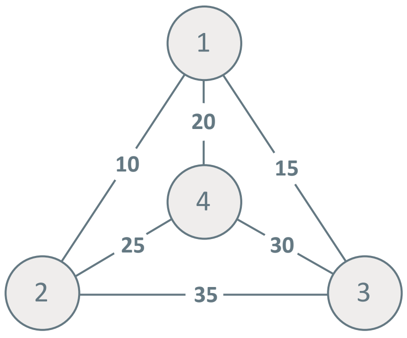


Figure 5.7: Instance of Traveling Salesman problem

- The **Vehicle Routing Problem** involves finding the optimal routes for a fleet of vehicles to deliver goods or services to a set of customers while minimizing the total distance traveled. Applications include logistics, delivery services, and garbage collection.
- The **Pickup and Delivery Problem** involves finding the optimal routes for vehicles to pick up and deliver goods or people to different locations. Applications include taxi services, emergency medical services, and shuttle services.
- The **Train Timetabling Problem** involves finding the optimal train schedules in a railway network while minimizing delay percentage and ensuring efficient use of resources. Applications include railway transportation and scheduling.

The following code can be used to create an instance of the TSP. The function accepts the number of locations to be visited and the distance range (minimum and maximum distance) between two locations. It then returns:

- a distance matrix that includes the distance between every possible pair of locations.
- a set of numeric location ids (one for each location).
- the location that serves as the start and end of the route. This is referred to as the 'startstop' location.

```python
import random
import numpy
from itertools import combinations

def create_problem_instance(num_locations, distance_range):
    # initializes the distance matrix to be full of zeros
    dist_matrix = numpy.zeros((num_locations, num_locations))
    # creates location ids: 0,1,2,3,4,...
    location_ids = set(range(num_locations))
    # creates all possible location pairs
    location_pairs = combinations(location_ids, 2)
    for i,j in location_pairs: # for each pair
        distance = random.randint(*distance_range) # samples a distance within range
        # the distance from i to j is the same as the distance from j to i
        dist_matrix[j,i] = distance
        dist_matrix[i,j] = distance

    # returns the distance matrix, location ids and the startstop vertix
    return dist_matrix, location_ids, random.randint(0, num_locations - 1)
```

The following code used the above function to create a TSP instance with 8 locations and pairwise distances between 5 and 20:

```python
dist_matrix, location_ids, startstop = create_problem_instance(8, (5, 20))
print(dist_matrix)
print(startstop)
```

```
[[ 0. 19. 17. 15. 18. 17.  7. 15.]
 [19.  0. 15. 18. 11.  6. 20.  5.]
 [17. 15.  0. 17. 15.  7.  5. 11.]
 [15. 18. 17.  0. 19.  7.  7. 16.]
 [18. 11. 15. 19.  0. 17. 20. 17.]
 [17.  6.  7.  7. 17.  0. 15. 14.]
 [ 7. 20.  5.  7. 20. 15.  0. 14.]
 [15.  5. 11. 16. 17. 14. 14.  0.]]
3
```

Notice that the diagonal represents the distances from the nodes to themselves (dist_matrix[i,i]), and thus is zero.

Creating a Brute-Force Solver for the Traveling Salesman Problem

The following function uses brute force to exhaustively enumerate all possible routes (permutations) and return the shortest one. It accepts the distance matrix and the startstop location returned by the create_problem_instance() function. Note that a solution to a TSP instance is a permutation of cities, starting and ending at the startstop city.

```python
from itertools import permutations

def brute_force_solver(dist_matrix, location_ids, startstop):
    # excludes the starstop location
    location_ids = location_ids - {startstop}
    # generate all possible routes (location permutations)
    all_routes = permutations(location_ids)
    best_distance = float('inf') # initializes to the highest possible number
    best_route = None # best route so far, initialized to None

    for route in all_routes: # for each route
        distance = 0 # total distance in this route
        curr_loc = startstop # current location

        for next_loc in route:
            distance += dist_matrix[curr_loc,next_loc] # adds the distance of this step
            curr_loc = next_loc # goes to the next location
        distance += dist_matrix[curr_loc,startstop] # goes to the starstop location
        if distance < best_distance: # if this route has lower distance than the best route
            best_distance = distance
            best_route = route

    # adds the startstop location at the beginning and end of the best route and returns
    return [startstop] + list(best_route) + [startstop], best_distance
```

The brute force solver uses the permutations() tool to create all possible routes. Note that the startstop location is excluded from the permutations, as it must always appear at the start and end of each route. For example, if we have 4 locations 0,1,2,3 and 0 is the startstop location, then the list of possible permutations would be:

```python
for route in permutations({1,2,3}):
    print(route)
```

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

The brute force solver computes the total distance of each route and finally returns the one with the shortest distance. The following code applies the solver to the TSP instance generated above.

```
brute_force_solver(dist_matrix, location_ids, startstop)
```

```
([3, 5, 2, 7, 1, 4, 0, 6, 3], 73.0)
```

Similar to the brute-force solvers that were described in the previous lessons, this solver is only applicable to small TSP instances. This is because the number of possible routes is a number that grows exponentially as N gets larger and is equal to $(N-1)!$. For example, for N=15, the number of possible routes is equal to 14! = 87,178,291,200.

Using MIP to Solve the Traveling Salesman Problem

To use MIP to solve the TSP, a mathematical formulation that covers both the objective function and the constraints of the TSP needs to be created.

The formulation requires a binary decision variable x_{ij} for every possible transition $i \rightarrow j$ from a location i to another location j. For a problem with N locations, the number of possible transitions is equal to $N \times (N-1)$. If x_{ij} is equal to 1, the solution includes a transition from location i to location j. Otherwise, if x_{ij} is equal to 0, then this transition is not included in the solution.

Accessing elements in a 2-dimensional numpy array can be easily done via the [i,j] syntax. For example:

```
arr = numpy.full((4,4), 0) # creates a 4x4 array full of zeros

print(arr)

arr[0, 0] = 1
arr[3, 3] = 1

print()
print(arr)
```

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

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

The code also uses the product() tool from 'itertools' to compute all possible location transitions. For example:

```
ids = {0, 1, 2}
for i, j in list(product(ids, ids)):
    print(i, j)
```

```
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
```

The following code uses the Python **mip** library to create an MIP solver. It then adds one binary decision variable for every possible transition in the TSP instance generated above.

```
from itertools import product # used to generate all possible transition
from mip import BINARY
from mip import Model,INTEGER

solver = Model() # creates a solver
solver.verbose = 0 # setting this to 1 will print info on the progress of the solver

# 'product' creates every transition from every location to every other location
transitions = list(product(location_ids, location_ids))

N = len(location_ids) # number of locations

# creates a square numpy array full of 'None' values
x = numpy.full((N, N), None)

# adds binary variables indicating if transition (i->j) is included in the route
for i, j in transitions:
    x[i, j] = solver.add_var(var_type = BINARY)
```

The above code uses the numpy.full() tool to create N×N numpy array for storing the binary x variables.

After adding the x decision variables, the following code can be used to formulate the objective function for the TSP. The function iterates over every possible transition $i \rightarrow j$ and multiplies its distance dist_matrix[i,j] with its decision variable x[i,j]. If the transition is included in the solution, then x[i,j]=1 and dist_matrix[i,j] will be considered. Otherwise, dist_matrix[i,j] will be multiplied by 0 and will be ignored.

```
# the minimize tool is used then the objective function has to be minimized
from mip import xsum, minimize

# objective function: minimizes the distance
solver.objective = minimize(xsum(dist_matrix[i,j]*x[i][j] for i,j in
transitions))
```

The next step is to ensure that the solver only reports solutions that visit each location, except for the startstop, exactly once, as the TSP requires. Visiting each location once means that a valid route can only:

- arrive at each location exactly once.
- depart from each location exactly once.

These arrive/depart constraints can be easily added as follows:

```
# for each location id
for i in location_ids:
    solver += xsum(x[i,j] for j in location_ids - {i}) == 1 # exactly 1 arrival
    solver += xsum(x[j,i] for j in location_ids - {i}) == 1 # exactly 1 departure
```

The complete formulation of the TSP includes one more type of constraint to ensure the computation of connected routes. Consider the TSP instance in figure 5.8. Location 0 is assumed to be the startstop location.

In this instance, the shortest possible route is 0→3→4→1→2, with a total travel distance of 24. However, without a connectivity constraint, a solution with two unconnected routes 0→3→4→0 and 1→2→1 is also valid. This 2-route solution satisfies the arrive/depart constraints defined in the code above, as every location is entered and exited exactly once. However, this is not an acceptable TSP solution.

A solution with a single connected route can be enforced by adding the decision variable y_i for every location i. These variables will capture the order in which each location is visited in the solution.

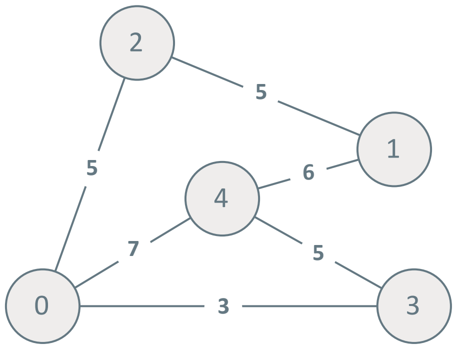


Figure 5.8: TSP instance

```
# adds a decision variable for each location
y = [solver.add_var(var_type = INTEGER) for i in location_ids]
```

For example, if the solution is 0→3→4→1→2→0, then the y values would be y3=0, y4=1, y1=2, y2=3. Location 0 is the startstop location, so its y value is not considered.

These new decision variables can be used to ensure connectivity by adding a new constraint for each transition $i \to j$ that does not include the startstop:

```
# adds a connectivity constraint for every transition that does not include the startstop
for (i, j) in product(location_ids - {startstop}, location_ids - {startstop}):
    # ignores transitions from a location to itself
    if i != j:
        solver += y[j] - y[i] >= (N+1) * x[i, j] - N
```

If a transition $i \to j$ has $x_{ij}=1$ and is included in the solution, then the above inequality becomes y[j] >= y[i] + 1. This states that locations that are visited later are required to have higher y values. Combined with the arrive/depart constraints, a route that does not include the startstop is valid only if:

- If it starts and ends with the same location, to ensure that all locations have exactly one arrival and one departure.
- It assigns higher y values to locations that are visited later since y[j] has to be greater than y[i] for all transitions included in the route. This also avoids adding the same edge from a different direction (e.g., $i \to j$ and $j \to i$).

However, if a location serves as both the start and end of a route, then it needs to have a y value that is both higher and lower than those of all the others in the route. Given that this is impossible, adding the connectivity constraint eliminates any solutions with routes that do not include the startstop.

As an example, consider the 1→2→1 route in the 2-route solution of the TSP instance shown in the figure above. The connectivity constraint requires that $y_2 \geq y_1 + 1$ and $y_1 \geq y_2 + 1$. This is not possible, so the solution would be eliminated.

In constrast, the correct solution 0→3→4→1→2→0 requires that $y_4 \geq {}_{y3} + 1$, $y_1 \geq y_4 + 1$, and $y_2 \geq y_1 + 1$. These can be satisfied by setting y_3=0, y_4=1, y_1=2, y_2=3. Connectivity constraints do not apply to transitions that include the startstop node.

The following function puts everything together to create a complete MIP solver for the TSP:

```python
from itertools import product
from mip import BINARY,INTEGER
from mip import Model
from mip import xsum, minimize

def MIP_solver(dist_matrix, location_ids, startstop):
    solver = Model()# creates a solver
    solver.verbose = 0 # setting this to 1 will print info on the progress of the solver
    # creates every transition from every location to every other location
    transitions = list(product(location_ids,location_ids))
    N = len(location_ids) # number of locations
    # create an empty square matrix full of 'None' values
    x = numpy.full((N, N), None)
    # adds binary decision variables indicating if transition (i->j) is included in the route
    for i, j in transitions:
        x[i, j]=solver.add_var(var_type = BINARY)
    # objective function: minimizes the distance
    solver.objective = minimize(xsum(dist_matrix[i,j]*x[i][j] for i,j in transitions))
    # Arrive/Depart Constraints
    for i in location_ids:
        solver += xsum(x[i,j] for j in location_ids - {i}) == 1 # exactly 1 arrival
        solver += xsum(x[j,i] for j in location_ids - {i}) == 1 # exactly 1 departure
    # adds a binary decision variable for each location
    y = [solver.add_var(var_type=INTEGER) for i in location_ids]
    # adds connectivity constraints for transitions that do not include the startstop
    for (i, j) in product(location_ids - {startstop}, location_ids - {startstop}):
        if i != j: # ignores transitions from a location to itself
            solver += y[j] - y[i] >=(N+1)*x[i,j] - N
    solver.optimize() #solves the problem
    # prints the solution
    if solver.num_solutions: # if a solution was found
        best_route = [startstop] # stores the best route
        curr_loc = startstop # the currently visited location
        while True:
            for next_loc in location_ids:# for every possible next location
                if x[curr_loc,next_loc].x == 1: # if x value for the curr_loc->next_loc transition is 1
                    best_route.append(next_loc) # appends the next location to the route
                    curr_loc=next_loc # visits the next location
                    break
            if next_loc == startstop: # exits if route returns to the startstop
                break
    return best_route, solver.objective_value # returns the route and its total distance
```

The following code generates 100 TSP instances with 8 locations and a distance range between 5 and 20. It also uses the brute-force and the MIP solver to solve each instance and reports the percentage for which the two methods reported routes with the same distance.

```
same_count = 0
for i in range(100):
    dist_matrix, location_ids, startstop=create_problem_instance(8, [5,20])
    route1, dist1 = brute_force_solver(dist_matrix, location_ids, startstop)
    route2, dist2 = MIP_solver(dist_matrix, location_ids, startstop)
    # counts how many times the two solvers produce the same total distance
    if dist1 == dist2:
        same_count += 1
print(same_count / 100)
```

```
1.0
```

The results verify that the MIP solver reports the optimal solution for 100% of the problem instances. The following code also demonstrates the speed of the MIP solver, by using it to solve 100 larger instances with 20 locations.

```
import time

start = time.time() # starts timer
for i in range(100):
    dist_matrix, location_ids, startstop = create_problem_instance(20, [5,20])
    route, dist = MIP_solver(dist_matrix, location_ids, startstop)

stop=time.time() # stops timer
print(stop - start) # prints the elapsed time in seconds
```

```
188.90074133872986
```

Even though the exact execution time will depend on the processing power of the machine that you use to run this Jupyter notebook, it should generally take just a few minutes to compute the solution for all 100 datasets.

This is quite impressive, considering that the number of possible routes for each of the 100 instances translates to 19! = 121,645,100,000,000,000 different routes. Such a large number of routes is far beyond the capabilities of the brute-force approach. However, by efficiently searching this massive space of all possible solutions, the MIP solver can find the optimal route quickly.

Despite its advantages, mathematical programming also has its limitations. It requires a solid understanding of mathematical modeling and may not be suitable for complex problems where the objective function and constraints are hard to express via mathematical formulas. In addition, even though mathematical programming is much faster than the brute-force approach, it might still be too slow for large datasets. In such cases, the heuristic approach demonstrated in the previous two lessons presents a much faster alternative.

Exercises

1 Explain how mathematical programming can be used to solve complex optimization problems.

2 What are the advantages and disadvantages of the MIP approach for solving optimization problems?

3 Analyze two optimization problems that can be solved with mathematical programming and outline their state and decision variables.

4 List three different optimization problems from the family of routing problems.

5 You want to create a brute-force solver function for the Traveling Salesman Problem. Complete the following code so that the solver function returns the best route and best total distance.

```
from itertools import permutations

def brute_force_solver(dist_matrix, location_ids, startstop):
    # excludes the startstop location
    location_ids = ______________ - {_______________}
    # generates all possible routes (location permutations)
    all_routes = ________________(__________________)
    best_distance = float('inf') # initializes to the highest possible number
    best_route = None # best route so far, initialized to None

    for route in all_routes: # for each route
        distance = 0 # total distance in this route
        curr_loc = ______________ # current location

        for next_loc in route:
            distance += _________[curr_loc, next_loc] # adds the distance of this step
            curr_loc = ________________ # goes the next location
        distance += ____________________[curr_loc, _______________] # goes to
back to the startstop location
        if distance < best_distance: # if this route has lower distance than the best route
            best_distance = distance
            best_route = route

    # adds the startstop location at the beginning and end of the best route and returns
    return [startstop] + list(best_route) + [startstop], best_distance
```

6 You want to create an MIP solver for the Traveling Salesman Problem. Complete the following code so that the decision variables and connectivity constraints are selected correctly.

```
def MIP_solver(dist_matrix, location_ids, startstop):

    solver = ______________() # creates a solver
    solver.verbose = 0 # setting this to 1 will print info on the progress of the solver
    # creates every transition from every location to every other location

    transitions = list(_________________(location_ids, location_ids))
    N = len(location_ids) # number of locations
    # creates an empty square matrix full of 'None' values
    x = numpy.full((N, N), None)
    # adds binary decision variables indicating if transition (i->j) is included in the route
    for i, j in transitions:

        x[i, j] = solver._______________(var_type=_________________)

    # objective function: minimizes the distance

    solver.objective = _________________(xsum(dist_matrix[i, j] * x[i][j] for
i, j in transitions))

    # Arrive/Depart Constraints
    for i in location_ids:

        solver += xsum(_________________ for j in location_ids - {i}) == 1

        solver += xsum(_________________ for j in location_ids - {i}) == 1

    # Adds a binary decision variable for each location

    y = [solver._________________(var_type=_________________) for i in
location_ids]

    # Adds connectivity constraints for transitions that do not include the startstop
    for (i, j) in product(location_ids - {startstop}, location_ids -
{startstop}):
        if i != j: # ignores transitions from a location to itself
            solver += y[j] - y[i] >= (N + 1) * x[i, j] - N

    solver._______________() # solves the problem
```

Project

Suppose you work for a delivery company, and your manager has asked you to find the most efficient route to deliver packages to multiple locations in the city.

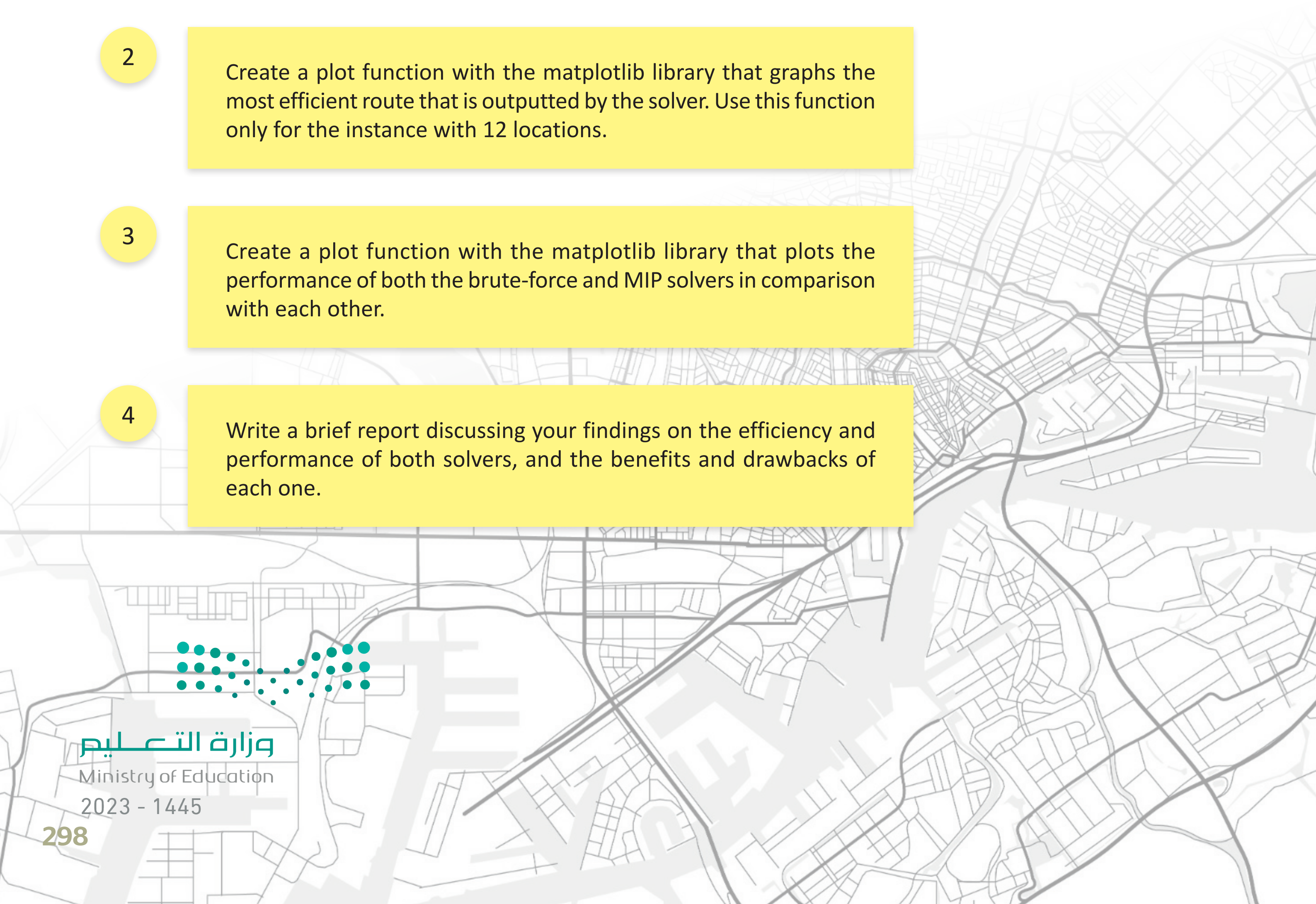
The goal is to find the shortest possible route that visits each location exactly once and returns to the starting location. This problem is an instance of the Traveling Salesman Problem (TSP).

1 You will create various instances of the TSP problem with 3 to 12 locations. Each instance will have a distance range of 5 to 20 units.

2 Create a plot function with the matplotlib library that graphs the most efficient route that is outputted by the solver. Use this function only for the instance with 12 locations.

3 Create a plot function with the matplotlib library that plots the performance of both the brute-force and MIP solvers in comparison with each other.

4 Write a brief report discussing your findings on the efficiency and performance of both solvers, and the benefits and drawbacks of each one.



Wrap up

Now you have learned to:

> Select the appropriate optimization approaches to solve complex problems.

> Solve resource allocation problems by applying Python code.

> Solve scheduling problems by applying Python code.

> Solve the Knapsack problem using different optimization algorithms.

> Solve the Traveling Salesman problem using different optimization algorithms.

KEY TERMS

Brute-Force Solver	Knapsack Problem	Optimization Problem
Constraint Programming	Linear Programming	Quadratic Programming
Greedy Heuristic Algorithm	Local Search	Scheduling Problem
Greedy Solver	Mathematical Programming	Team Formation
Heuristic Algorithm	Integer Programming	Traveling Salesman Problem

6. AI and Society

In this unit, you will analyze how AI ethics influence and guide the development of sophisticated AI systems. You will evaluate how large scale AI systems impact societies and the environment and how they are regulated for ethical and sustainable use. Then, you will use Webots simulator to program a drone for autonomous movement and patrolling of an area with image analysis.

Learning Objectives

In this unit, you will learn to:

> Identify what AI ethics is.
> Interpret how bias and fairness impact the ethical use of AI systems.
> Evaluate how the transparency and explainability problem in AI can be solved.
> Analyze how large scale AI systems influence society and how they are regulated.
> Program a drone device for autonomous movement.
> Develop an image analysis system for the drone used to patrol an area.

Tools

> Webots
> OpenCV library

Lesson 1

Introduction to AI Ethics

Link to digital lesson

www.ien.edu.sa

Overview of AI Ethics

As AI continues to advance, it has become increasingly important to consider the ethical implications of this technology. As a citizen of the modern world, it is important to understand the significance of AI ethics in developing and using responsible AI systems.

One of the main reasons AI ethics is important is that AI systems can potentially affect people's lives significantly. For example, AI algorithms can be used to make hiring and medical treatment decisions. If these algorithms are biased or discriminatory, they can lead to unjust outcomes that harm individuals and communities.

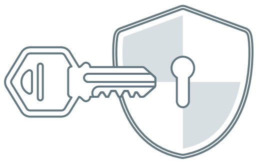
AI Ethics

AI ethics refers to the principles, values, and moral standards that guide AI systems' development, deployment and use.

Real-World Examples of Ethical Concerns in AI

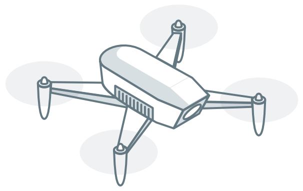
Discriminatory algorithms

There were cases where AI systems have been found to perpetuate biases and discriminate against certain groups of people. For example, a study by the National Institute of Standards and Technology found that facial recognition technology has higher error rates for people with darker skin tones, which can lead to false identifications and wrongful arrests. Another example is the use of AI algorithms in the criminal justice system, where studies have shown that these algorithms can be biased against minorities and lead to harsher sentences.



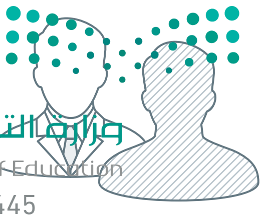
Invasion of privacy

AI systems that collect and analyze data can threaten personal privacy. For example, in 2018, a political consulting firm, had harvested data from millions of Facebook users without their consent and used it to influence political campaigns. This incident raised concerns about using AI and data analytics to manipulate public opinion and infringe on individuals' privacy rights.



Autonomous weapons

The development of autonomous weapons, which can operate without human intervention, has raised ethical concerns about using AI in warfare. Critics argue that these weapons can make life-or-death decisions without human oversight and can be programmed to target specific groups of people, which could violate international humanitarian law and lead to civilian casualties.



Job displacement

The increasing use of AI and automation in various industries has raised concerns about job displacement and the impact on workers' livelihoods. While AI can improve efficiency and productivity, it can also lead to job losses and exacerbate income inequality, which can have negative social and economic consequences.

Bias and Fairness in AI

Bias can occur in AI systems when the data used to train the algorithm is unrepresentative or contains underlying prejudices. Bias in AI systems can occur on any data that the system outputs represent, like products, opinions, communities, and trends, among others.

An example of a biased algorithm is an automated hiring system that uses AI to screen job candidates. Suppose the algorithm is trained on biased data, such as historical hiring patterns that favor certain demographic groups. In that case, it may perpetuate those biases and unfairly screen out qualified candidates from the groups, ignoring categories that are not well represented in the data set. For example, suppose the algorithm favors candidates who attended elite universities or worked at prestigious companies. In that case, it may disadvantage candidates who did not have access to those opportunities or who come from less privileged backgrounds. This can lead to a lack of diversity in the workplace and perpetuate systemic inequalities. Therefore, it is important to develop and use AI hiring algorithms that are based on fair and transparent criteria and do not perpetuate biases.

Fairness in AI refers to how AI systems produce unbiased outcomes and treat all individuals and groups fairly. Achieving AI fairness requires identifying and addressing biases in the data, algorithms, and decision-making processes. For example, one approach to achieving fairness in AI is to use a process called "debiasing," where biased data is identified and removed or modified to ensure that the algorithm produces more accurate and unbiased outcomes.

AI Bias

In the context of AI, bias refers to the tendency of machine learning algorithms to produce outcomes that systematically favor or disfavor certain alternatives or groups, leading to inaccurate predictions and potential discrimination against certain products or populations.

Table 6.1: Factors that contribute to biased AI

Biased training data	AI algorithms learn from the data they are trained on, so if the data is biased or unrepresentative, the algorithm may produce biased outcomes. For example, if an image recognition algorithm is trained on a dataset that predominantly features lighter-skinned individuals, it may have difficulty in recognizing individuals with darker skin tones accurately.
Lack of diversity in the development teams	If the development team is not diverse and does not represent a range of cultural and technical varieties, they may not recognize the biases in the data or the algorithm. A team that only consists of individuals from a particular geographic region or culture leads to a lack of consideration for other regions or cultures that may be represented in the data used to train the AI model.
Lack of oversight and accountability	The lack of oversight and accountability in the development and deployment of AI systems can lead to the perpetuation of biases. Without adequate oversight and accountability mechanisms from companies and governments, testing for bias in AI systems may not be carried out and there may be no recourse for individuals or communities harmed by biased outcomes.
Lack of experience and knowledge in the development team	Development teams lacking experience may not identify or address biases indicators in the training data. A lack of knowledge in designing and testing AI models for fairness may perpetuate existing biases.

Reducing Bias and Promoting Fairness in AI Systems

Diverse and representative data

This involves using data that reflects the diversity of the group it represents. Additionally, it is important to regularly review and update the data used to train AI systems to ensure that it remains relevant and unbiased.

Debiasing techniques

Debiasing techniques involve identifying and removing biased data from AI systems to improve accuracy and fairness. This can include techniques such as oversampling, undersampling, and data augmentation to ensure the AI system is exposed to various data points.

Explainability and transparency

Making AI systems more transparent and explainable can help to reduce bias by allowing users to understand how the system makes decisions. This involves clarifying the decision-making process and allowing users to explore and test the system's outputs.

Human-in-the-loop design

Incorporating human-in-the-loop design into AI systems can help to reduce bias by allowing humans to intervene and correct the system's outputs when necessary. This involves designing AI systems with a feedback loop enabling humans to review and approve the system's decisions.

Ethical principles

Incorporating ethical principles, such as fairness, transparency, and accountability, into the design and implementation of AI systems, ensuring that they are developed and used ethically and responsibly. This involves establishing clear ethical guidelines for using AI systems and regularly reviewing and updating these guidelines as necessary.

Regular monitoring and evaluation

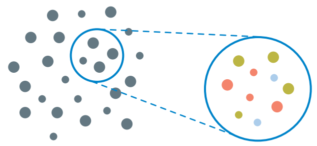
Regularly monitoring and evaluating AI systems is essential for identifying and correcting bias. This involves testing the system's outputs and conducting regular audits to ensure it operates fairly and accurately.

Evaluating user's feedback

User feedback can help identify bias in the system, as users are often more aware of their own experience and can provide better insights into potential bias than AI algorithms can. For example, users can provide feedback on how they perceive the AI system's performance or provide helpful suggestions for ways to improve the system and make it less biased.

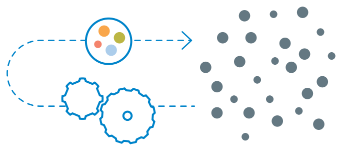
Oversampling

Oversampling in machine learning is increasing a class's samples in a dataset to improve the model's accuracy. It is done by randomly duplicating existing points from the class or generating new points from the same class.



Undersampling

Undersampling is the process of reducing the size of the dataset by deleting a subset of the larger dataset to focus on the more important data points. This is particularly useful if the dataset contains an imbalance of classes or different data groups.



Data Augmentation

Data augmentation is the process of generating new training data from existing data to enhance the performance of machine learning models. Examples include image flipping, rotation, cropping, color changing, affine transformation, and noise addition.

The Problem of Moral Responsibility in AI

The problem of moral responsibility when using advanced AI systems is a complex and multifaceted issue that has attracted significant attention in recent years.

One of the key challenges with advanced AI systems is that they can make decisions and take actions that can have significant positive or negative consequences for individuals and society. However, who should be held morally responsible for these outcomes is not always clear.

One perspective is that the **developers and designers of AI systems** should bear responsibility for any negative outcomes that result from their use. This view emphasizes the importance of ensuring that AI systems are designed with ethical considerations and that developers are held accountable for any harm their creations may cause.

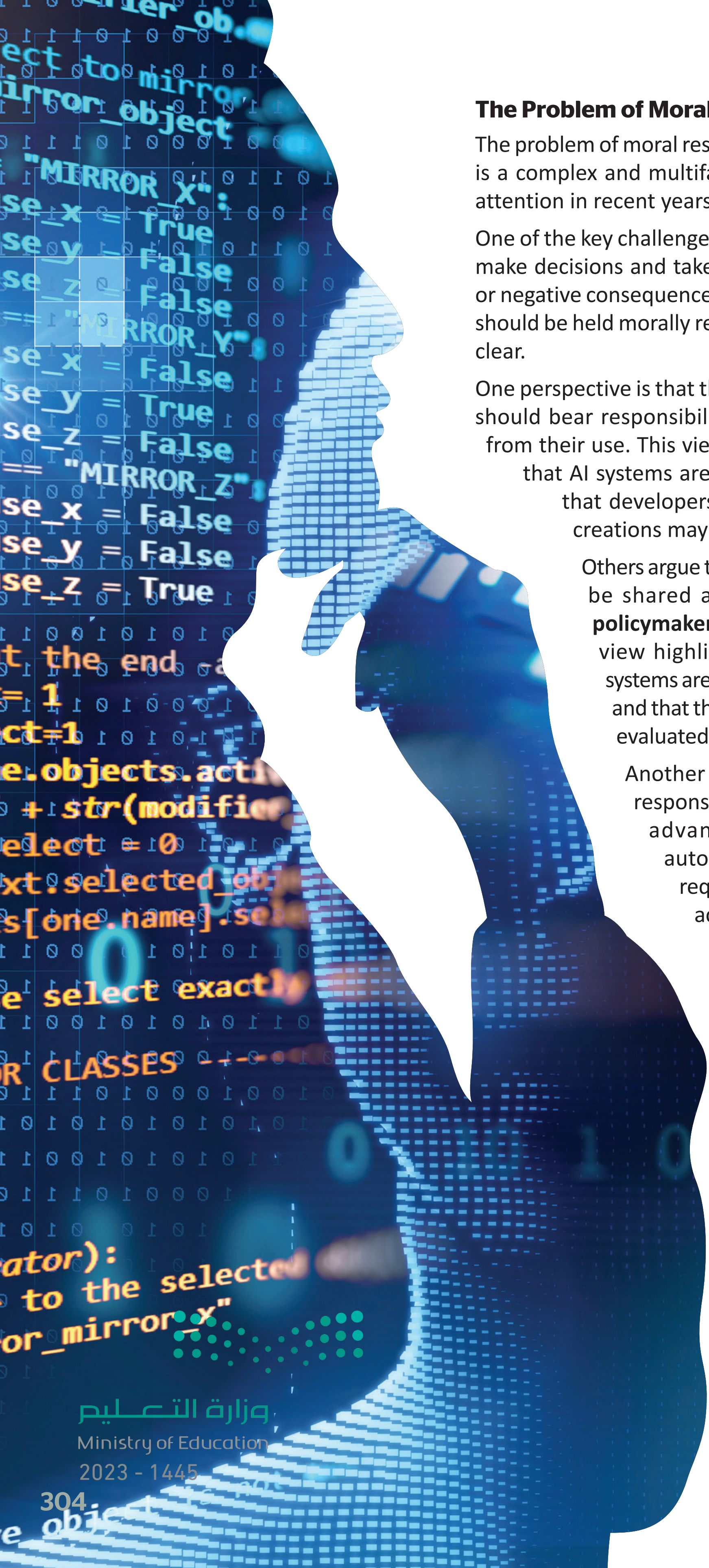
Others argue that the responsibility for AI outcomes should be shared among **broader stakeholders, including policymakers, regulators, and technology users.** This view highlights the importance of ensuring that AI systems are used in ways that align with ethical principles and that the risks associated with their use are carefully evaluated and managed.

Another view is that **AI systems** are moral agents responsible for their actions. This theory holds that advanced AI systems can have agency and autonomy, making them more than tools and requiring them to be accountable for their own acts, but there are various problems with this theory.

AI systems can make judgments and act but are not moral agents for multiple reasons. First, AI systems lack consciousness and subjective experiences, which are essential for moral agency.

Moral agency usually involves reflecting on one's ideals and actions. Second, people train AI systems to follow specified rules and goals, which limits their moral judgment. AI systems can replicate moral decision-making but lack free will and personal autonomy.

Finally, AI system creators and deployers are responsible for their acts. Thus, AI systems can aid ethical decision-making but are not moral agents.



Transparency and Explainability in AI and the Black-Box Problem

The black-box problem in AI is the challenge of understanding how an AI-based system makes decisions or produces outputs. This can make it difficult to trust, explain, or improve the system. Lack of openness and explainability might affect people's trust in the model. Medical diagnosis and autonomous vehicle judgments can be especially challenging. Biases in machine learning models are another black box concern. The biases in the data these models are trained on can lead to unfair or discriminating results.

Additionally, the accountability for decisions made by a black box model can be difficult to determine. It can be challenging to hold anyone responsible for those decisions, particularly with the need for human oversight, such as in the case of autonomous weapons systems. The lack of transparency in AI decision-making makes it challenging to identify and fix problems with the model. It can be difficult to make improvements and ensure it functions correctly without knowing how the model makes its decisions. There are several strategies to addressing the black box problem in AI.

One strategy is to use explainable AI techniques to make machine learning models more transparent and interpretable. This can involve techniques such as natural language explanations or visualizations to aid in understanding the decision-making process. Another approach is to use more interpretable machine learning models, such as decision trees or linear regression. These models may be less complex and easier to understand, but they may not be as powerful or accurate as more complex models. Addressing the black box problem in AI is crucial for building trust in machine learning models and ensuring they are used ethically and fairly.

Black-Box System

A black-box system is one that does not reveal its internal working processes to humans. An input is fed and an output is produced without knowing how it works, as depicted in figure 6.1.

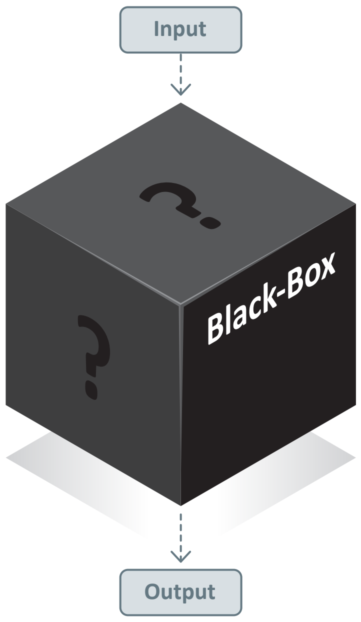
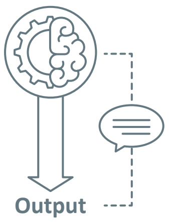


Figure 6.1: Black-Box System

Methods for Enhancing the Transparency and Explainability of AI Models

LIME



LIME (Local Interpretable Model-Agnostic Explanations), which you have used previously for NLP tasks, is a technique that generates local explanations for individual predictions made by a model. LIME creates a simpler, interpretable model approximating the complex black-box model's behavior around a specific prediction. This simpler model is then used to explain how the model arrived at its decision for that particular prediction. The advantage of LIME is that it provides human-readable explanations that non-technical stakeholders can easily understand, even for complex models like deep neural networks.

SHAP

SHAP (SHapley Additive exPlanations) is another method for explaining the output of machine learning models. SHAP is based on the concept of Shapley values from game theory and assigns a value (or weight) for each feature's contribution to the prediction. SHAP can be used with any model, and it provides explanations in the form of feature importance scores, which can help to identify which features are the most influential in the model's output.

Another technique for improving AI explainability such as decision trees and decision rules, which are interpretable models that can be easily visualized. Decision trees partition the feature space based on the most informative feature and provide explicit rules to make decisions. Decision trees are particularly useful when the data is tabular and there are a limited number of features. However, these models are also limited as the interpretability of the generated decision tree decreases with the tree size. For example, it is difficult to understand trees consisting of thousands of nodes and hundreds of levels.

Finally, another approach uses techniques such as sensitivity analysis to help understand how input changes or assumptions can impact the model's output. This approach can be particularly useful in identifying the sources of uncertainty in the model and in understanding the model's limitations.

Value-Based Reasoning in AI Systems

The goal is to create AI systems more aligned with human values and ethics, ensuring that they act in beneficial, fair, and responsible ways.

The first step in value-based reasoning involves **understanding and representing ethical values within AI systems**. These systems must be capable of interpreting and internalizing values or ethical guidelines provided by their human creators or stakeholders. This process may involve learning from examples, human feedback, or explicit rules. By clearly understanding these values, AI systems can better align their actions with the desired ethical principles.

Value-Based Reasoning

Value-based reasoning in AI systems refers to the process by which artificial intelligence agents make decisions or derive conclusions based on a predefined set of values, principles, or ethical considerations.

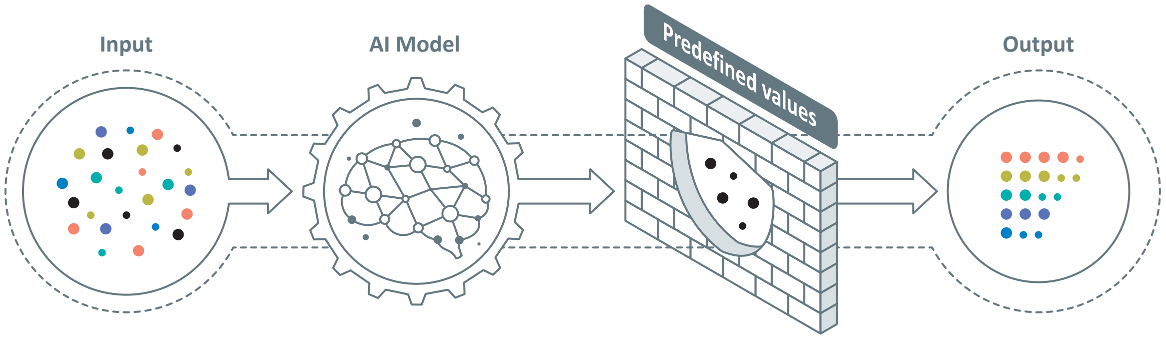


Figure 6.2: Representation of Value-Based Reasoning

The second aspect of value-based reasoning focuses on **evaluating decisions or actions based on internalized values**. AI systems must assess the potential outcomes of different decisions or actions by considering each option's consequences, risks, and benefits. This evaluation process should consider the underlying values the AI system has been designed to uphold, ensuring that it makes informed and value-aligned choices.

Lastly, value-based reasoning requires AI systems to **make decisions that align with established values**. After evaluating various options and their potential outcomes, the AI system should select the decision or action that best reflects the ethical principles and goals it was designed to follow. By making value-aligned decisions, AI agents can act in ways consistent with the ethical guidelines set by their creators, promoting responsible and beneficial behavior. For example, AI systems are being used in healthcare to assist with diagnosis and treatment decisions. These systems must be able to reason about the ethical implications of different treatments, such as the potential side effects or the impact on quality of life, and make decisions prioritizing patient well-being. Another example is AI systems used in finance to assist with investment decisions.

These systems must be able to reason about the ethical implications of different investments, such as the impact on the environment or social welfare, and make decisions that align with the investor's values.

It is important to note that responsibility does not solely rely on the AI system, but rather a collaboration between the AI and human experts. The AI system will assist in decision-making by summarizing the case and presenting the tradeoffs to the user expert, who ultimately takes the final decision. This ensures that the human expert retains control and is accountable for the final outcome, while also benefiting from the insights and analysis provided by the AI system.

AI and Environmental Impact

The impact of AI on the environment and our relation to the environment is complex and multifaceted.

Potential benefits

On the one hand, AI has the potential to help us better understand and address environmental challenges, such as climate change, pollution, and biodiversity loss. AI can help us in analyzing vast amounts of data and predict the impact of different human activities on the environment. It can also help in designing more efficient and sustainable systems, such as energy grids, agriculture, transportation systems, and buildings.

Figure 6.4: AI analyzing large amounts of data

Potential risk or harm

However, there are also concerns about the environmental impact of AI itself. The development and use of AI systems require significant energy and resources, which can contribute to greenhouse gas emissions and other environmental impacts. For example, training a single AI model can require as much energy as several cars use in their lifetimes. Additionally, producing electronic components in AI systems can contribute to environmental pollution, such as using toxic chemicals and generating electronic waste.

Figure 6.3: AI systems require significant energy and resources

Moreover, AI can potentially change our relationship with the environment in ways that are not always positive. For example, using AI in agriculture may lead to more intensive and industrialized farming practices, negatively impacting soil health and biodiversity. Similarly, the use of AI in transportation may lead to more reliance on cars and other modes of transportation, which can contribute to air pollution and habitat destruction.

Conclusion

Overall, the impact of AI on the environment and our relation to the environment depends on how we develop and use AI systems. It is important to consider AI's potential environmental impacts and develop and use AI systems in ways that prioritize sustainability, efficiency, and the planet's health.

Regulatory Frameworks and Industry Standards

Regulatory frameworks and industry standards are critical in promoting ethical AI applications. Regulations and standards can help ensure that organizations developing and using AI systems are accountable for their actions. By setting clear expectations and consequences for non-compliance, regulations, and standards can incentivize organizations to prioritize ethical considerations when developing and using AI systems.

Transparency

Regulations and standards can promote transparency in AI systems by requiring organizations to disclose how their systems work and what data they use. This can help build trust with stakeholders and reduce concerns about potential biases or discrimination in AI systems.

Risk assessment

The risk of unintended consequences or negative outcomes from using AI can also be reduced with appropriate regulations and standards. By requiring organizations to conduct risk assessments. This means identifying potential risks and hazards and implementing appropriate safeguards, regulations and standards can help minimize potential harm to individuals and society.

Clear AI developing and deploying frameworks

Regulations and standards can also encourage innovation by providing a clear framework for developing and using AI systems. Using regulations and standards to establish a level playing field and providing guidance on ethical considerations, can help organizations develop and deploy AI systems in ways that are consistent with ethical and social values.

Regulatory frameworks and industry standards are important in promoting ethical AI applications. By providing clear guidance and incentives for organizations to prioritize ethical considerations, regulations and standards, ensuring that AI systems are developed and used in ways that are aligned with social and ethical values.

Sustainable AI Development in the Kingdom of Saudi Arabia

AI technologies and systems are expected to become a major disruptor in the financial sectors of many countries and may significantly affect the job market. It is predicted that in the coming years, about 70% of the routine work currently performed by workers will be fully automated. The AI industry is expected to create 97 million new jobs and add 16 trillion US dollars to global GDP.

The Saudi Data and Artificial Intelligence Authority (SDAIA) has developed strategic goals for the Kingdom to use sustainable AI technologies for its development. KSA will be a worldwide hub for Data & AI. They also hosted the first Global AI summit in KSA, where global leaders and innovators can discuss and shape AI's future for society's benefit. Another aim is to transform the Kingdom's workforce by developing a local Data & AI talent supply. As AI is transforming labor markets globally, most sectors need to adapt and integrate Data & AI into education, professional training, and public knowledge. By doing so, KSA can gain a competitive advantage in terms of employment, productivity, and innovation.

SDAIA
الهيئة السعودية للبيانات
والذكاء الاصطناعي
Saudi Data & AI Authority

The final goal is to attract companies and investors through flexible and stable regulatory frameworks and incentives. Regulations will focus on developing policies and standards for AI, including ethical use. The framework will promote and support ethical development of AI research and solutions while providing data protection and privacy standards guidelines. This will provide stability and direction for stakeholders operating in the Kingdom.

Example

The Kingdom of Saudi Arabia plans to use AI systems and technologies as the base of its NEOM and THE LINE megacity projects. The NEOM project is a futuristic city that will be powered by clean energy, have advanced transportation systems, and provide high-tech services. It will be a platform for cutting-edge technologies, including AI, and will use smart city solutions to optimize energy consumption, traffic management, and other urban services. AI systems will be used to enhance the quality of life for residents and to improve sustainability.

Similarly, THE LINE will be a linear, zero-carbon city built with AI technologies. THE LINE will use AI systems to automate its infrastructure and transportation systems, creating a seamless, efficient experience for residents. The city will be powered by clean energy and will prioritize sustainable living. AI-powered systems will be used to monitor and optimize energy usage, traffic flow, and other urban services. Overall, AI systems and technologies will play a crucial role in developing these megacity projects, enabling them to become sustainable, efficient, and innovative cities of the future.

NEOM

International AI Ethics Guidelines

As illustrated in the table below, UNESCO has developed a guideline document detailing the values and principles with which new AI systems and technologies should be developed and maintained.

Table 6.2: Values and principles of AI ethics

Values	Principles
• Respect, protection and promotion of human dignity, human rights and fundamental freedoms • Environment and ecosystem flourishing • Ensuring diversity and inclusiveness • Living in harmony and peace	• Proportionality and doing no harm • Safety and security • Fairness and non-discrimination • Sustainability • Privacy • Human oversight and determination • Transparency and explainability • Responsibility and accountability • Awareness and literacy • Multi-stakeholder and adaptive governance and collaboration

Exercises

1

Read the sentences and tick ✓ True or False.	True	False
1. AI ethics is only concerned with the development of AI systems.	○	○
2. AI and automation have the potential to lead to job displacement.	○	○
3. A lack of diversity in AI development teams can lead to biases being overlooked or unaddressed.	○	○
4. Incorporating ethical principles into AI systems can help ensure their responsible development and use.	○	○
5. Human-in-the-loop design requires that AI systems work without any human intervention.	○	○
6. The black box problem in AI refers to the difficulty in understanding how AI algorithms arrive at their decisions or predictions.	○	○
7. AI models can be designed to adapt their decisions or outcomes according to established ethical values.	○	○
8. The widespread use of AI only has positive implications on the environment.	○	○

2 Describe how AI and automation might lead to job displacement.

3 Outline how biased training data can contribute to biased AI outcomes.

4 Define the black-box problem in AI systems.

5 Compare how AI systems can have both positive and negative impact on the environment.

Lesson 2

Applications of Robotics I

Link to digital lesson

www.ien.edu.sa

Revolutionizing the World with Robotics

Robotics is a rapidly growing field that is revolutionizing the way people work, live, and interact with their environment. It has a wide range of applications, from industrial manufacturing to space exploration, medical procedures to home cleaning, and entertainment to military missions.

A key advantage of robotics is their ability to perform repetitive tasks with a high degree of accuracy and precision. Robots can work tirelessly and without error, making them ideal for tasks that are too dangerous or difficult for humans to perform. For example, in the manufacturing industry, robots are used to perform tasks such as welding, painting, and assembling products. In the medical field, robots are used to perform surgeries with greater precision, and in space exploration, robots are used to explore and study distant planets.

Robotics

Robotics is the study of robots, which are machines that can perform a variety of tasks autonomously, semi-autonomously, or under human control.

Robotics and Simulators

Two significant challenges in robotics include the cost and time required to build and test physical robots; this is where simulators come in. Simulators are widely used in robotics research, education, and industry, as they provide a cost-effective and safe way to test and experiment with robots.

Simulators allow developers to create virtual environments that mimic real-world scenarios, allowing them to test their robots' abilities and performance in a variety of situations. They can simulate different weather conditions, terrains, and obstacles that robots may encounter in the real world. Simulators can also simulate the interactions between multiple robots and between robots and humans, allowing developers to study and refine the ways in which robots interact with their environment.

Simulator

Software that allows developers to test and refine their robot designs and algorithms in a virtual world before building physical robots.

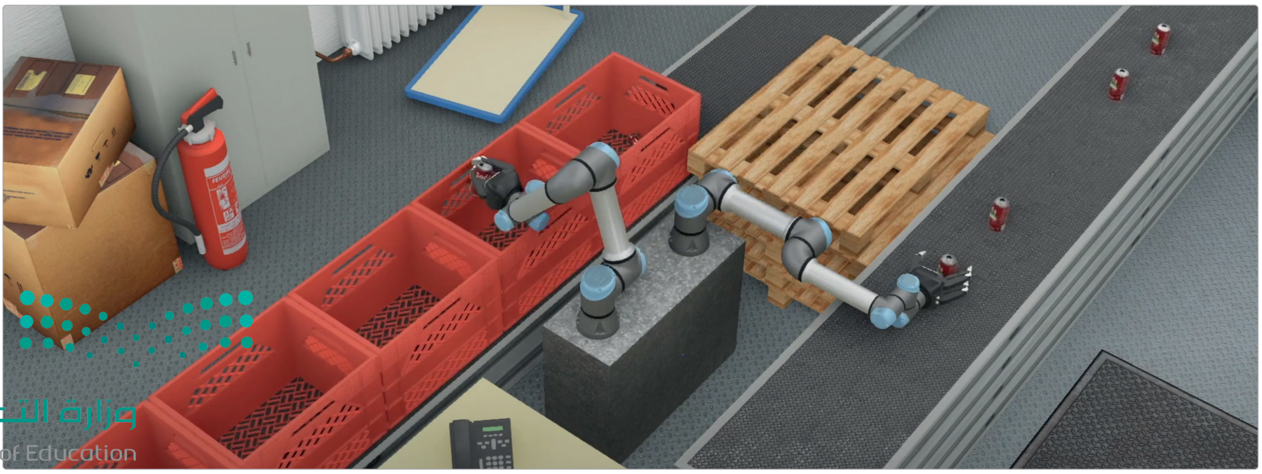


Figure 6.5: Simulation of industrial arms

Another advantage of simulators is that they allow developers to easily modify and test different robot designs and algorithms without the need for expensive hardware. This allows for faster iteration and experimentation, leading to faster development cycles and more efficient designs.

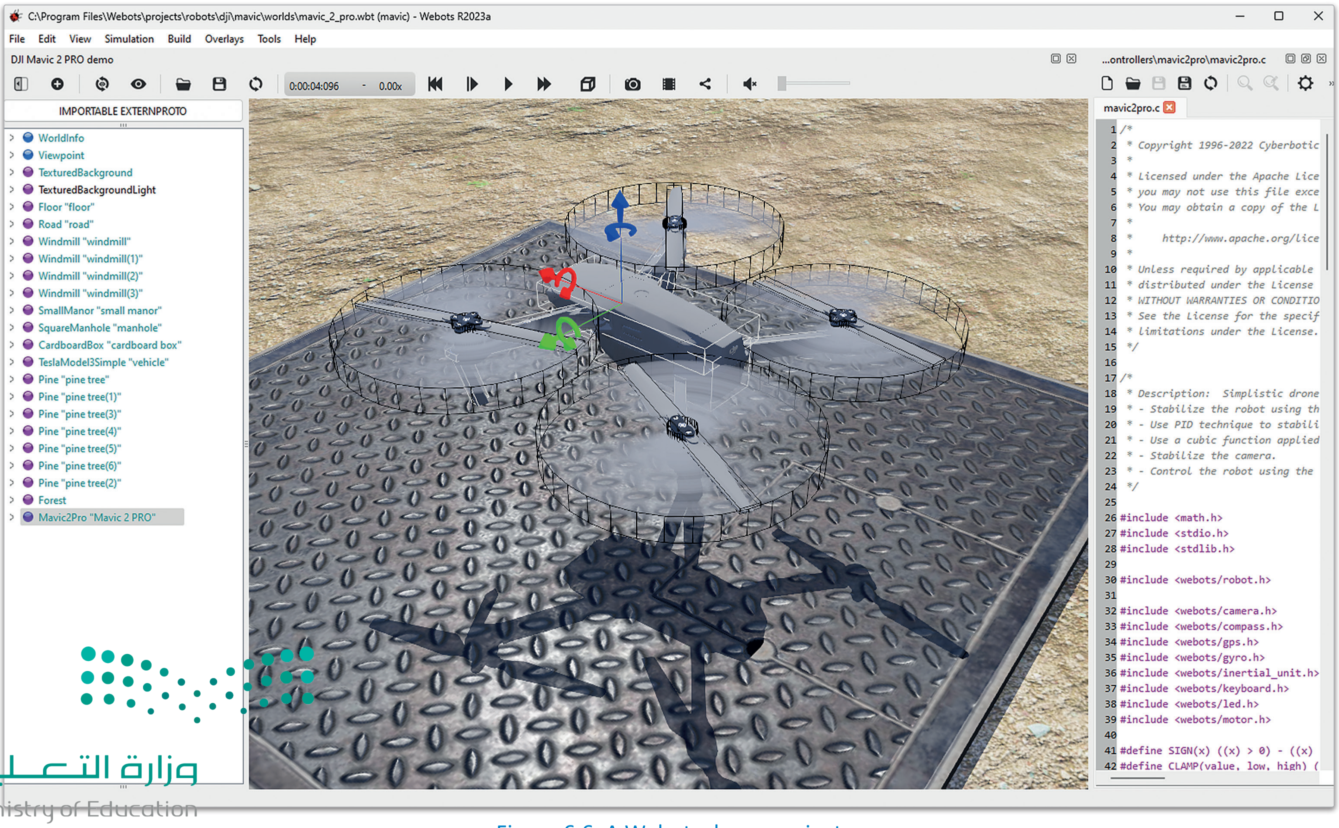
In general, robotics is a rapidly growing field with a wide range of applications and simulators which play a crucial role in robotics development by allowing developers to test and refine their robot designs and algorithms in a safe, cost-effective, and efficient way. As technology continues to advance, the applications of robotics and the use of simulators are only expected to grow, paving the way for a more automated and interconnected world.

Webots

Webots is a powerful software tool that can be used to simulate robots and their environments and an excellent platform for introduction to the world of robotics and artificial intelligence AI. With Webots, students can design, simulate, and test their own robotic systems and algorithms without the need for expensive hardware.

Using Webots in AI is particularly useful because it allows students to experiment with machine learning algorithms and test their performance in a simulated environment. By creating virtual robots and environments, students can explore the capabilities and limitations of AI, and learn how to program intelligent systems that can make decisions based on real-time data.

You can download Webots from this link:
https://github.com/cyberbotics/webots/releases/download/R2023a/webots-R2023a_setup.exe


C:\Program Files\Webots\projects\robots\dji\mavic\worlds\mavic_2_pro.wbt (mavic) - Webots R2023a
File
Edit
View
Simulation
Build
Overlays
Tools
Help
DJI Mavic 2 PRO demo
0:00:04:096
0.00x
IMPORTABLE EXTERNPROTO
...ontrollers\mavic2pro\mavic2pro.c
mavic2pro.c

WorldInfo
Viewpoint
TexturedBackground
TexturedBackgroundLight
Floor "floor"
Road "road"
Windmill "windmill"
Windmill "windmill(1)"
Windmill "windmill(2)"
Windmill "windmill(3)"
SmallManor "small manor"
SquareManhole "manhole"
CardboardBox "cardboard box"
TeslaModel3Simple "vehicle"
Pine "pine tree"
Pine "pine tree(1)"
Pine "pine tree(3)"
Pine "pine tree(4)"
Pine "pine tree(5)"
Pine "pine tree(6)"
Pine "pine tree(2)"
Forest
Mavic2Pro "Mavic 2 PRO"

```
/*
 * Copyright 1996-2022 Cyberbotic
 *
 * Licensed under the Apache Lice
 * you may not use this file exce
 * You may obtain a copy of the L
 *
 *     http://www.apache.org/lice
 *
 * Unless required by applicable
 * distributed under the License
 * WITHOUT WARRANTIES OR CONDITIO
 * See the License for the specif
 * limitations under the License.
 */

/*
 * Description:  Simplistic drone
 * - Stabilize the robot using th
 * - Use PID technique to stabili
 * - Use a cubic function applied
 * - Stabilize the camera.
 * - Control the robot using the
 */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include <webots/robot.h>

#include <webots/camera.h>
#include <webots/compass.h>
#include <webots/gps.h>
#include <webots/gyro.h>
#include <webots/inertial_unit.h>
#include <webots/keyboard.h>
#include <webots/led.h>
#include <webots/motor.h>

#define SIGN(x) ((x) > 0) - ((x)
#define CLAMP(value, low, high) (
```

Figure 6.6: A Webots drone project

Area Surveillance

During this lesson and the next one, you will use Webots to run a simulation of a drone patrolling over a house and you will upgrade it to detect human silhouettes to act as surveillance. The simulation consists of a drone taking off from rest on the ground and commencing a patrol around the house. In the next lesson, you will also be adding computer vision capabilities to the drone using its camera with the **OpenCV** library. This will make it possible to analyze images taken by the camera.

The drone is controlled through a Python script; it is responsible for controlling all of the drone's devices, including the motors of the propellers, camera, GPS (Global Positioning System), etc. It also contains the code to synchronize all the motors to move the drone to various waypoints and stabilize it in the air.

Waypoint

Specific geographical location in 3D space that a drone is programmed to fly to and pass through. They are used to create predefined flight paths for the drone to follow and can be set using GPS coordinates or other location-based systems.

Starting with Webots

In this lesson, you will be introduced to Webots in order to become familiar with its environment. Webots simulation consists of two components:

- The definition of one or more **robots** and their environments in a Webots **world** file.
- One or more **controller** programs for the mentioned **robots**.

A Webots **world** is a 3D description of a robot's attributes. Every object is defined, including its location, orientation, geometry, appearance (such as color or brightness), physical characteristics, type, and more. Objects can contain other objects in the hierarchical systems that make up worlds. A robot might, for instance, have two wheels, a distance sensor, a joint that houses a camera, etc. A world file just specifies the name of the controller that is necessary for each robot; it does not contain the controller code for the robots. Worlds are saved in ".wbt" files. Each Webots project has a "worlds" subdirectory where the ".wbt" files are stored.

A Webots **controller** is a computer program that controls a robot specified in a world file. Any of the programming languages that Webots supports for controller development, such as C++ and Java, can be used, however for this project, you will use Python (.py). Webots launches each of the given controllers as a separate process when a simulation begins, and it associates the controller processes with the simulated robots. Although several robots can share the same controller code, each robot will run its own process. Each controller's source and binary files are stored together in a controller directory. Each Webots project contains a controller directory under the "controllers" subdirectory.

The Webots Environment

When you open the program you will notice several fields and windows. The key components of the Webots interface include:

Menu bar: Located at the top of the interface, the menu bar provides access to various commands and options for working with the simulation, such as creating or importing a robot model, configuring the simulation environment, and running simulations.

Toolbar: The toolbar is a collection of buttons located under the menu bar that provides quick access to frequently used functions, such as adding objects to the scene, starting and stopping the simulation, and changing the camera view.

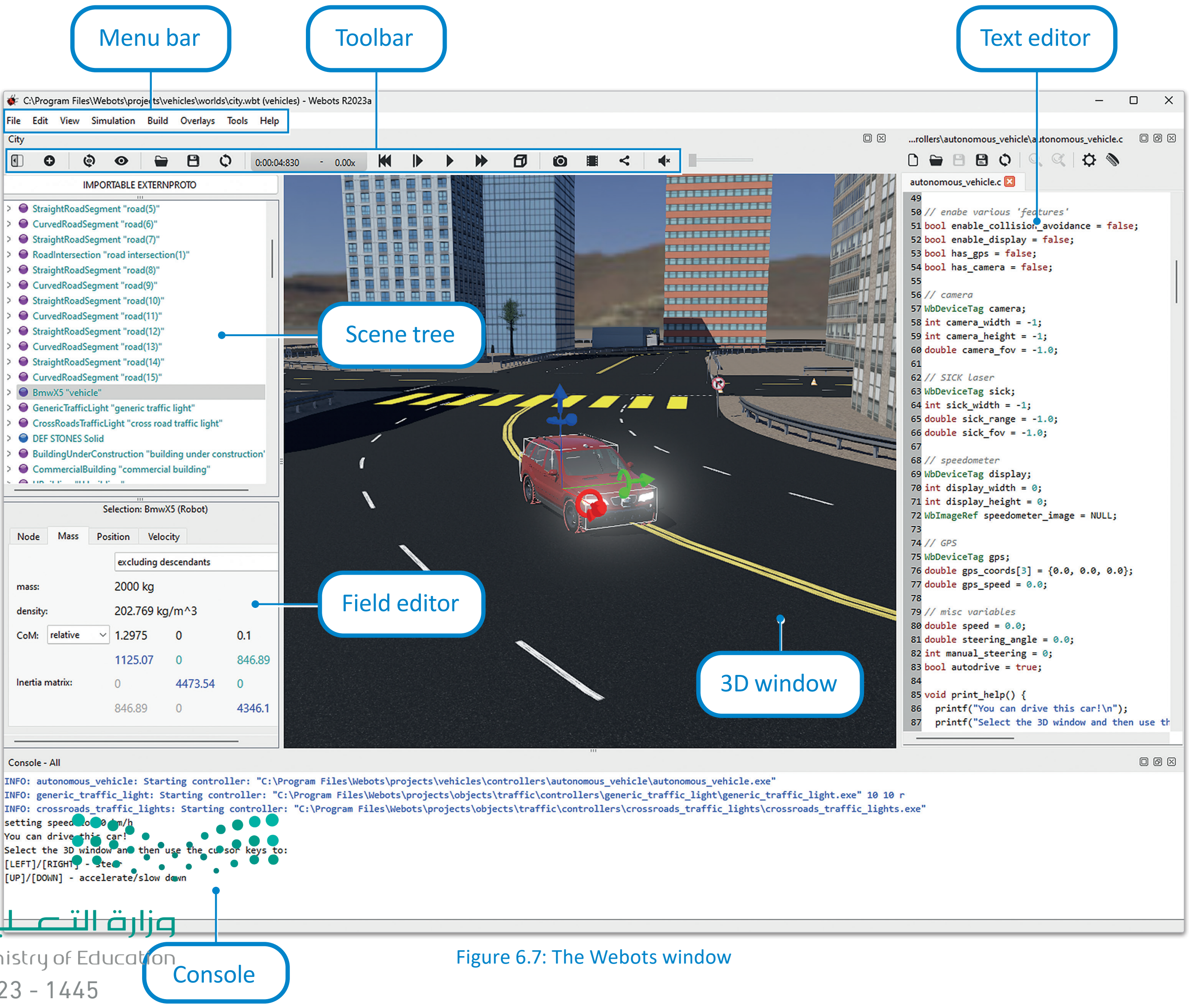
Scene tree: The scene tree is a hierarchical representation of the objects in the simulated environment. It allows users to easily navigate and manipulate the scene, such as adding or deleting objects, changing object properties, and grouping objects for easier management.

Field editor: The field editor is a graphical interface for editing the properties of objects in the simulated environment. Users can use it to adjust object parameters such as position, orientation, size, material, and physical properties.

3D window: The 3D window is the main view of the simulated environment, showing the objects and their interactions in a 3D space. Users can navigate the 3D window using various camera controls, such as pan, zoom, and rotate.

Text editor: The text editor is a tool for editing source code or other text-based files used in the simulation. It provides syntax highlighting and other helpful features for writing and debugging code, such as auto-completion and error highlighting.

Console: The console is a window that displays text-based output from the simulation, including error messages and debugging information. It is useful for troubleshooting problems that may arise during the simulation.


Menu bar
Toolbar
Text editor
Scene tree
Field editor
3D window
Console
C:\Program Files\Webots\projects\vehicles\worlds\city.wbt (vehicles) - Webots R2023a
File Edit View Simulation Build Overlays Tools Help
City
0:00:04:830 - 0.00x
IMPORTABLE EXTERNPROTO
...rollers\autonomous_vehicle\autonomous_vehicle.c
autonomous_vehicle.c
Console - All

StraightRoadSegment "road(5)"
CurvedRoadSegment "road(6)"
StraightRoadSegment "road(7)"
RoadIntersection "road intersection(1)"
StraightRoadSegment "road(8)"
CurvedRoadSegment "road(9)"
StraightRoadSegment "road(10)"
CurvedRoadSegment "road(11)"
StraightRoadSegment "road(12)"
CurvedRoadSegment "road(13)"
StraightRoadSegment "road(14)"
CurvedRoadSegment "road(15)"
BmwX5 "vehicle"
GenericTrafficLight "generic traffic light"
CrossRoadsTrafficLight "cross road traffic light"
DEF STONES Solid
BuildingUnderConstruction "building under construction"
CommercialBuilding "commercial building"

Selection: BmwX5 (Robot)

Node | Mass | Position | Velocity

excluding descendants

mass: 2000 kg

density: 202.769 kg/m^3

CoM: relative 1.2975 0 0.1

Inertia matrix:
1125.07 0 846.89
0 4473.54 0
846.89 0 4346.1

```
// enabe various 'features'
bool enable_collision_avoidance = false;
bool enable_display = false;
bool has_gps = false;
bool has_camera = false;

// camera
WbDeviceTag camera;
int camera_width = -1;
int camera_height = -1;
double camera_fov = -1.0;

// SICK laser
WbDeviceTag sick;
int sick_width = -1;
double sick_range = -1.0;
double sick_fov = -1.0;

// speedometer
WbDeviceTag display;
int display_width = 0;
int display_height = 0;
WbImageRef speedometer_image = NULL;

// GPS
WbDeviceTag gps;
double gps_coords[3] = {0.0, 0.0, 0.0};
double gps_speed = 0.0;

// misc variables
double speed = 0.0;
double steering_angle = 0.0;
int manual_steering = 0;
bool autodrive = true;

void print_help() {
  printf("You can drive this car!\n");
  printf("Select the 3D window and then use th
```

```
INFO: autonomous_vehicle: Starting controller: "C:\Program Files\Webots\projects\vehicles\controllers\autonomous_vehicle\autonomous_vehicle.exe"
INFO: generic_traffic_light: Starting controller: "C:\Program Files\Webots\projects\objects\traffic\controllers\generic_traffic_light\generic_traffic_light.exe" 10 10 r
INFO: crossroads_traffic_lights: Starting controller: "C:\Program Files\Webots\projects\objects\traffic\controllers\crossroads_traffic_lights\crossroads_traffic_lights.exe"
setting speed to [illegible]0 km/h
You can drive this car!
Select the 3D window and then use the cursor keys to:
[LEFT]/[RIGHT] - steer
[UP]/[DOWN] - accelerate/slow down
```

Figure 6.7: The Webots window

First, you have to install the necessary libraries you will use in your project. To install the OpenCV library via PyCharm:

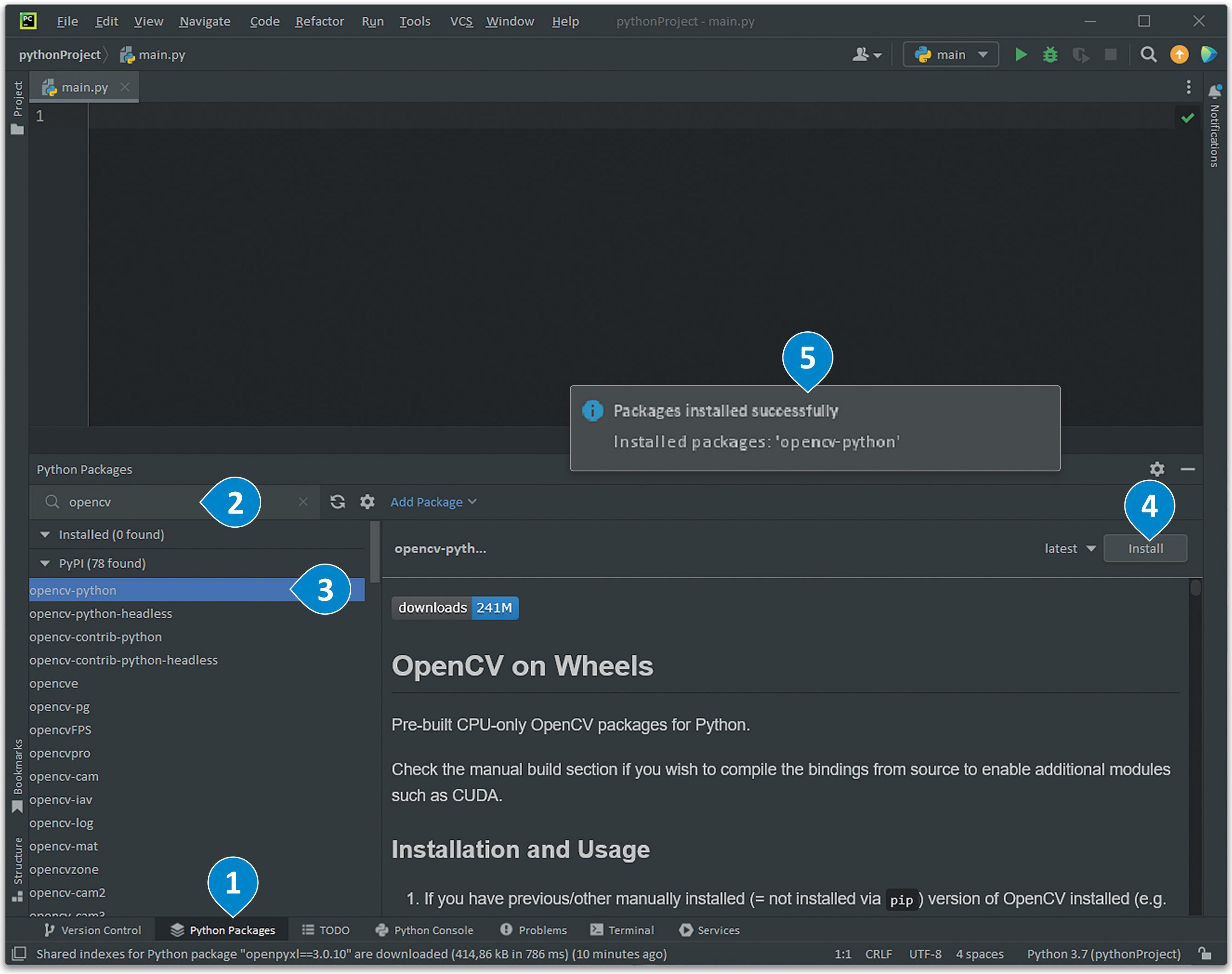
To install OpenCV:

> On PyCharm window, click on **Packages**. 1

> Type **opencv** to the search bar. 2

> Select **opencv-python** 3 and click **Install**. 4

> A message will inform you that the installation is done. 5



Packages installed successfully
Installed packages: 'opencv-python'

OpenCV on Wheels

Pre-built CPU-only OpenCV packages for Python.

Check the manual build section if you wish to compile the bindings from source to enable additional modules such as CUDA.

Installation and Usage

1. If you have previous/other manually installed (= not installed via `pip`) version of OpenCV installed (e.g.

Shared indexes for Python package "openpyxl==3.0.10" are downloaded (414,86 kB in 786 ms) (10 minutes ago)   1:1 CRLF UTF-8 4 spaces Python 3.7 (pythonProject)

Figure 6.8: Installing OpenCV

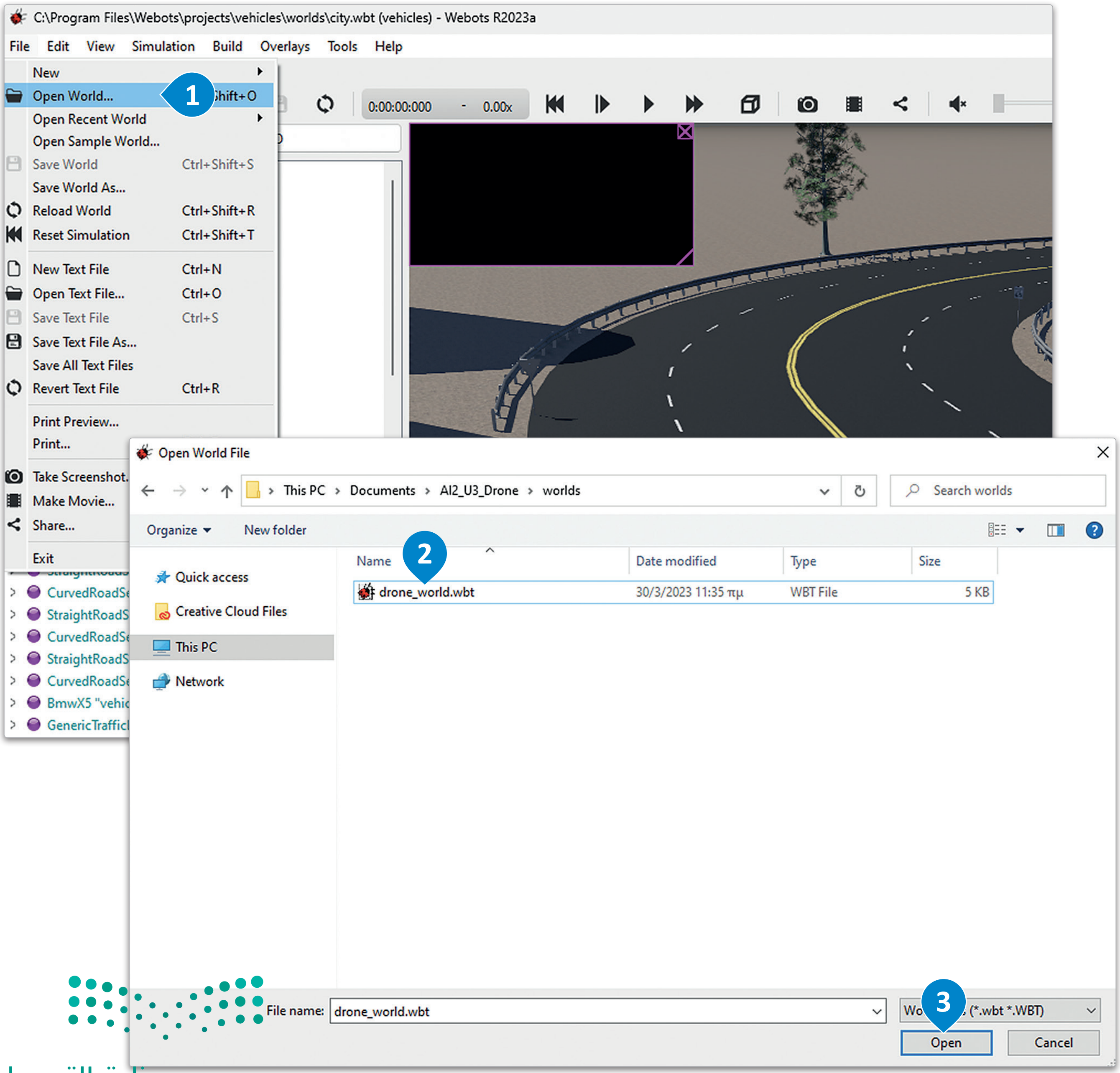
Likewise, you can install the Pillow library, by searching for "pillow".

Let's take a look at the project! First you will have to find and load the Webots world file:

To open a Webots world:

> Click **File > Open World...** from the **Menu bar**. 1

> Find the **drone_world.wbt** file in the **worlds** directory 2 and open it. 3



Name	Date modified	Type	Size
drone_world.wbt	30/3/2023 11:35 πμ	WBT File	5 KB

File name: drone_world.wbt
Wo... (*.wbt *.WBT)

Figure 6.9: Opening a Webots world

Ministry of Education
2023 - 1445

Next open the Python script file that will be used to control the drone:

To open a controller script:

> Click **File > Open Text File...** from the Menu bar. 1

> Find the **drone_controller.py** file in the **controllers, drone_controller** directory 2 and open it. 3

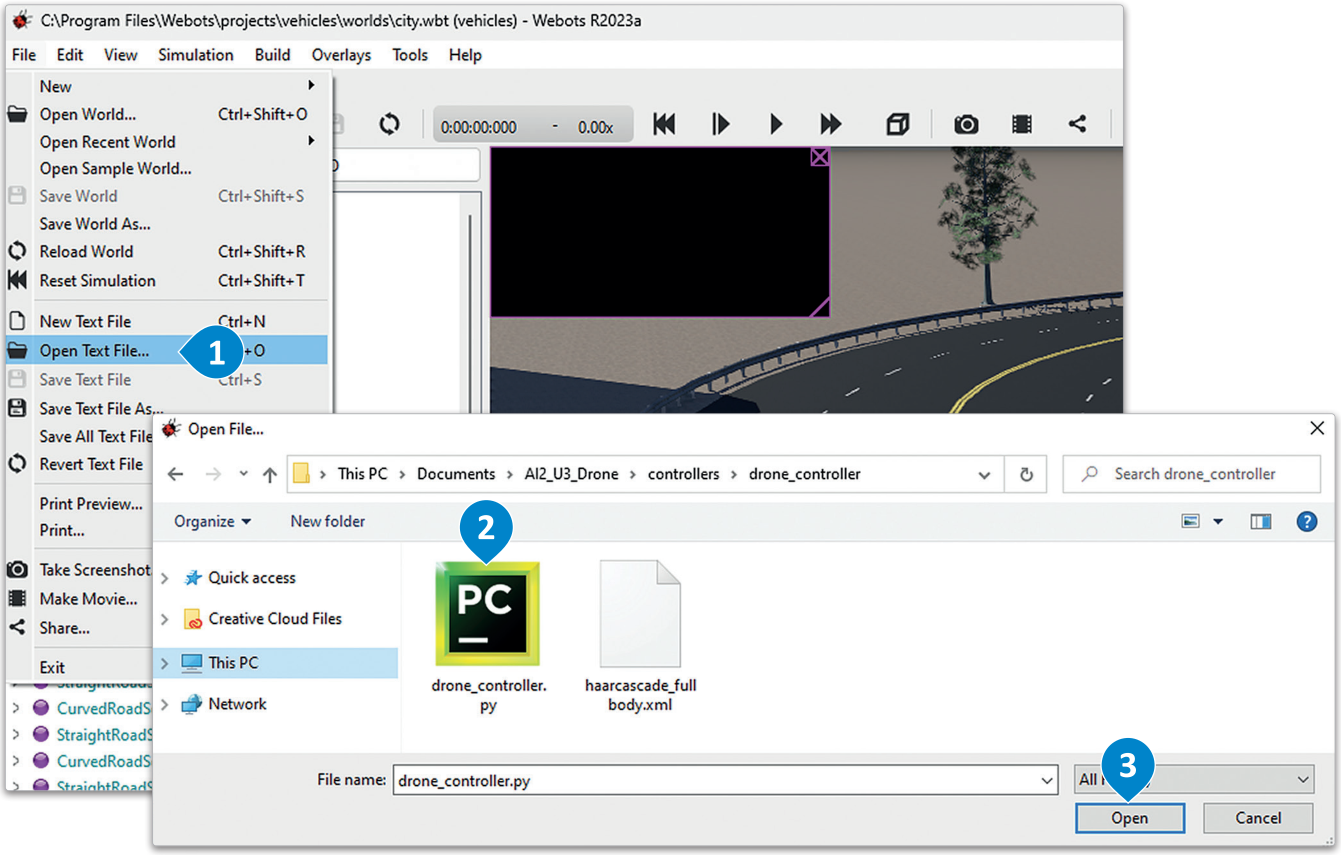


Figure 6.10: Opening a Webots controller script

Object Position and Rotation

X, Y, and Z are three-dimensional coordinates used to represent the position of an object in space. X represents the horizontal axis, Y represents the vertical axis, and Z represents the depth axis. They are similar to the real world coordinates of latitude, longitude and altitude, used to describe a location on Earth.

Pitch, roll, and yaw are rotational orientations that can be used to describe the movement of an object relative to a reference frame, as shown in figure 6.11. Pitch is the rotation of an object around its X-axis, which tilts the object up or down relative to a horizontal plane. Roll is the rotation of an object around its Y-axis, which tilts the object sideways or from side to side. Yaw is the rotation of an object around its Z-axis, which turns the object left or right relative to a reference frame.

Together, these six values (X, Y, Z, pitch, roll, and yaw) can be used to describe the position and orientation of an object in three-dimensional space. They are commonly used in robotics, navigation systems, and other applications that require precise positioning and control.

Drone Devices

The drone is equipped with several sensors, allowing it to collect input from its environment. getDevice() and enable() are functions provided by the simulator to interface with various sensors and actuators of a simulated robot.

The **getDevice()** function is used to get readings from a device, such as a sensor or an actuator, from the Webots robot model. It takes a string argument that specifies the name of the device to be accessed.

The **enable()** function is used to activate a device so that it can start providing data or performing an action.

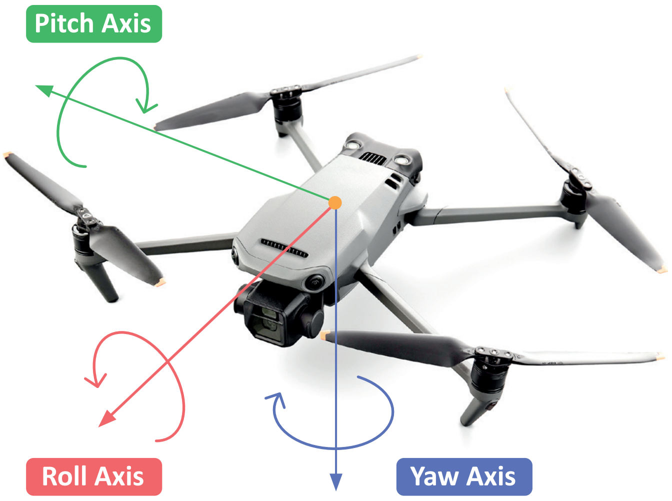


Figure 6.11: Rotational axes

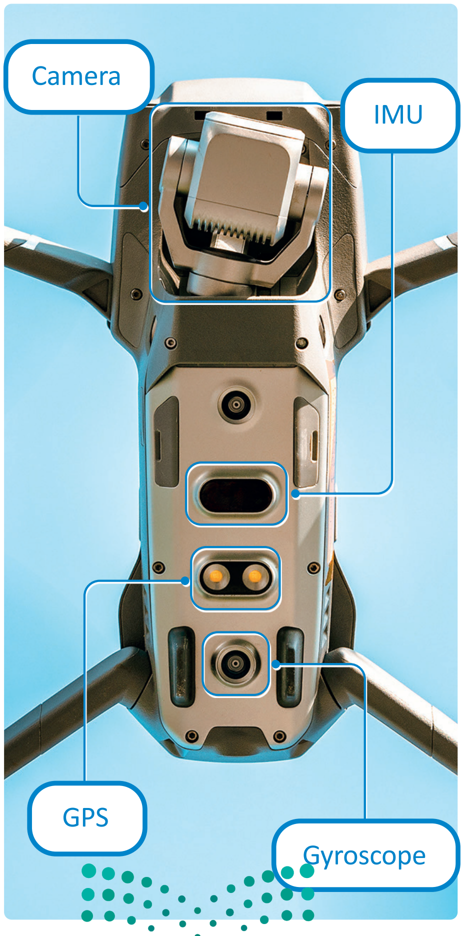


Figure 6.12: Drone with sensors and camera

The **IMU** (Inertial Measurement Unit) can measure the drone's linear acceleration and angular velocity; it measures forces such as gravity, in addition to the rotational forces acting on the drone. It can provide information about the drone's attitude (pitch, roll, and yaw), which is critical for stabilization and control.

The **GPS** (Global Positioning System) is a satellite-based navigation system that provides precise location information to the drone. GPS enables the drone to know its current position, altitude, and velocity relative to the earth. This information is important for drone navigation and control.

Sensors are devices that detect physical quantities or environmental conditions and measure it, and convert them into an electrical signal for monitoring or control.

Actuators are devices that convert electrical signals into mechanical motion to perform a specific action or task.

In contrast to linear speed, which measures the distance traveled in unit time, angular speed is a measure of the change in the central angle of a rotating object with respect to time. It is usually measured in radians per second (rad/s) or degrees per second (°/s).

The **gyroscope** is a sensor that measures angular velocity, or the rate of rotation around a specific axis. The gyroscope is especially useful in detecting and correcting small changes in the drone's orientation, which is important for maintaining stability and control during flight.

The drone's **camera** will be used to capture images during flight. It can be mounted on the drone and by adjusting the **camera pitch** angle with the **setPosition()** function, the drone can capture images from different perspectives and angles. In this project, the position is set to 0.7, which is about 45 degrees looking downwards.

The drone's **four propeller** devices are actuators that control the rotational speed and direction of the quadcopter. Quadcopters are drones that are equipped with four rotors, with two rotors rotating clockwise and the other two rotating counterclockwise. The rotation of these rotors generates lift and allows the drone to take off and maneuver in the air. Just like the rest of the devices, the motors are retrieved and set into position but the **setVelocity()** function is also used to set an initial velocity to the propellers.

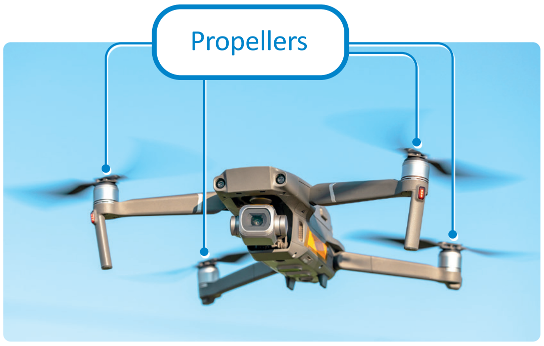


Figure 6.13: Four propeller drone device

Moving to a Target

To move from one location to the other, the drone uses the move_to_target() function; it contains the control logic. It takes a list of coordinates as argument, in the form of pairs [x, y], to be used as waypoints.

At first, it checks if the target position has been initialized, and if not, sets it to the first waypoint. Then, it checks if the drone has reached the target position with a precision of target_precision and if so, the function proceeds to the next target waypoint.

The angle between the current position of the drone and its target position has to be computed, in order to know how sharp it has to turn in the next step. This value is also normalized to the range of $[-\pi, \pi]$.

Next, it computes the yaw and pitch disturbances required to turn the drone towards the target waypoint and adjust the drone's pitch angle, respectively.

Motor Calculations

Lastly, the velocity that has to be set to the motors must be calculated. This is done by initially reading the sensor values: the roll, pitch and yaw from the IMU, and getting values of the x, y and z positions from the GPS while getting values of the roll and pitch accelerations from the gyroscope.

The various constants defined early in the code are used to make calculations and adjustments in conjunction with the sensor inputs and finally the correct thrust is set.

INFORMATION

By controlling the speed and direction of these four propellers, the quadcopter can move in any direction and maintain stable flight. For example, by increasing the speed of the two rotors on one side and decreasing the speed of the other two rotors, the drone can tilt and move in a specific direction.

```
from controller import Robot
import numpy as np     # used for mathematic operations
import os     # used for folder creation
import cv2     # used for image manipulation and human detection
from PIL import Image     # used for image object creation
from datetime import datetime     # used for date and time

# auxiliary function used for calculations
def clamp(value, value_min, value_max):
    return min(max(value, value_min), value_max)

class Mavic (Robot):

    # constants of the drone used for flight
    # thrust for the drone to lift
    K_VERTICAL_THRUST = 68.5
    # vertical offset the drone uses as targets for stabilization
    K_VERTICAL_OFFSET = 0.6
    K_VERTICAL_P = 3.0            # P constant of the vertical PID
    K_ROLL_P = 50.0               # P constant of the roll PID
    K_PITCH_P = 30.0              # P constant of the pitch PID

    MAX_YAW_DISTURBANCE = 0.4
    MAX_PITCH_DISTURBANCE = -1
    # precision between the target position and the drone position in meters
    target_precision = 0.5

    def __init__(self):
        # initializes the drone and sets the time interval between updates of the simulation
        Robot.__init__(self)
        self.time_step = int(self.getBasicTimeStep())

        # gets and enables devices
        self.camera = self.getDevice("camera")
        self.camera.enable(self.time_step)

        self.imu = self.getDevice("inertial unit")
        self.imu.enable(self.time_step)

        self.gps = self.getDevice("gps")
        self.gps.enable(self.time_step)

        self.gyro = self.getDevice("gyro")
        self.gyro.enable(self.time_step)

        self.camera_pitch_motor = self.getDevice("camera pitch")
        self.camera_pitch_motor.setPosition(0.7)

        self.front_left_motor = self.getDevice("front left propeller")
        self.front_right_motor = self.getDevice("front right propeller")
        self.rear_left_motor = self.getDevice("rear left propeller")
        self.rear_right_motor = self.getDevice("rear right propeller")
        motors = [self.front_left_motor, self.front_right_motor,
                  self.rear_left_motor, self.rear_right_motor]
        for motor in motors: # mass initialization of the four motors
            motor.setPosition(float('inf'))
            motor.setVelocity(1)
```

The controller library contains the Robot class, whose methods will be used to control the drone.

Imports of libraries needed for calculations and processing

Constants found empirically used in calculations for flight and stabilization

```
        self.current_pose = 6 * [0]   # X, Y, Z, yaw, pitch, roll
        self.target_position = [0, 0, 0]
        self.target_index = 0
        self.target_altitude = 0

    def move_to_target(self, waypoints):

        # Moves the drone to the given coordinates
        # Parameters:
        #    waypoints (list): list of X,Y coordinates
        # Returns:
        #    yaw_disturbance (float): yaw disturbance (negative value to go on the right)
        #    pitch_disturbance (float): pitch disturbance (negative value to go forward)

        if self.target_position[0:2] == [0, 0]:   # initialization
            self.target_position[0:2] = waypoints[0]

        # if the drone is at the position with a precision of target_precision
        if all([abs(x1 - x2) < self.target_precision for (x1, x2)
                    in zip(self.target_position, self.current_pose[0:2])]):

            self.target_index += 1
            if self.target_index > len(waypoints) - 1:
                self.target_index = 0
            self.target_position[0:2] = waypoints[self.target_index]

        # computes the angle between the current position of the drone and its target position
        # and normalizes the resulting angle to be within the range of [-pi, pi]
        self.target_position[2] = np.arctan2(
            self.target_position[1] - self.current_pose[1],
            self.target_position[0] - self.current_pose[0])
        angle_left = self.target_position[2] - self.current_pose[5]
        angle_left = (angle_left + 2 * np.pi) % (2 * np.pi)
        if (angle_left > np.pi):
            angle_left -= 2 * np.pi

        # turns the drone to the left or to the right according to the value
        # and the sign of angle_left and adjusts pitch_disturbance
        yaw_disturbance = self.MAX_YAW_DISTURBANCE * angle_left / (2 * np.pi)
        pitch_disturbance = clamp(
            np.log10(abs(angle_left)), self.MAX_PITCH_DISTURBANCE, 0.1)

        return yaw_disturbance, pitch_disturbance

    def run(self):

        # time intevals used for adjustments in order to reach the target altitude
        t1 = self.getTime()

        roll_disturbance = 0
        pitch_disturbance = 0
        yaw_disturbance = 0
```

Initialization of the drone's position (x, y, z) and rotation (roll, pitch, yaw)

```
# specifies the patrol coordinates
waypoints = [[-30, 20], [-60, 30], [-75, 0], [-40, -10]]
# target altitude of the drone in meters
self.target_altitude = 8

while self.step(self.time_step) != -1:

    # reads sensors
    roll, pitch, yaw = self.imu.getRollPitchYaw()
    x_pos, y_pos, altitude = self.gps.getValues()
    roll_acceleration, pitch_acceleration, _ = self.gyro.getValues()
    self.current_pose = [x_pos, y_pos, altitude, roll, pitch, yaw]

    if altitude > self.target_altitude - 1:
        # as soon as it reaches the target altitude,
        # computes the disturbances to go to the given waypoints
        if self.getTime() - t1 > 0.1:
            yaw_disturbance, pitch_disturbance = self.move_to_target(
                waypoints)
            t1 = self.getTime()

    # calculates the desired input values for roll, pitch, yaw,
    # and altitude using various constants and disturbance values
    roll_input = self.K_ROLL_P * clamp(roll, -1, 1) +
                 roll_acceleration + roll_disturbance
    pitch_input = self.K_PITCH_P * clamp(pitch, -1, 1) +
                  pitch_acceleration + pitch_disturbance
    yaw_input = yaw_disturbance
    clamped_difference_altitude = clamp(self.target_altitude -
                                   altitude + self.K_VERTICAL_OFFSET, -1, 1)
    vertical_input = self.K_VERTICAL_P *
                     pow(clamped_difference_altitude, 3.0)

    # calculates the motors' input values based on the
    # desired roll, pitch, yaw, and altitude values
    front_left_motor_input = self.K_VERTICAL_THRUST + vertical_input
                           - yaw_input + pitch_input - roll_input
    front_right_motor_input = self.K_VERTICAL_THRUST + vertical_input
                            + yaw_input + pitch_input + roll_input
    rear_left_motor_input = self.K_VERTICAL_THRUST + vertical_input
                          + yaw_input - pitch_input - roll_input
    rear_right_motor_input = self.K_VERTICAL_THRUST + vertical_input
                           - yaw_input - pitch_input + roll_input

    # sets the velocity of each motor based on the motors' input values calculated above
    self.front_left_motor.setVelocity(front_left_motor_input)
    self.front_right_motor.setVelocity(-front_right_motor_input)
    self.rear_left_motor.setVelocity(-rear_left_motor_input)
    self.rear_right_motor.setVelocity(rear_right_motor_input)

robot = Mavic()
robot.run()
```

The waypoints of the route the drone will be flying

Now it's time to insert the script into the drone and run the simulation:

To insert a controller and run the simulation:

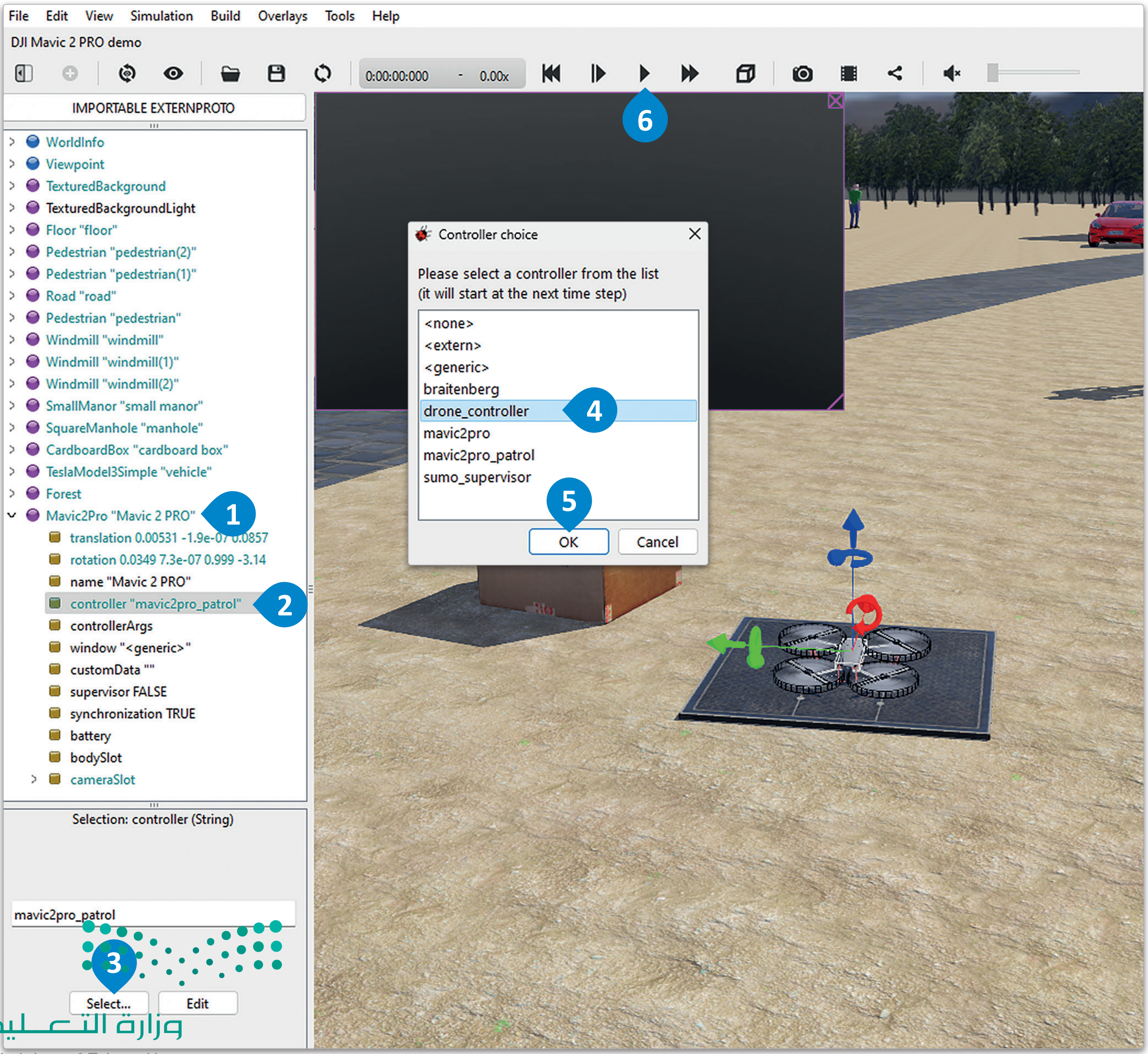
> Click **Mavic2Pro "Mavic 2 Pro"** in the **Scene tree** 1 and click **controller "mavic2pro".** 2

> Click **Select...** in the **Field editor.** 3

> Select **drone_controller** 4 and click **OK.** 5

> Click **Run the simulation in real-time** from the **Toolbar.** 6

When making changes to your scripts, do not forget to save them by pressing Ctrl + S.



WorldInfo
Viewpoint
TexturedBackground
TexturedBackgroundLight
Floor "floor"
Pedestrian "pedestrian(2)"
Pedestrian "pedestrian(1)"
Road "road"
Pedestrian "pedestrian"
Windmill "windmill"
Windmill "windmill(1)"
Windmill "windmill(2)"
SmallManor "small manor"
SquareManhole "manhole"
CardboardBox "cardboard box"
TeslaModel3Simple "vehicle"
Forest
Mavic2Pro "Mavic 2 PRO"
translation 0.00531 -1.9e-07 0.0857
rotation 0.0349 7.3e-07 0.999 -3.14
name "Mavic 2 PRO"
controller "mavic2pro_patrol"
controllerArgs
window "<generic>"
customData ""
supervisor FALSE
synchronization TRUE
battery
bodySlot
cameraSlot

Selection: controller (String)

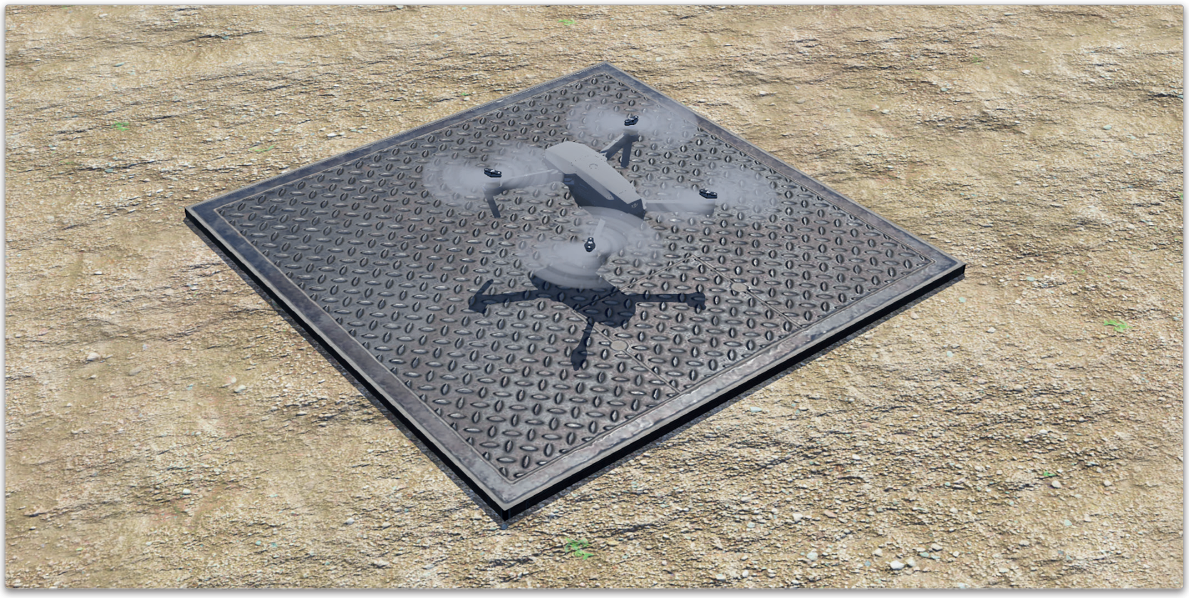
mavic2pro_patrol

Please select a controller from the list
(it will start at the next time step)

<none>
<extern>
<generic>
braitenberg
drone_controller
mavic2pro
mavic2pro_patrol
sumo_supervisor

Figure 6.14: Inserting the controller script and running the simulation

When you start the simulation, the drone's motors will power up and it will take off. Then it will follow the predeterminded route around the house, passing through the waypoints.



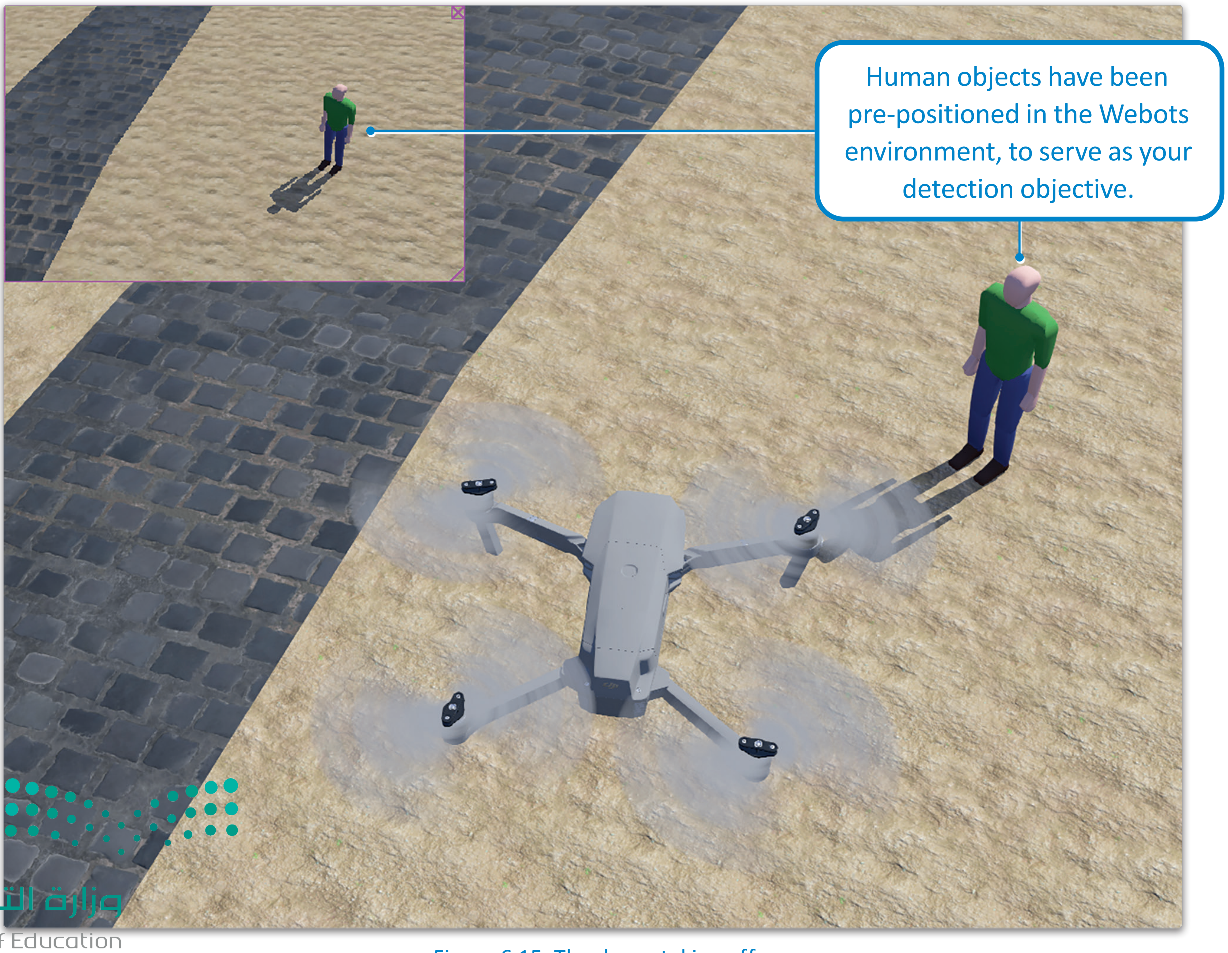

Human objects have been pre-positioned in the Webots environment, to serve as your detection objective.

Figure 6.15: The drone taking off

Exercises

1. Analyze the move_to_target() function and explain how the drone calculates its next position in the waypoints list. How can the drone's trajectory be optimized to minimize flight time between the waypoints?

2. Evaluate the limitations of the current drone control algorithm when faced with external factors such as wind, obstacles, or GPS inaccuracies. Propose and discuss improvements to the control algorithm that would make the drone more resilient to these challenges.

3 Explore the ethical implications of autonomous aerial drones in real-world applications, such as surveillance, package delivery, and search-and-rescue operations. Write down potential privacy concerns, safety issues, and the potential for misuse of this technology.

4 Add a feature that logs the drone's position, altitude, and orientation at regular intervals during its flight. Write down any patterns that you may find in the log data.

5 Experiment with different values for the PID controller constants (K_VERTICAL_P, K_ROLL_P, and K_PITCH_P). Observe how these changes affect the drone's stability and responsiveness. Discuss the trade-offs between stability and responsiveness.

Applications of Robotics II

Robotics, Computer Vision and AI

Computer vision and robotics are two cutting-edge fields of technology that together are rapidly changing the way people live and work. When combined, they open up a vast array of possibilities for automation, manufacturing, and developing other applications.

AI is a key component of both computer vision and robotics, enabling machines to learn and adapt to their environment over time. By using AI algorithms, robots can analyze and interpret vast amounts of visual data, allowing them to make decisions and take actions in real time. AI also enables robots to improve their performance and accuracy over time, as they learn from their experiences and adjust their behavior accordingly. This means that robots with computer vision and AI capabilities can perform increasingly complex tasks with greater efficiency and accuracy.

In this lesson, you will upgrade the initial drone project from the previous lesson to use computer vision in order to detect human figures near the house. These figures can be perceived as hostile in a real life scenario and the drone, using its camera, acts as surveillance system. This example can easily be applied and implemented to various other buildings, infrastructure, private and company properties, such as factories and energy plants.

To detect the human figures, you will be using the OpenCV library for Python. OpenCV (Open Source Computer Vision Library) is an open-source computer vision library that provides a range of computer vision and image processing algorithms as well as a set of programming tools for developing applications in these areas.

OpenCV can be used in robotics for tasks such as object detection and tracking, 3D reconstruction, and navigation. Its features also include object detection and recognition, face detection and recognition, image and video processing, camera calibration, machine learning, and more.

OpenCV is widely used in research and development projects in fields such as robotics, automation, surveillance, and medical imaging. It is also used in commercial applications such as face recognition, video surveillance, and augmented reality.

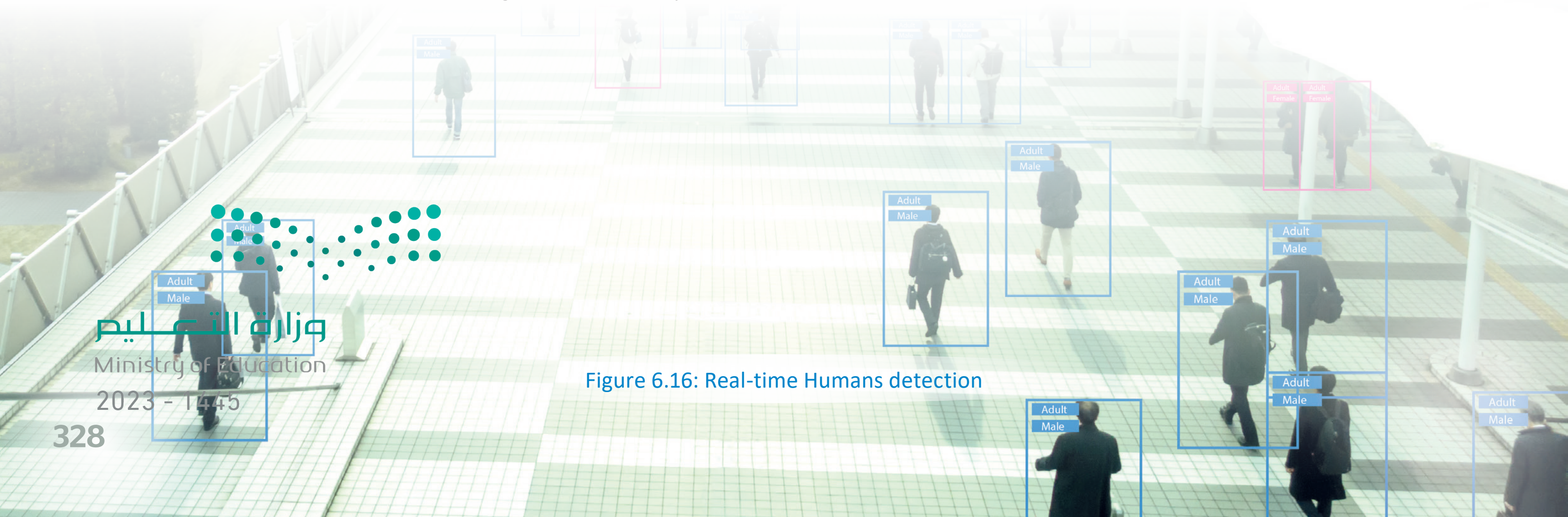


Figure 6.16: Real-time Humans detection

Let's take a look at the changes you will be making to add computer vision functionalities to your drone.

Adding a Timer

Capturing, processing and saving an image can be computationally expensive if calculated for every frame of the simulation. This is why you will be adding a timer that will be used so these actions are only performed every 5 seconds.

```python
# time intervals used for adjustments in order to reach the target altitude
t1 = self.getTime()
# time intervals between each detection for human figures
t2 = self.getTime()
```

Creating a Folder

The captured images in which human figures are detected will be saved in a folder. This is done as part of the security surveillance archive, so the images can be examined in the future.

First you must retrieve, with the **getcwd()** function, the path of the controller's current working directory (the folder the controller is in) in order for the program to know where to place the new folder. The new folder is named "detected" and the path's name is concatenated with the folder's name string with the **path.join()** function. The last step is to check whether the folder already exists and if not, the folder is created.

```python
# gets the current working directory
cwd = os.getcwd()
# sets the name of the folder where the images
# with detected humans will be stored
folder_name = "detected"
# joins the current working directory and the new folder name
folder_path = os.path.join(cwd, folder_name)

if not os.path.exists(folder_path):
# creates the folder if it doesn't exist already
    os.makedirs(folder_path)
    print(f"Folder \"detected\" created!")
else:
    print(f"Folder \"detected\" already exists!")
```

Image Processing

Now it is time to retrieve (read) the image from the device so as to process it before attempting detection. Notice that everything related to the image processing and up to its saving happens only every 5 seconds as it is inside the **"self.getTime() - t2 > 5.0"** condition.

```python
# initiates the image processing and detection routine every 5 seconds
if self.getTime() - t2 > 5.0:

    # retrieves image array from camera
    cameraImg = self.camera.getImageArray()
```

After the image is checked that it was retrieved successfully, the algorithm proceeds to modify some properties of the image. The image is 3-dimensional; it has dimensions of height, width and the color channels. The drone's camera captures images of 240 pixels in height and 400 pixels in width. It also uses 3-color channels to save the image information: red, green and blue.

In order to be used for detection, the image has to be manipulated first. For the functions to be applied properly later, it has to fit a particular structure. In this case, the sequence of the dimensions has to change from (height, width, color channels) to (color channels, height, width) by using the **transpose()** function. This function is given as arguments the camera image **cameraImg** and the new sequence **(2, 0, 1)**, assuming the original order was **(0, 1, 2)**.

The dimension sizes have to be adjusted too after the change in sequence. The **reshape()** function is used in the same manner, but with the respective dimension sizes **(3, 240, 400)** as the second argument.

```
# reshapes image array to (channels, height, width) format
cameraImg = np.transpose(cameraImg, (2, 0, 1))
cameraImg = np.reshape(cameraImg, (3, 240, 400))
```

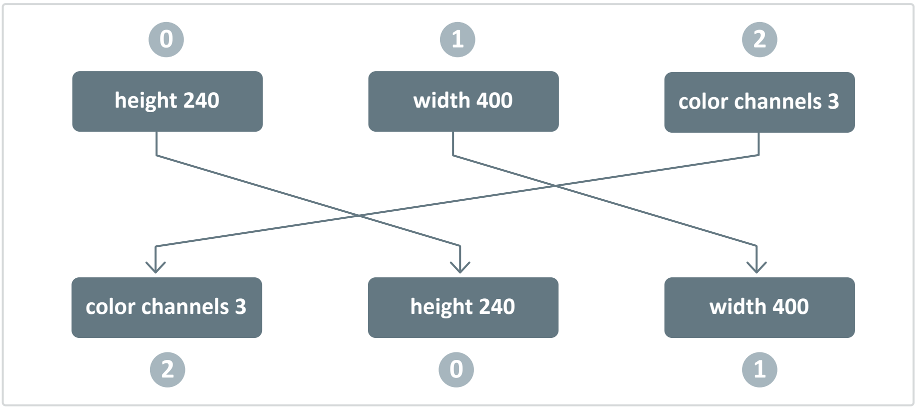


Figure 6.17: The dimensions' sequence change

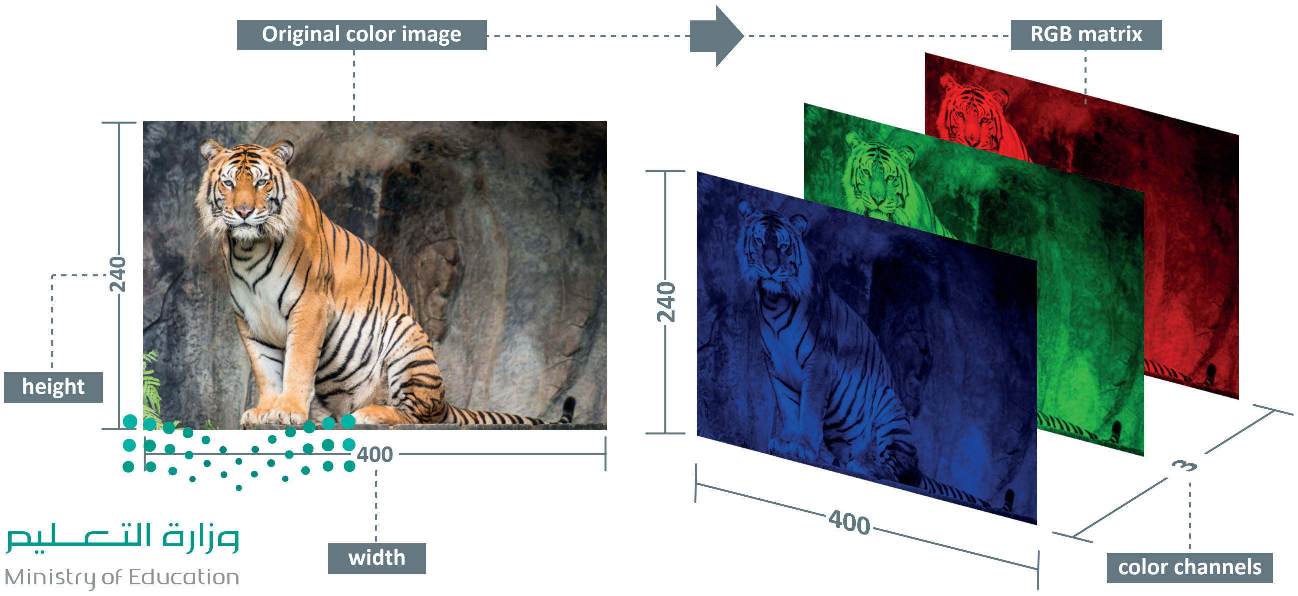


Figure 6.18: The dimensions of the image

Next, the image has to be changed into grayscale, as needed by the detection, but before that it must be stored in an Image object and have its 3 color channels combined. Here the color channels have to be merged and stored with the **merge()** function in reverse sequence, meaning in BGR (Blue, Green, Red) instead of RGB (Red, Green, Blue), (2, 1, 0) instead of (0, 1, 2) respectively.

```
# creates RGB image from merged channels
img = Image.new('RGB', (400, 240))
img = cv2.merge((cameraImg[2], cameraImg[1], cameraImg[0]))
```

Finally, the image is converted to grayscale with the **cvtColor()** function using the COLOR_BGR2GRAY argument, to change from BGR to grayscale.

```
# converts image to grayscale
gray = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)
```

Human Silhouette Detection

For the detection, you will use the Haar Cascade classifier. The Haar Cascade classifier is a machine learning-based object detection algorithm used to identify objects in images or videos. To use it, you need to train a machine learning model with a set of images that have the object you want to find, and others that do not. The algorithm looks for certain patterns in the pictures to determine where the object is. This algorithm is often used to find things like faces or people walking in a video. However, it might not work well in some situations where the object is partially/fully occluded or exposed to low illumination.

The classifier in your project is particularly trained for human detection. The **haarcascade_fullbody.xml** file provided to you is the pre-trained machine learning model you will use and part of the OpenCV library. It is given as argument to the **CascadeClassifier()** object and the function **detectMultiScale()** is called after to perform the detection.

```
# loads and applies the Haar cascade classifier to detect humans in image
human_cascade = cv2.CascadeClassifier('haarcascade_fullbody.xml')
humans = human_cascade.detectMultiScale(gray)
```

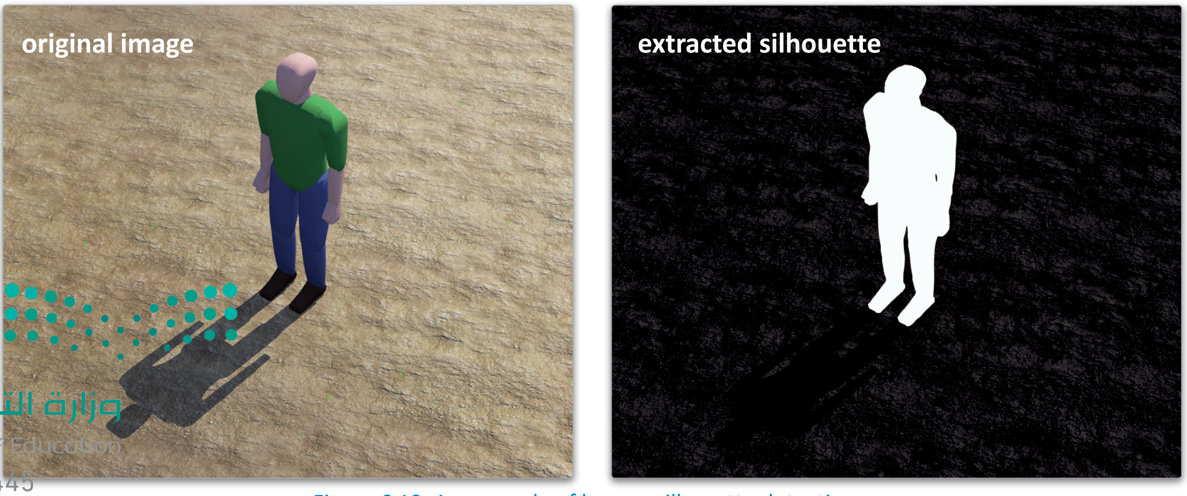


Figure 6.19: An example of human silhouette detection

Drone Report and Saving of the Detected Images

The final addition to your controller is a simple report system given by the drone in the form of printing a message on the console when a human form is detected and saving the image to the folder you created before.

The variable **humans** holds the rectangles (bounding boxes) inside which humans are detected, if they are found. The rectangles are defined by 4 variables: the pair of **x** and **y**, the two coordinates in the picture of the top left corner of the rectangle, and the pair of **w** and **h**, the width and height of the rectangle. For all detections found in the image, the function **rectangle()** marks the humans with a blue rectangle. The function takes as parameters the image, the top left corner **(x, y)** and bottom right corner **(x+w, y+h)** of the rectangle, and the rectangle's color and its width. Here, the rectangle is blue (B=255, G=0, R=0) and its width is 2.

The report system will retrieve the current date and time by using the **datetime.now()** function and print it on the console, along with the drone's coordinates at the time of the report.

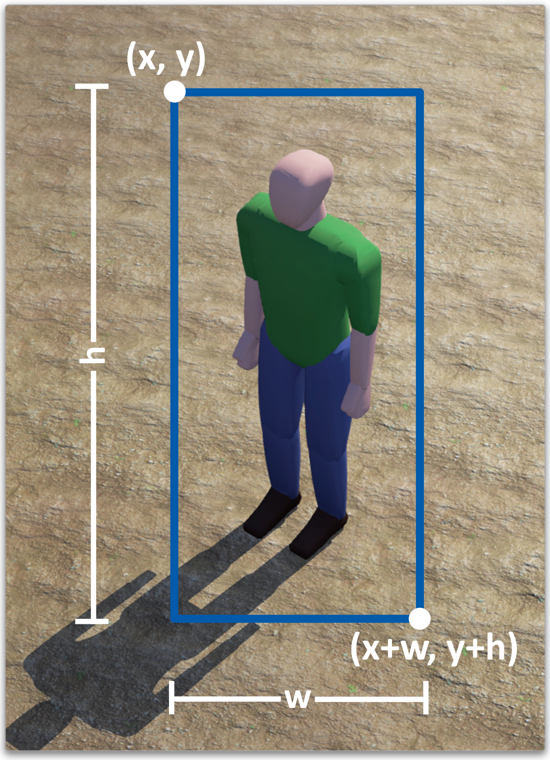


Figure 6.20: The variables of the rectangle

The date and time format is slightly modified by inserting dashes (-) and undersores (_) to be used as part of saved file's name and then saved in the folder with the function **imwrite()**. When everything is completed, the **getTime()** function resets the timer.

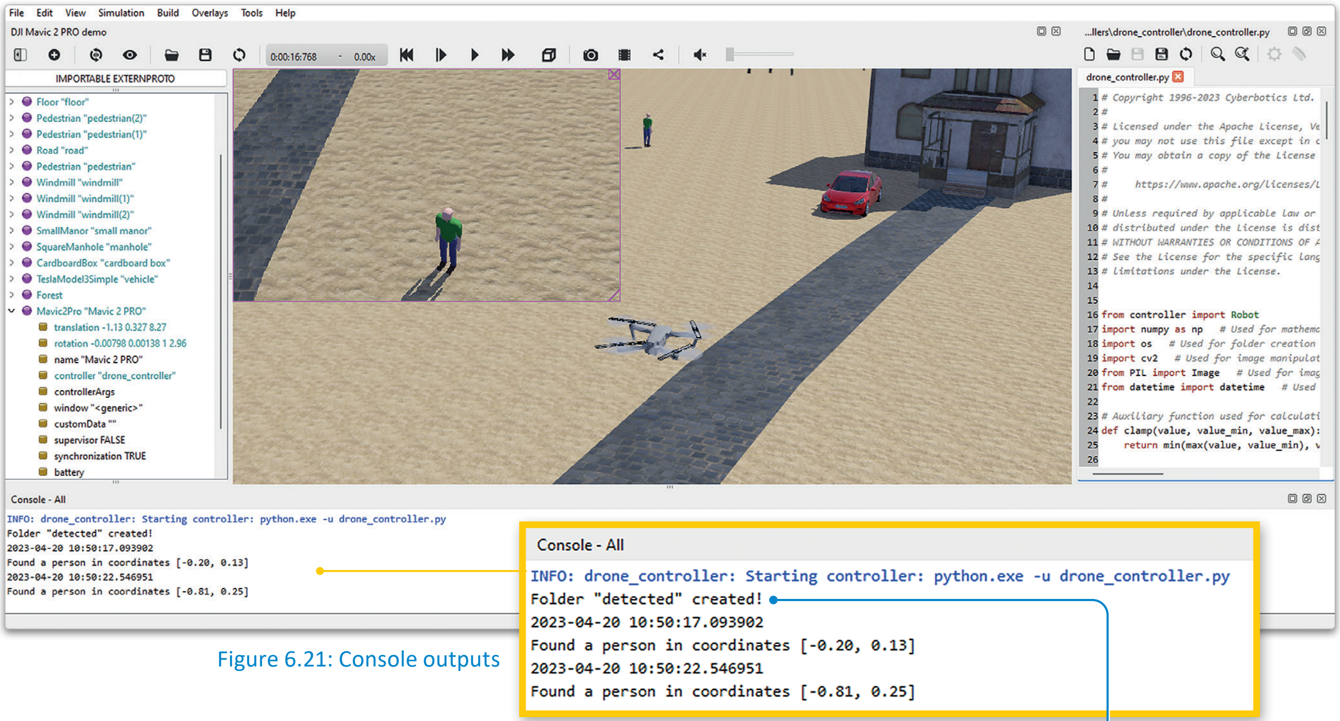
```
# loop, through detected human images, annotates them with a bounding box
# and prints a timestamp and an info message on the console
for (x, y, w, h) in humans:

    # the image, the top left corner, the bottom right corner, color and width of the rectangle
    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
    current_time = datetime.now()
    print(current_time)
    print("Found a person in coordinates [{:.2f}, {:.2f}]"
                .format(x_pos, y_pos))

    # saves annotated image to file with timestamp
    current_time = current_time.strftime("%Y-%m-%d_%H-%M-%S")
    filename = f"detected/IMAGE_{current_time}.png"
    cv2.imwrite(filename, img)

t2 = self.getTime()
```

In a string, the notation {:.2f} is used as a placeholder for a floating-point number with two decimal places. Here, two placeholders are used for the two variables, x_pos and y_pos.

After adding all these functionalities, the **run()** function of your controller should look like this:

```
def run(self):

    # time intervals used for adjustments in order to reach the target altitude
    t1 = self.getTime()
    # time intervals between each detection for human figures
    t2 = self.getTime()

    roll_disturbance = 0
    pitch_disturbance = 0
    yaw_disturbance = 0

    # specifies the patrol coordinates
    waypoints = [[-30, 20], [-60, 30], [-75, 0], [-40, -10]]
    # target altitude of the drone in meters
    self.target_altitude = 8

    # gets the current working directory
    cwd = os.getcwd()
    # sets the name of the folder where the images
    # with detected humans will be stored
    folder_name = "detected"
    # joins the current working directory and the new folder name
    folder_path = os.path.join(cwd, folder_name)

    if not os.path.exists(folder_path):
    # creates the folder if it doesn't exist already
        os.makedirs(folder_path)
        print(f"Folder \"detected\" created!")
    else:
        print(f"Folder \"detected\" already exists!")

    while self.step(self.time_step) != -1:

        # reads sensors
        roll, pitch, yaw = self.imu.getRollPitchYaw()
        x_pos, y_pos, altitude = self.gps.getValues()
        roll_acceleration, pitch_acceleration, _ = self.gyro.getValues()
        self.current_pose = [x_pos, y_pos, altitude, roll, pitch, yaw]

        if altitude > self.target_altitude - 1:
            # as soon as it reaches the target altitude,
            # computes the disturbances to go to the given waypoints
            if self.getTime() - t1 > 0.1:
                yaw_disturbance, pitch_disturbance = self.move_to_target(
                   waypoints)
                t1 = self.getTime()

        # initiates the image processing and detection routine every 5 seconds
        if self.getTime() - t2 > 5.0:

            # retrieves image array from camera
            cameraImg = self.camera.getImageArray()

            # checks if image is successfully retrieved
            if cameraImg:
                # reshapes image array to (channels, height, width) format
                cameraImg = np.transpose(cameraImg, (2, 0, 1))
                cameraImg = np.reshape(cameraImg, (3, 240, 400))

                # creates RGB image from merged channels
                img = Image.new('RGB', (400, 240))
                img = cv2.merge((cameraImg[2], cameraImg[1], cameraImg[0]))

                # converts image to grayscale
                gray = cv2.cvtColor(np.uint8(img), cv2.COLOR_BGR2GRAY)

                # loads and applies the Haar cascade classifier to detect humans in image
                human_cascade = cv2.CascadeClassifier('haarcascade_fullbody.xml')
                humans = human_cascade.detectMultiScale(gray)

                # loop, through detected human images, annotates them with a bounding box
                # and prints a timestamp and an info message on the console
                for (x, y, w, h) in humans:

                    cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
                    current_time = datetime.now()
                    print(current_time)
                    print("Found a person in coordinates [{:.2f}, {:.2f}]"
                        .format(x_pos, y_pos))

                    # saves annotated image to file with timestamp
                    current_time = current_time.strftime("%Y-%m-%d_%H-%M-%S")
                    filename = f"detected/IMAGE_{current_time}.png"
                    cv2.imwrite(filename, img)

                t2 = self.getTime()

        # calculates the desired input values for roll, pitch, yaw,
        # and altitude using various constants and disturbance values
        roll_input = self.K_ROLL_P * clamp(roll, -1, 1)
                                        + roll_acceleration + roll_disturbance
        pitch_input = self.K_PITCH_P * clamp(pitch, -1, 1)
                                        + pitch_acceleration + pitch_disturbance
        yaw_input = yaw_disturbance
        clamped_difference_altitude = clamp(self.target_altitude
                                        - altitude + self.K_VERTICAL_OFFSET, -1, 1)
        vertical_input = self.K_VERTICAL_P * pow(clamped_difference_altitude, 3.0)

        # calculates the motors' input values based on the desired roll, pitch, yaw, and altitude values
        front_left_motor_input = self.K_VERTICAL_THRUST
                    + vertical_input - yaw_input + pitch_input - roll_input
        front_right_motor_input = self.K_VERTICAL_THRUST
                    + vertical_input + yaw_input + pitch_input + roll_input
        rear_left_motor_input = self.K_VERTICAL_THRUST + vertical_input
                    + yaw_input - pitch_input - roll_input
        rear_right_motor_input = self.K_VERTICAL_THRUST + vertical_input
                    - yaw_input - pitch_input + roll_input

        # sets the velocity of each motor based on the motors' input values calculated above
        self.front_left_motor.setVelocity(front_left_motor_input)
        self.front_right_motor.setVelocity(-front_right_motor_input)
        self.rear_left_motor.setVelocity(-rear_left_motor_input)
        self.rear_right_motor.setVelocity(rear_right_motor_input)
```

Now, run the simulation to see your drone taking off and patroling around the house. Notice the new console outputs and the images created in the folder.


File Edit View Simulation Build Overlays Tools Help
DJI Mavic 2 PRO demo
0:00:16:768 - 0.00x
IMPORTABLE EXTERNPROTO
...llers\drone_controller\drone_controller.py
drone_controller.py
Console - All

```
Floor "floor"
Pedestrian "pedestrian(2)"
Pedestrian "pedestrian(1)"
Road "road"
Pedestrian "pedestrian"
Windmill "windmill"
Windmill "windmill(1)"
Windmill "windmill(2)"
SmallManor "small manor"
SquareManhole "manhole"
CardboardBox "cardboard box"
TeslaModel3Simple "vehicle"
Forest
Mavic2Pro "Mavic 2 PRO"
    translation -1.13 0.327 8.27
    rotation -0.00798 0.00138 1 2.96
    name "Mavic 2 PRO"
    controller "drone_controller"
    controllerArgs
    window "<generic>"
    customData ""
    supervisor FALSE
    synchronization TRUE
    battery
```

```python
# Copyright 1996-2023 Cyberbotics Ltd.
#
# Licensed under the Apache License, Ve
# you may not use this file except in c
# You may obtain a copy of the License
#
#     https://www.apache.org/licenses/L
#
# Unless required by applicable law or
# distributed under the License is dist
# WITHOUT WARRANTIES OR CONDITIONS OF A
# See the License for the specific lang
# limitations under the License.


from controller import Robot
import numpy as np   # Used for mathema
import os   # Used for folder creation
import cv2   # Used for image manipulat
from PIL import Image   # Used for imag
from datetime import datetime   # Used

# Auxiliary function used for calculati
def clamp(value, value_min, value_max):
    return min(max(value, value_min), v

```

```
Console - All
INFO: drone_controller: Starting controller: python.exe -u drone_controller.py
Folder "detected" created!
2023-04-20 10:50:17.093902
Found a person in coordinates [-0.20, 0.13]
2023-04-20 10:50:22.546951
Found a person in coordinates [-0.81, 0.25]
```

Figure 6.21: Console outputs

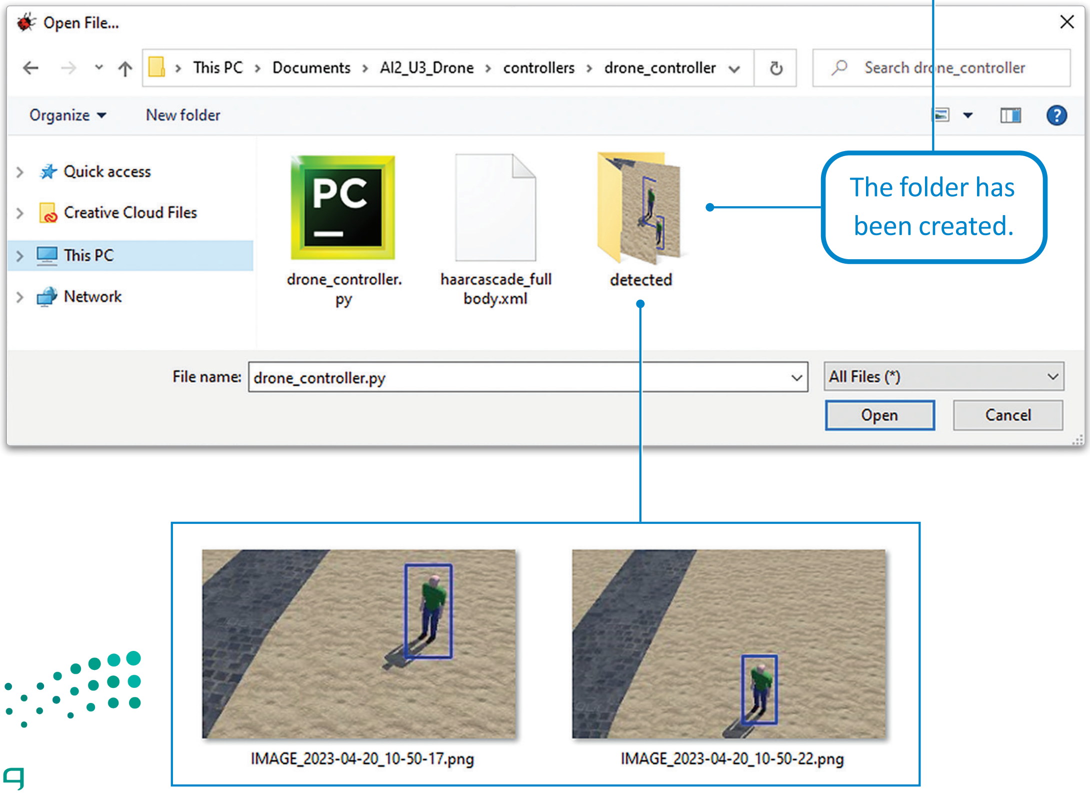

Open File...
This PC > Documents > AI2_U3_Drone > controllers > drone_controller
Search drone_controller
Organize
New folder
Quick access
Creative Cloud Files
This PC
Network
drone_controller.py
haarcascade_fullbody.xml
detected
The folder has been created.
File name: drone_controller.py
All Files (*)
Open
Cancel
IMAGE_2023-04-20_10-50-17.png
IMAGE_2023-04-20_10-50-22.png

Figure 6.22: Folder creation and images saved containing detections

Ministry of Education
2023 - 1445

Exercises

1 Modify your controller to not check if the folder already exists in the path. Does it create any complications in the execution of the simulation?

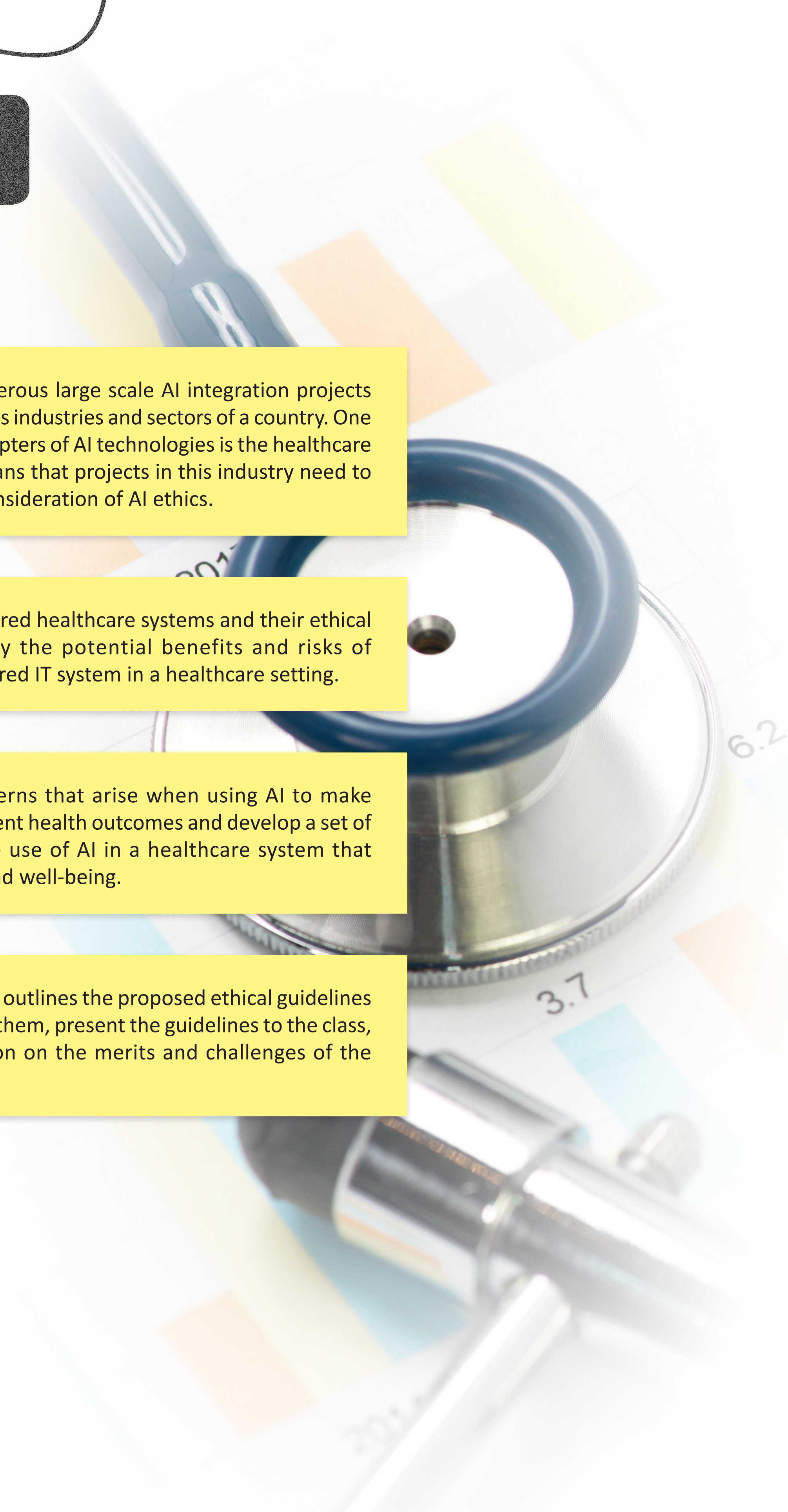
2 Modify your controller to perform a detection every 10 seconds. Do you notice any difference in the frequency of the console prints and the images saved?

3 What would happen to the image output if you merged the color dimensions in the normal sequence instead of the reversed one? Write down your observations below.

4 Experiment with the fourth and fifth arguments of the rectangle() function. Write down your observations below.

5 Modify your controller to also print the drone’s roll, pitch and yaw values when detecting a person.

Project



Nowadays there are numerous large scale AI integration projects being developed for various industries and sectors of a country. One of the most important adopters of AI technologies is the healthcare industry. But this also means that projects in this industry need to be developed with the consideration of AI ethics.

1 Research existing AI-powered healthcare systems and their ethical implications and Identify the potential benefits and risks of implementing an AI-powered IT system in a healthcare setting.

2 Analyze the ethical concerns that arise when using AI to make decisions that impact patient health outcomes and develop a set of ethical guidelines for the use of AI in a healthcare system that prioritize patient safety and well-being.

3 Create a presentation that outlines the proposed ethical guidelines and the reasoning behind them, present the guidelines to the class, and engage in a discussion on the merits and challenges of the proposed guidelines.

Wrap up

Now you have learned to:

> Provide an overview of AI ethics.

> Examine how bias and lack of fairness can lead to misuse of AI systems.

> Outline the methods to mitigate the transparency and explainability problem in AI.

> Evaluate how government regulations and standards guide the ethical and sustainable use of AI systems.

> Program an aerial drone to move through an environment without human intervention.

> Modify an aerial drone's system to include surveillance capabilities through image analysis.

KEY TERMS

AI Ethics	Gyroscope	Propeller
Bias	Human Detection	Robotics
Black-Box Problem	IMU (Inertial Measurement Unit)	Roll
Debiasing	Motor	Simulator
Drone Surveillance	OpenCV	Value-Based Reasoning
GPS (Global Positioning System)	Pitch	Yaw