RWANDA POLYTECHNIC

HUYE COLLEGE

MODULE NAME AND CODE: ITLPA701 PYTHON AND FUNDAMENTALS OF AI

RTQF Level: 7

Credits: 10

Sector: ICT

Sub-sector: IT

Module Leader: NTAMBARA Etienne

E-mail: entambara@rp.ac.rw

Academic Year: 2024-2025

Rwanda Polytechnic - Huye college
Academic year: 2024-2025

Abstract

This module aims to introduce learners to the fundamental of applying Python programming language in various projects development, what makes it so massively popular, and its benefits and limitations. After completion of this module, learner will be able to apply python concept, developing AI based programs, web development (Server-side), software development, Data analysis as well as system scripting.

Contents

List of Figures

Source Code

1 Introduction

1.1 What is python

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python's extensive standard library and large community make it suitable for various applications such as web development, data analysis, machine learning, artificial intelligence, scientific computing, and automation. Its clean syntax and ease of use make it an excellent choice for both beginners and experienced developers, enabling rapid development and efficient problem-solving.

1.2 What can python Do

Python is a versatile programming language capable of performing a wide range of tasks across different fields. It can be used for web development (using frameworks like Django and Flask), data analysis and visualization (with libraries such as Pandas and Matplotlib), machine learning and artificial intelligence (using TensorFlow, PyTorch, and scikit-learn), automation and scripting, scientific computing, game development, and cybersecurity. Additionally, Python is widely used for software development, database management, and building APIs. Its simplicity and extensive library ecosystem make it suitable for both small-scale projects and large, complex systems.

1.3 Prerequisite

While Python is known for its ease of use and beginner-friendly syntax, having a basic understanding of computer concepts can be helpful when getting started. Familiarity with ideas like variables, data types, loops, and conditional statements can ease the learning process, although no extensive programming background is required. Basic math and logic skills also enhance problem-solving abilities, making it easier to grasp how Python programs function.

2 Prepare Python environment

2.1 Identify Python version

2.1.1 Definition of terms

Python: is a high-level, interpreted programming language known for its simplicity, readability, and versatility. It is widely used in various fields, including web development, data science, artificial intelligence, and automation[1].

Programming Language: is a set of instructions and rules used to communicate with a computer and develop software applications. It allows programmers to write code that a machine can execute to perform specific tasks.

Use of Python: Python is used in various domains, including:

1. **Server-side Development**: Python is commonly used for backend development in web applications. Frameworks like Django and Flask help in handling database interactions, user authentication, and web page rendering.

2. **Software Development**: Python is used to build desktop applications, mobile apps, and enterprise solutions. It provides tools for GUI development, automation, and integration with other programming languages.

3. **Mathematics**: Python supports mathematical computations through libraries such as NumPy, SciPy, and SymPy. It is widely used in data analysis, machine learning, and scientific research.

4. **System Scripting**: Python can be used for automating system administration tasks, such as file handling, process management, and network scripting. It is commonly employed in DevOps and cybersecurity.

To identify the version of Python installed on your system, you can use the following command:

```
python --version //cmd

python3 --version  //cmd


//python script
import sys
print(sys.version)
```

Listing 1: Checking python version

2.1.2 Characteristic of python

1. **Structure**: Python has a well-organized structure that emphasizes readability and simplicity:

   (a) **Indentation-based syntax**: Python uses indentation instead of brackets () to define code blocks, making it clean and readable.
   (b) **Object-oriented**: Supports object-oriented programming (OOP) principles such as classes and inheritance.
   (c) **Dynamic typing**: Variables do not require explicit declaration, making coding flexible and reducing complexity.
   (d) **Interpreted language**: Python code is executed line by line, which simplifies debugging.

2. **Application**: Python is used in various domains due to its flexibility and extensive libraries:
   (a) **Web development**: Frameworks like Django and Flask help build powerful web applications.
   (b) **Data science and AI**: Libraries like TensorFlow, NumPy, and Pandas make Python ideal for machine learning and data analysis.
   (c) **Cybersecurity**: Used for penetration testing and security automation with tools like Scapy and PyCryptodome.
   (d) **Embedded systems**: Python is used in IoT and robotics through platforms like Raspberry Pi.
   (e) **Game development**: Libraries like Pygame allow developers to create 2D games.
3. **Checking**: Python provides various mechanisms for error handling and code validation
   (a) **Type checking**: Python is dynamically typed, but tools like **mypy** can enforce static type checking.
   (b) **Error handling**: **try-except** blocks handle runtime errors gracefully
   (c) **Linting tools**: Tools like **pylint** and **flake8** analyze Python code for syntax errors and best practices.
   (d) **Unit testing**: The **unittest** module helps in automated testing of code
4. **Integration**: Python integrates seamlessly with various technologies and languages
   (a) **Database integration**: Supports databases like MySQL, PostgreSQL, and MongoDB using libraries like **SQLAlchemy** and **PyMongo**.
   (b) **Language interoperability**: Works with C, C++, and Java through tools like **Cython, JPype**, and **SWIG**.
   (c) **Cloud computing**: Supports AWS, Google Cloud, and Azure for cloud-based applications.
   (d) **API development**: Python can be used to build and consume APIs using **FastAPI** and **Flask-RESTful**.

2.1.3 Application of python

1. **Flexibility**: Python is highly flexible and can be used across various fields due to its adaptability
   (a) **Cross-platform compatibility**: Runs on Windows, macOS, Linux, and even mobile operating systems.
   (b) **Multi-paradigm support**: Supports procedural, object-oriented, and functional programming.
   (c) **Extensive libraries**: Has a vast collection of libraries and frameworks for web development, data science, AI, automation, and more.
   (d) **Scripting and automation**: Used for automating repetitive tasks, such as data processing and system administration.
2. **Audience**: Python is designed for a broad range of users, including:
   (a) **Beginners**: Its simple syntax makes it an ideal first programming language.
   (b) **Software developers**: Used for building web applications, desktop applications, and system scripts.
   (c) **Data scientists and AI researchers**: Preferred for machine learning, data analysis, and artificial intelligence.
   (d) **Cybersecurity experts**: Used in ethical hacking, penetration testing, and security automation.
   (e) **Engineers and scientists**: Supports numerical computing and simulations in physics, engineering, and bioinformatics.
3. **Prerequisite**: To start using Python effectively, some basic knowledge and skills are helpful
   (a) **Basic programming concepts**: Understanding variables, loops, and functions is beneficial but not mandatory.
   (b) **Mathematical skills**: Helpful for data science, AI, and scientific computing applications.
   (c) **Logical thinking**: Problem-solving and algorithmic thinking are essential for writing efficient Python programs.

(d) **Familiarity with English**: Since Python syntax is mostly based on English words, reading and writing English can help in understanding code more easily.

2.2 Add python Directory

2.2.1 Local Environment set up

When Python is installed, its executable file needs to be included in the system's environment variables (PATH) to be accessible from any command line or terminal.

1. **Windows**

   (a) Install Python from the official website: Python Downloads

   (b) During installation, **check the box**: Add Python to PATH

   (c) If Python is already installed, manually add it to PATH: **Open Control Panel → System → Advanced system settings**. Click **Environment Variables**. Under **System Variables**, find and edit **Path** Add the Python installation directory. Click OK and restart the terminal

2. **Linux / Unix**

   (a) Install Python using package managers:

```
    sudo apt update && sudo apt
    install python3

    //or
    sudo yum install python3


```

Listing 2: Installing Python

   (b) Verify installation:

```
        python3 --version


```

Listing 3: Verify installation

   (c) Add Python to PATH (if needed):

```
        echo "export PATH=$PATH:/
    usr/local/bin/python3" >> ~/.
    bashrc
source ~/.bashrc
```

Listing 4: Add path

3. **Mac (Macintosh)**

   (a) **Install Python using Homebrew**:

```
        brew install python3


```

Listing 5: Install python

   (b) **Verify installation**:

```
        python3 --version


```

Listing 6: Verify installation

   (c) **If needed, add Python to PATH**:

```
        echo "export PATH=$PATH:/
    usr/local/bin/python3" >> ~/.
    zshrc
source ~/.zshrc



```

Listing 7: Add path

4. **Raspberry Pi**

   (a) **Python is pre-installed in Raspberry Pi OS. To verify**:

```
    python3 --version
```

Listing 8: Verify if python installed

   (b) **If not installed, use**:

```
    sudo apt update && sudo apt
    install python3
```

Listing 9: Installing python

   (c) **Add Python to PATH if required**:

```
    echo "export PATH=$PATH:/usr/
    bin/python3" >> ~/.bashrc
source ~/.bashrc
```

Listing 10: Add path

Local Environment Setup: After installing Python, set up the environment for efficient development:

1. Install Virtual Environment (Optional but Recommended)

```
    python3 -m venv myenv


```

Listing 11: Install virtual environment

   Activate it: **Windows**: myenv
   Scripts
   activate **Linux/Mac**: source myenv/bin/activate

2. Install Essential Packages

```
    pip install --upgrade pip setuptools
     wheel


```

Listing 12: Activating environment

3. Verify Installation

```
    python --version
    pip --version


```

Listing 13: Verify installation

2.2.2 Running Python

Python can be executed in two main ways: via the **command line** or using an **Integrated Development Environment** (IDE).

1. **Running Python from the Command Line**
   (a) Open the command prompt (Windows) or terminal (Mac/Linux)
   (b) Type python (or python3 on some systems) and press Enter to start the interactive Python shell.
   (c) To run a Python script, navigate to its directory and execute:

```
python script.py
//or
python3 script.py
```

Listing 14: Run python Script

   (d) To exit the interactive shell, type **exit()** or press **Ctrl + Z** (Windows) / **Ctrl + D** (Mac/Linux).

2. **Running Python in an Integrated Development Environment (IDE)**
   (a) **Built-in options**: Python comes with IDLE, a simple IDE that allows writing, editing, and running Python code.
   (b) **Popular IDEs for Python**: **PyCharm**: Feature-rich, great for large projects. **Visual Studio Code (VS Code)**: Lightweight, highly customizable. **Jupyter Notebook**: Best for data science and AI. **Spyder**: Common for scientific computing. **Thonny**: Beginner-friendly.
   (c) Open the IDE, create a new Python script (**.py** file), write your code, and run it using the built-in execution feature.

2.3 Identify library

2.3.1 Definition of key terms

1. **Library**: A library in Python is a collection of pre-written modules and functions that provide specific functionalities, such as data manipulation, machine learning, or web development. Examples include **NumPy** for numerical computing, **Pandas** for data analysis, and **TensorFlow** for AI.

2. **Package**: is a structured collection of Python modules that are grouped together in a directory containing a special file called **__init__.py**. It helps organize code into reusable components. For example, **scipy** is a package that contains multiple scientific computing modules.

3. **Import**: The **import** statement in Python is used to **load and use modules or packages** in a program. For example:

```

import math  # Imports the math module
print(math.sqrt(16))  # Uses the sqrt function from the math module
```

Listing 15: Importing module

4. **Install**: Installing in Python refers to downloading and adding external libraries or packages to your Python environment. This is done using package managers like PIP. For example, to install NumPy:

```

pip install numpy
```

Listing 16: Install Numpy

5. **PIP (Python Package Installer)**: PIP is the default package manager for Python, used to **install, upgrade**, and **manage libraries** and **dependencies**. It allows users to download packages from the **Python Package Index (PyPI)**. To check if PIP is installed:

```
//check if pip is installed
pip --version

//install package
pip install package_name


//installing flask
pip install flask
```

Listing 17: Install PIP

2.3.2 Identification of problem

1. **Problem**: A problem in Python development refers to an issue or challenge that arises when writing, running, or deploying Python programs. This could include **syntax errors, runtime errors, logic errors, missing dependencies, or performance inefficiencies**. For example, a common problem is a **ModuleNotFoundError**, which occurs when trying to import a package that is not installed.

2. **Solution**: Solutions vary depending on the problem. For example:
   (a) **Syntax errors**: Check and correct the code syntax based on Python rules.
   (b) **Runtime errors**: Debug using **try-except** blocks to handle exceptions.

(c) **Missing dependencies**: Install required packages using PIP (**pip install package_name**).

(d) **Performance inefficiencies**: Optimize code using efficient algorithms and built-in functions.

2.3.3 Getting library

To use external libraries in Python, follow these steps:

1. **PIP (Python Package Installer)**: PIP is the default package manager for Python, used to install and manage libraries from the Python Package Index (PyPI).

   Check if PIP is installed:

```
pip --version

```

Listing 18: Check if PIP installed

   If PIP is not installed, proceed to the next step.

2. **Install PIP (if not already installed)**

```

//for window
python -m ensurepip --default-pip

//Linux/Mac (via package manager)
sudo apt install python3-pip  # For Debian/Ubuntu
sudo yum install python3-pip  # For CentOS/RHEL
brew install pip              # For macOS (using Homebrew)


//upgrade pip to the latest version
pip install --upgrade pip

```

Listing 19: Install PIP

3. **Browse for Libraries**: Visit PyPI(Python Package Index) to search for available libraries. Search for the required library and find installation details.

4. **Download and Install a Library**

```
//use pip to install desired library
pip install library_name

//installing numpy
pip install numpy

//install multiple libraries at once
pip install numpy pandas matplotlib

```

Listing 20: Download and Install

5. **Import the Library**: Once installed, import the library into your Python script:

```

import numpy as np  # Importing NumPy with an alias

```

Listing 21: Importing library

6. **Use the Library**: Use functions and features provided by the library

```

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

```

Listing 22: Using library

3 Develop Python concept

3.1 Writing Python syntax

3.1.1 Execute syntax

To execute Python code, you can use a code editor with the appropriate setup, including a Python extension for syntax highlighting, debugging, and more. Here's how you can set it up:

1. **Code Editor**: A code editor is where you write your Python code. Popular editors include: Visual Studio Code (VS Code), PyCharm, Sublime Text, Atom, Notepad++ (for Windows), Thonny (beginner-friendly). These editors make it easier to write Python code and support syntax highlighting, auto-completion, and more.

2. **Python Extension**: To enable Python-specific features such as syntax highlighting, auto-completion, and running scripts, you need to install the Python extension in your code editor.

   (a) **For Visual Studio Code**: Install VS Code. Open VS Code, go to the **Extensions** panel (on the left side), and search for Python. Click **Install** on the official Python extension by Microsoft. Once installed, you'll get Python syntax support, linting, debugging, and the ability to run scripts directly in the editor.

   (b) **For PyCharm**: comes with Python support built-in, so no additional installation is needed.

   (c) **For Sublime Text**: Install **Package Control** (if not already installed). Use Package Control to install the **Python** package for syntax highlighting and other features.

3. **Install Syntax for Running Python Code**: Once your code editor and Python extension are set up, ensure you can run Python code properly.

   (a) **For VS Code**: **Install Python** on your system (if not already done). Open the command palette (press Ctrl+Shift+P), and type **Python: Select Interpreter**. Choose the Python interpreter that matches your installed version. Open a new Python file (**.py**), and write your code. Run the Python script by pressing **F5** or using the Run button in the top-right corner.

   (b) **For PyCharm**: Install Python and configure it in **Settings → Project Interpreter**. Create a new Python file (**.py**), write your code, and press the **Run** button to execute it.

   (c) **For Sublime Text**: Ensure Python is installed on your system. You can install a **build system** in Sublime to run Python: Navigate to **Tools → Build System → Python**. Then press **Ctrl+B** (Windows/Linux) or **Cmd+B** (Mac) to run the code.

3.1.2 Use Command line

You can execute Python programs directly from the command line using **Python mode**. Here's a step-by-step guide to help you write, compile, and execute Python programs, including how to handle indentation.

1. **Python Mode**: When using the command line or terminal, you can enter Python mode by typing python (or python3 depending on your system setup). This opens an interactive Python shell where you can write and execute Python code interactively.

   To enter Python mode:

```
python

//or
python3

```

Listing 23: python mode

   In Python mode, you can type Python commands directly, and they will be executed line by line. For example:

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


```

Listing 24: example of python script

2. **Writing a Python Program (in a Code Editor or Command Line)**: While you can write code directly in Python mode, typically, you write your program in a file using a code editor (e.g., VS Code, Notepad++) or directly in the terminal.

   To write a Python program in a file, open a text editor, create a new file, and save it with the **.py** extension (e.g., **program.py**).

   Example code:

```
# Simple Python program
print("Hello, Python!")


```

Listing 25: simple program

3. **Compile the Python Program**: Unlike languages like C or Java, Python is an **interpreted language**, so you don't explicitly "compile" the program. Instead, you **execute** the program directly from the command line.

   To "compile" (i.e., run) your Python program, simply execute it using the following command:

```
python program.py

//or
python3 program.py


```

Listing 26: Compiling file

   This will run the Python script, and you will see the output in the terminal.

4. **Execute the Python Program**: After creating your **.py** file, executing it from the command line will output the result of the program.

   For example, executing **program.py**:

```

python program.py

//Output
Hello, Python!


```

Listing 27: example of compiling

5. **Save the Program**: To save your Python program:

   After writing the code in your text editor, go to **File → Save As**. Save the file with the **.py** extension, such as **program.py**. Make sure the file is saved in a directory where you can easily access it from the command line.

6. **Indentation in Python**: Python uses **indentation** (spaces or tabs) to define code blocks rather than braces () as in other programming languages. Correct indentation is crucial in Python.

   Example of correctly indented Python code:

```

    if True:
        print("This is inside the if statement")  # Indentation inside the block
        x = 10
        print(x)


```

Listing 28: Example of idented code

Incorrect indentation would cause an IndentationError:

```

    if True:
    print("This is inside the if statement")  # Missing indentation

```

Listing 29: Example of incorect idented code

Make sure you use consistent indentation—either spaces (usually 4 spaces per level) or tabs—but don't mix both. Python's interpreter relies on this to determine the scope of loops, functions, and other control structures.

3.1.3 Apply comments

1. **Purpose**: Comments are used to explain code, improve readability, and leave notes for developers. They are ignored by the interpreter

2. **Creating a Comment**: Comments begin with and continue to the end of the line.

3. **Single-line Comment**: A comment on a single line. Example:

```

# This is a single-line comment
```

Listing 30: single line comment

4. **Multi-line Comment**: Multiple single-line comments or a string with triple quotes.

```

    # This is a multi-line comment
    # that spans across multiple lines.
```

Listing 31: Multi line comment

or

```

"""
This is a multi-line comment
using triple quotes.
"""
```

Listing 32: Multi line example 2

3.2 Perform declaration

3.2.1 Definition of Key terms

1. **Declaration**: In Python, declaration refers to the process of defining a variable and assigning it a value. Unlike some other languages, Python does not require an explicit declaration of variable types; they are inferred dynamically based on the value assigned.

```

    x = 10  # Declaring a variable 'x' and assigning it the value 10


```

Listing 33: example of declaration

2. **Variables**: A variable is a symbolic name used to store data in Python. Variables hold values, which can be of various types, such as integers, strings, or lists. Variables can be reassigned to different values during the execution of a program

```

    name = "Alice"   # Variable 'name' stores the string "Alice"
    age = 25         # Variable 'age' stores the integer 25


```

Listing 34: Variable

In Python, you don't need to declare a variable type explicitly, as it's inferred dynamically based on the assigned value

3.2.2 Assigning values

1. **Single Value Assignment**: You can assign a single value to a variable.

```

    x = 10  # Assigning the integer 10 to variable 'x'
    name = "Alice"  # Assigning the string "Alice" to variable 'name'


```

Listing 35: Single value assignment

2. **Multiple Values Assignment**: You can assign multiple values to multiple variables in a single line. This is done using **comma-separated assignment**.

```

    x, y, z = 1, 2, 3  # Assigning 1 to 'x', 2 to 'y', and 3 to 'z'

    //alternatively
    a = b = c = 5  # Assigning the value 5 to 'a', 'b', and 'c'


```

Listing 36: Multi value assignment

3.2.3 Types of variables

1. **Local Variables**: A local variable is defined inside a function and can only be accessed within that function. It is not visible to other functions outside its scope

```

    def my_function():
        x = 10  # 'x' is a local variable
        print(x)

        my_function()


```

Listing 37: Local variable

2. **Global Variables**: A global variable is defined outside of any function and can be accessed throughout the entire program. It is visible to all functions.

```

    x = 10  # 'x' is a global variable

    def my_function():
        print(x)  # Accessing the global variable 'x'

    my_function()


```

Listing 38: Global variable

3. **Global Keyword**: The global keyword allows you to modify a global variable inside a function. Without it, a function will create a local variable instead of modifying the global one.

```

    x = 10  # Global variable

    def modify_global():
        global x  # Declare 'x' as global to modify it
        x = 20  # Modifying the global variable

    modify_global()
    print(x)  # Output: 20


```

Listing 39: Global keyword

3.3 Defferentiate data type

1. **Text**: In Python, text data is represented by the string type. Strings are sequences of characters enclosed in either single quotes (') or double quotes (").

```

    text = "Hello, World!"  # A string (text) data type


```

Listing 40: Example of Text

2. **Sequence**: Sequence types represent ordered collections of elements. Common sequence types include:

   (a) **list**: Ordered, mutable collections of items. Example

```

        fruits = ["apple", "banana", "cherry"]  # A list


```

Listing 41: Example of list

   (b) **tuple**: Ordered, immutable collections of items. Example

```

        coordinates = (4, 5)  # A tuple


```

Listing 42: Example of turple

3. **Mapping**: Mapping types store data in key-value pairs. The most common mapping type in Python is the **dictionary**.

```

    person = {"name": "John", "age": 30}  # A dictionary


```

Listing 43: Example of Mapping

4. **String**: String data type is used to store text. It is a sequence of characters, and strings are immutable in Python

```

    greeting = "Hello!"  # A string


```

Listing 44: Example of String

5. **Boolean**: The boolean data type represents truth values and has two possible values: **True** or **False**.

```

    is_active = True  # A boolean value


```

Listing 45: Example of Boolean

3.3.1 Define build-in data type

3.3.2 Numbers

Numeric data types in Python represent numbers. There are three types:

1. **int**: Integer numbers. Example

```
        age = 25   # An integer


```

Listing 46: Example of integer number

2. **float**: Floating-point numbers (decimals). Example:

```
        price = 19.99   # A floating-
    point number


```

Listing 47: Example of float number

3. **complex**: Complex numbers with a real and imaginary part. Example:

```
        complex_number = 3 + 4j   # A
    complex number



```

Listing 48: Example of complex number

4 Develop Python application

4.1 Use of operators

4.1.1 Arthimetic operators

```

    #Addition
    result = 5 + 3   # result = 8

    #Subtraction
    result = 5 - 3   # result = 2

    #Multiplication
    result = 5 * 3   # result = 15

    #Division
    result = 5 / 2   # result = 2.5

    #Modulus
    result = 5 % 2   # result = 1

    Exponentiation
    result = 2 ** 3   # result = 8

    #Floor Division: Divides and returns the
     largest integer less than or equal to
    the result.
    result = 5 // 2   # result = 2
```

Listing 49: Arthimetic operation

4.1.2 Comparison operators

```

    /* Equal (==): Checks if two values are
    equal. */
    result = 5 == 5   # result = True

    /* Not Equal (!=): Checks if two values
    are not equal. */
    result = 5 != 3   # result = True

    /* Greater Than (>): Checks if the left
    value is greater than the right. */
    result = 5 > 3   # result = True

    /* Less Than (<): Checks if the left
    value is less than the right. */
    result = 3 < 5   # result = True
```

Listing 50: Comparison operations

4.1.3 Logical Operator

1. **AND (and)**:Returns **True** if both conditions are true

```

    result = (5 > 3) and (7 > 4)   #
    result = True


```

Listing 51: And operation

2. **OR (or)**: Returns **True** if at least one condition is true.

```

    result = (5 > 3) or (2 > 4)   #
    result = True


```

Listing 52: OR operation

3. **NOT (not)**: Reverses the Boolean value (returns **True** if the condition is false, and **False** if the condition is true).

```

    result = not(5 > 3)   # result =
    False


```

Listing 53: not operation

4.1.4 Other operators

You can explore more operator such as Assignment, Betwise, Membership, and Identity operators online via different resources such as w3schools.

4.2 Determine collection of data

4.2.1 Definition of list

1. **Definition**: A list is a mutable, ordered collection of items. It can contain elements of different types (integers, strings, etc.).

```python
my_list = [1, 2, 3, "apple"]
```

Listing 54: Example of list

2. **Access Items**: Items in a list can be accessed using their index (starting from 0)

```python
print(my_list[0])  # Output: 1
```

Listing 55: Accessing item

3. **Negative Indexing**: You can access items from the end using negative indexing.

```python
print(my_list[-1])  # Output: "apple"
```

Listing 56: Negative indexing

4. **Range of Index**: Use slicing to access a range of items.

```python
print(my_list[1:3])  # Output: [2, 3]
```

Listing 57: Range of Index

5. **Change Item Value**: Modify an item by reassigning its value at a specific index.

```python
my_list[1] = 10  # Change item at index 1
```

Listing 58: Changing item value

6. **Loop**:

```python
Loop through the list using a for loop
```

Listing 59: Example of loop

7. **Check if Item Exists**: Use in to check if an item is in the list

```python
print("apple" in my_list)  # Output: True
```

Listing 60: Checking if item exist

8. **List Length**: Use **len()** to get the number of items in a list.

```python
print(len(my_list))  # Output: 4
```

Listing 61: Length of list

9. **Add Items**: Use **append()** to add an item to the end of the list or **insert()** to add at a specific index.

```python
my_list.append("banana")  # Adds "banana" at the end
my_list.insert(2, "orange")  # Adds "orange" at index 2
```

Listing 62: Adding item

10. **Remove Item**: Use remove() to remove an item by value or pop() to remove by index.

```python
    my_list.remove(10)  # Removes first occurrence of 10
my_list.pop(1)  # Removes item at index 1
```

Listing 63: Removing Item

11. **Copy a List**: Use copy() or slicing to create a copy of the list

```python
new_list = my_list.copy()  # Creates a copy of the list
```

Listing 64: Copy a list

12. **The List Constructor**: Create a list using the list() constructor.

```python
new_list = list((1, 2, 3))  # Creates a list from a tuple
```

Listing 65: Creating list

13. **List Methods**: Common list methods include append(), remove(), pop(), insert(), sort(), and reverse().

```
my_list.sort()  # Sorts the list in ascending order
```

Listing 66: Example of list method

4.2.2 Definition of Tuple

1. **Definition**: A **list** is a mutable, ordered collection of items. It can contain elements of different types (integers, strings, etc.).

```
my_list = [1, 2, 3, "apple"]
```

Listing 67: Example of tuple

2. **Access Items**: Items in a list can be accessed using their index (starting from 0).

```
print(my_list[0])  # Output: 1
```

Listing 68: Accessing Item

3. **Negative Indexing**: You can access items from the end using negative indexing

```
print(my_list[-1])  # Output: "apple"
```

Listing 69: Negative indexing

4. **Range of Index**: Use slicing to access a range of items

```
print(my_list[1:3])  # Output: [2, 3]
```

Listing 70: Range of Index

5. **Change Item Value**: Modify an item by reassigning its value at a specific index

```
my_list[1] = 10  # Change item at index 1
```

Listing 71: Change item value

6. **Loop**: Loop through the list using a for loop

```
for item in my_list:
print(item)
```

Listing 72: Example of loop

7. **Check if Item Exists**: Use **in** to check if an item is in the list

```
print("apple" in my_list)  # Output: True
```

Listing 73: Check if Item exist

8. **List Length**: Use **len()** to get the number of items in a list

```
print(len(my_list))  # Output: 4
```

Listing 74: Length of list

9. **Add Items**: Use **append()** to add an item to the end of the list or **insert()** to add at a specific index.

```
my_list.append("banana")  # Adds "banana" at the end
my_list.insert(2, "orange")  # Adds "orange" at index 2
```

Listing 75: Adding item

10. **Remove Item**: Use remove() to remove an item by value or pop() to remove by index.

```
my_list.remove(10)  # Removes first occurrence of 10
my_list.pop(1)  # Removes item at index 1
```

Listing 76: Removing item

11. **Copy a List**: Use copy() or slicing to create a copy of the list

```
new_list = my_list.copy()  # Creates a copy of the list
```

Listing 77: Copy a list

12. **The List Constructor**: Create a list using the list() constructor

```python
new_list = list((1, 2, 3))  # Creates a list from a tuple
```

Listing 78: creating list

13. **List Methods**: Common list methods include append(), remove(), pop(), insert(), sort(), and reverse().

```python
my_list.sort()  # Sorts the list in ascending order
```

Listing 79: list method

4.2.3 Definition of Set

1. **Definition**: A set is an unordered, mutable collection of unique elements (no duplicates).

```python
my_set = {1, 2, 3, "apple"}
```

Listing 80: Example of a set

2. **Access Items**: Sets do not support indexing, but you can loop through items.

```python
for item in my_set:
    print(item)
```

Listing 81: Access Item

3. **Add Items**: Use add() to add a single item or update() to add multiple items.

```python
my_set.add(4)  # Adds 4 to the set
my_set.update([5, 6])  # Adds multiple items
```

Listing 82: Add Item

4. **Get the Length of a Set**: Use **len()** to count items in the set

```python
print(len(my_set))  # Output: Number of elements
```

Listing 83: checking length

5. **Remove Item**: Use remove() (raises an error if item not found) or discard() (does not raise an error).

```python
my_set.remove(2)  # Removes 2 from the set
my_set.discard(10)  # No error if 10 is not in the set
```

Listing 84: Removing item

6. **Join Two Sets**: Use union() or update()

```python
set1 = {1, 2, 3}
set2 = {3, 4, 5}
new_set = set1.union(set2)  # Combines sets, removing duplicates
```

Listing 85: Joining two sets

7. **The Set Constructor**: Use set() to create a set from another iterable (e.g., list or tuple).

```python
my_set = set([1, 2, 3, 4])
```

Listing 86: Seting constructor

8. **Set Methods**: add(), remove(), discard(), pop(), union(), update(), clear(), difference(), intersection().

```python
set1 = {1, 2, 3}
set2 = {2, 3, 4}
intersection = set1.intersection(set2)  # Output: {2, 3}
```

Listing 87: Setting method

4.2.4 Definition of Dictionary

1. **Definition**: A dictionary is an unordered, mutable collection of key-value pairs. Each key must be unique.

```python
my_dict = {"name": "John", "age": 30, "city": "Kigali"}
```

Listing 88: Example of Dictionary

2. **Access Items**: Use keys to retrieve values.

```python
    print(my_dict["name"])  # Output: John
print(my_dict.get("age"))  # Output: 30
```

```
```

Listing 89: Accessing Item

3. **Change Values**: Modify a value by updating its key.

```python
    my_dict["age"] = 31  # Updates age to 31
```

Listing 90: Changes value

4. **Loop Through a Dictionary**: Iterate over keys, values, or both

```python
    for key in my_dict:
    print(key, my_dict[key])  # Prints key-value pairs
```

Listing 91: Looping through dictionary

5. **Check if a Key Exists**: Use in to check if a key is in the dictionary

```python
    print("name" in my_dict)  # Output: True
```

Listing 92: Checking if key exist

6. **Dictionary Length**: Use **len()** to count key-value pairs.

```python
    print(len(my_dict))  # Output: 3
```

Listing 93: Checking length

7. **Add Items**: Assign a new key-value pair

```python
    my_dict["country"] = "Rwanda"  # Adds a new key-value pair
```

Listing 94: Adding item

8. **Remove Items**: Use pop() (removes a key) or del

```python
    my_dict.pop("age")  # Removes key 'age'
del my_dict["city"]  # Removes key 'city'
```

Listing 95: Removing Item

9. **Copy Dictionaries**: Use copy() to create a copy

```python
    new_dict = my_dict.copy()
```

Listing 96: Copying dictionary

10. **Nested Dictionaries**: A dictionary inside another dictionary.

```python
    my_dict = {"student": {"name": "Alice", "age": 22}}
print(my_dict["student"]["name"])  # Output: Alice
```

Listing 97: Nested dictionary

11. **The dict() Constructor**: Create a dictionary using dict()

```python
    my_dict = dict(name="John", age=30, city="Kigali")
```

Listing 98: Using dict() constructor

12. **Dictionary Methods**: keys(), values(), items(), pop(), update(), clear()

```python
    print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])
```

Listing 99: Example of dictionary method

4.3 Understand condition statement

4.3.1 Explaination of Logical condition

Logical conditions are used in if-else statements and loops to control the flow of a program based on conditions.

1. **Equals (==)**: Checks if two values are equal

```python
    if 5 == 5:
    print("Equal")  # Output: Equal
```

Listing 100: Equal condition

2. **Not Equals (!=)**: Checks if two values are different

```python
    if 5 != 3:
    print("Not Equal")  # Output: Not Equal
```

Listing 101: Not equal condition

3. **Less Than (¡)**: Checks if the left value is smaller than the right.

```python
if 3 < 5:
print("Less than")  # Output: Less
than
```

Listing 102: Less than condition

4. **Greater Than (¿)**: Checks if the left value is larger than the right.

```python
if 7 > 4:
print("Greater than")  # Output:
Greater than
```

Listing 103: greater than condition

These conditions are commonly used in decision-making structures like if, elif, else to execute different blocks of code based on conditions.

4.3.2 Eplaination of IF statemnt

The if statement is used for decision-making in Python. It executes a block of code if a specified condition is True.

1. **Indentation**: Python uses indentation (spaces or tabs) to define blocks of code inside the **if, elif**, and **else** statements.

```python
if True:
    print("This is inside the if
block")  # Indented correctly
```

Listing 104: Using Identation

2. **IF Statement (if)**: Executes a block of code if the condition is True

```python
age = 18
if age >= 18:
    print("You are an adult")  #
Output: You are an adult
```

Listing 105: example of If statement

3. **ELIF Statement (elif)**: Checks another condition if the previous **if** condition was **False**.

```python
age = 16
if age > 18:
    print("Adult")
elif age == 16:
    print("You are 16")  # Output:
You are 16
```

Listing 106: Example of ELIF statement

4. **ELSE Statement (else)**:

```python
age = 15
if age > 18:
    print("Adult")
elif age == 16:
    print("You are 16")
else:
    print("You are a minor")  #
Output: You are a minor
```

Listing 107: Example of ELSE statement

5. **Shorthand IF**: A single-line if statement

```python
if 5 > 3: print("5 is greater than 3
")  # Output: 5 is greater than 3
```

Listing 108: Shorthand IF condition

6. **Shorthand IF...ELSE**: A compact way to write if-else in one line

```python
print("Adult") if age >= 18 else
print("Minor")  # Output: Minor
```

Listing 109: Shorthand IF..ELSE

The if-elif-else structure is essential for handling conditional logic in Python!

4.4 Identify other functions and classes

4.4.1 Use of Looping

Looping is used to execute a block of code multiple times until a specific condition is met.

1. **For Loop**: Iterates over a sequence (list, tuple, dictionary, string, etc.).

```python
for i in range(5):
    print(i)  # Output: 0 1 2 3 4
```

Listing 110: example of for looop

2. **While Loop**: Repeats as long as the condition is **True**.

```python
x = 0
while x < 5:
    print(x)
    x += 1  # Output: 0 1 2 3 4
```

Listing 111: example of while loop

3. **Continue Statement**: Skips the rest of the current loop iteration and moves to the next.

```python
for i in range(5):
if i == 2:
    continue  # Skips 2
print(i)  # Output: 0 1 3 4
```

Listing 112: continue statement

4. **Break Statement**: Exits the loop completely when a condition is met

```python
for i in range(5):
    if i == 3:
        break  # Stops at 3
    print(i)  # Output: 0 1 2
```

Listing 113: Break statement

4.4.2 Definition of Functions

A function is a block of reusable code that performs a specific task. Functions help in code organization and reusability.

1. **Creating a Function**: Use the def keyword to define a function.

```python
def greet():
    print("Hello, Welcome!")
```

Listing 114: Defining a function

2. **Calling a Function**: Execute a function by using its name followed by parentheses ().

```python
greet()  # Output: Hello, Welcome!
```

Listing 115: calling a function

3. **Arguments**: Functions can take inputs (parameters) to perform operations

```python
def add(a, b):
    return a + b
print(add(3, 5))  # Output: 8
```

Listing 116: Arguments of function

4. **Default Parameter Value**: If no value is provided, the default is used

```python
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet()  # Output: Hello, Guest!
greet("John")  # Output: Hello, John
!
```

Listing 117: Default paramater value

5. **Passing a List as an Argument**: Functions can accept lists as arguments.

```python
def print_list(items):
    for item in items:
        print(item)
print_list(["Apple", "Banana", "
Cherry"])
```

Listing 118: Passing a list argument

6. **Lambda (Anonymous Function)**: A small, one-line function using **lambda**

```python
square = lambda x: x * x
print(square(4))  # Output: 16
```

Listing 119: Using Lambda

7. **Arrays**: Arrays in Python are implemented using lists

```python
numbers = [1, 2, 3, 4, 5]
print(numbers[2])  # Output: 3
```

Listing 120: Example of Arrays

4.4.3 Definition of Classes/Objects

A **class** is a blueprint for creating objects, and an **object** is an instance of a class. Classes encapsulate data and functions into a single entity.

1. **Create Class**: Define a class using the class keyword.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Listing 121: Defining a class

2. **Create Object**: Instantiate a class to create an object

```python
person1 = Person("John", 30)
```

Listing 122: Creating Object

3. **Object Methods**: Functions defined inside a class to perform operations on its objects.

```python
class Person:
def greet(self):
    print(f"Hello, {self.name}")
person1 = Person("John", 30)
person1.greet()  # Output: Hello,
John
```

Listing 123: Object method

4. **The *self* Parameter**: **self** refers to the instance of the class, allowing access to its attributes and methods.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Listing 124: Using self parameter

5. **Modify Object Property**: Change the value of an object's property

```python
person1.age = 31  # Modify the age
of person1
```

Listing 125: Modifying object property

6. **Delete Object Property**: Use **del** to delete an object's property

```python
del person1.age  # Deletes the 'age'
 attribute of person1
```

Listing 126: Deleting object property

7. **Delete Object**: Use **del** to delete the object itself.

```python
del person1  # Deletes the object
person1
```

Listing 127: Deleting object

8. **Pass Statement**: A placeholder for an empty code block (does nothing).

```python
class Person:
pass  # Empty class
```

Listing 128: example of Pass statement

9. **Inheritance**: Inheritance allows a class (child class) to inherit properties and methods from another class (parent class).

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

dog = Dog()
dog.speak()  # Output: Animal speaks
dog.bark()  # Output: Dog barks
```

Listing 129: example of inheritance

10. **Iterators**: Iterators are objects that allow traversing through all the elements in a collection. A class can implement **__iter__()** and **__next__()** to make it an iterator.

```python
class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

counter = Counter(1, 3)
for number in counter:
    print(number)  # Output: 1 2 3
```

Listing 130: Example of iterator

4.4.4 Definition of other tools

1. **Python Scope**: Scope refers to the region of a program where a variable or name is recognized. In Python, variables can have global or local scope. **Local Scope**: Variables defined within a function. **Global Scope**: Variables defined outside any function.

```python
    x = 10  # Global scope

    def my_function():
        y = 5  # Local scope
        print(x, y)
```

Listing 131: global scope

2. **Python Modules**: A module is a file containing Python code, which can define functions, classes, and variables. You can import a module using the import statement.

```python
    import math
print(math.sqrt(16))  # Output: 4.0
```

Listing 132: Example of a Module

3. **Python Dates**: The **datetime** module allows working with dates and times

```python
    import datetime
    current_date = datetime.date.today()
    print(current_date)  # Output:
    current date (e.g., 2025-02-04)
```

Listing 133: example of date module

4. **Python JSON**: The **json** module is used to parse JSON (JavaScript Object Notation) data into Python objects and vice versa.

```python
    import json

    # Convert a dictionary to JSON
    data = {"name": "John", "age": 30}
    json_data = json.dumps(data)
    print(json_data)  # Output: {"name":
     "John", "age": 30}
```

Listing 134: json module

5. **Python PIP**: PIP is the package installer for Python. It is used to install libraries and packages from the Python Package Index (PyPI).

```
    pip install requests  # Install the
    'requests' library
```

Listing 135: Installing PIP

6. **Python Try...Except**: Used for handling exceptions or errors in code, allowing the program to continue running even when an error occurs.

```python
    try:
    x = 10 / 0  # Division by zero

    except ZeroDivisionError:
        print("Cannot divide by zero!")
  # Output: Cannot divide by zero!
```

Listing 136: example of Try .. Except

7. **User Input**: The **input()** function allows the user to enter data, which is returned as a string.

```python
    name = input("Enter your name: ")
    print(f"Hello, {name}!")
```

Listing 137: Allowing User input

5 Develop python scripting

5.1 Perform file handling

5.1.1 Practice to read file

Python provides built-in functions to handle files. You can open, read, write, and close files using various methods.

1. **Open a File**: Use the **open()** function to open a file. You can specify the mode (read, write, etc.) when opening a file.

```python
    file = open("example.txt", "r")  #
    Open file in read mode
```

Listing 138: Opening file

2. **Read a File**: Once the file is open, you can read its contents using methods like: **read()**: Reads the entire file as a string. **readline()**: Reads the next line from the file. **readlines()**: Reads all lines of the file and returns them as a list.

```python
    content = file.read()  # Read the
    entire file
    print(content)
```

```

```

Listing 139: Reading file

3. **File Permissions**: When opening a file, you can specify its mode. The most common modes are: **"r"**: Read (default mode, opens the file for reading). **"w"**: Write (opens the file for writing, creates a new file or overwrites an existing one). **"a"**: Append (opens the file for writing, appending data at the end). **"rb", "wb"**, etc.: Read or write in binary mode (useful for binary files)

```

    # Opening file in read-only mode
    file = open("example.txt", "r")

    # Trying to open file with restricted permission (write access) in read mode
    try:
        file = open("restricted_file.txt", "r")  # Error occurs if no read permission
    except PermissionError:
        print("Permission denied to read the file.")


```

Listing 140: File permission

4. **Closing the File**: It's important to close the file after you're done with it to free up system resources

```

    file.close()  # Close the file after reading


```

Listing 141: closing file

5. **Using with to Handle Files**: It is recommended to use the with statement to handle files, as it automatically closes the file when done

```

    with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # No need to manually close the file


```

Listing 142: Using file

6. **Example Code for Reading a File**:

```

    # Open the file in read mode
    with open("example.txt", "r") as file:
        content = file.read()  # Read the entire content
        print(content)  # Display file contents


```

Listing 143: Reading file

By using these methods, you can manage files efficiently and ensure proper permission handling in your Python programs.

5.1.2 Practice to read file

Python allows you to create new files and write to both new and existing files using different file modes. Here's how you can do it:

1. **Create a New File**: To create a new file, you can use the **"w"** (write) mode. If the file already exists, this mode will overwrite the existing content.

   Example to create a new file and write to it:

```

    with open("new_file.txt", "w") as file:
    file.write("This is a new file.\n")
    file.write("It contains some text.\n")


```

Listing 144: Creating new file

   This will create a file named **new_file.txt** in the current directory and write two lines of text into it. If the file already exists, it will be overwritten.

2. **Write to an Existing File**: If you want to add content to an existing file without overwriting it, you can open the file in **"a"** (append) mode.

   Example to write to an existing file (without overwriting):

```

    with open("existing_file.txt", "a") as file:
    file.write("Adding more content to the file.\n")
    file.write("This will append new data.\n")


```

Listing 145: Writing to an existing file

3. **Example of Creating and Writing to Files**:

```

    # Creating and writing to a new file
    with open("new_file.txt", "w") as file:
        file.write("This is the first line of the file.\n")
        file.write("This is the second line.\n")

    # Writing to an existing file (appending)
```

```
with open("existing_file.txt", "a")
as file:
    file.write("This is a new line
added to the file.\n")


```

Listing 146: Creating and write file

5.1.3 Pactice to delete file

Python provides several ways to delete files and folders using the os and os.path modules.

1. **Remove File**: You can remove a file using the os.remove() function. It deletes the specified file.

   Example to remove a file:

```

import os

# Remove a file
os.remove("file_to_delete.txt")  #
This will delete the file


```

Listing 147: Removing file

   If the file doesn't exist, this will raise a FileNotFoundError.

2. **Check if a File Exists**: Before deleting a file, you might want to check if it exists. You can do this using the **os.path.exists()** function.

   Example to check if a file exists before removing it:

```

import os

# Check if the file exists
if os.path.exists("file_to_delete.
txt"):
    os.remove("file_to_delete.txt")
    print("File deleted successfully
.")
else:
    print("The file does not exist."
)


```

Listing 148: Checking if file exist before

3. **Delete Folder**: To delete an empty folder, you can use **os.rmdir()**. If the folder contains files or other directories, you need to use **shutil.rmtree()** to remove it along with all its contents.

```

import os

# Remove an empty directory
os.rmdir("empty_folder")


```

```
#Example to remove a non-empty
folder:
import shutil

# Remove a folder and all its
contents
shutil.rmtree("folder_to_delete")


```

Listing 149: Deleting file

4. **Full Example for File and Folder Deletion**:

```

import os
import shutil

# Remove file if it exists
file_path = "file_to_delete.txt"
if os.path.exists(file_path):
    os.remove(file_path)
    print(f"{file_path} deleted
successfully.")
else:
    print(f"{file_path} does not
exist.")

# Remove an empty folder if it
exists
folder_path = "empty_folder"
if os.path.exists(folder_path) and
os.path.isdir(folder_path):
    os.rmdir(folder_path)
    print(f"Folder {folder_path}
deleted successfully.")
else:
    print(f"{folder_path} does not
exist or is not a folder.")

# Remove a non-empty folder
non_empty_folder = "folder_to_delete
"
if os.path.exists(non_empty_folder)
and os.path.isdir(non_empty_folder):
    shutil.rmtree(non_empty_folder)
    print(f"Non-empty folder {
non_empty_folder} deleted
successfully.")
else:
    print(f"{non_empty_folder} does
not exist or is not a folder.")


```

Listing 150: full example

5.2 Determine Python Libraries

Here are brief explanations of some essential Python libraries commonly used for data manipulation, scientific computing, and machine learning:

1. **Numpy**: Numpy is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on them. **Common Use**: Numerical computations, linear algebra, data manipulation, and matrix operations

```python

import numpy as np
arr = np.array([1, 2, 3])
print(np.sum(arr))  # Output: 6


```

Listing 151: Using Numerical

2. **Pandas**: Pandas is a data manipulation and analysis library. It provides data structures like DataFrame and Series to handle and analyze structured data easily.

   Common Use: Data manipulation, cleaning, and analysis. It is ideal for handling tabular data (rows and columns).

```python

import pandas as pd
data = {'Name': ['John', 'Anna'], 'Age': [28, 24]}
df = pd.DataFrame(data)
print(df)


```

Listing 152: Example of using pandas

3. **Matplotlib**: Matplotlib is a plotting library used to create static, interactive, and animated visualizations in Python. It is widely used for data visualization.

   Common Use: Plotting graphs such as line charts, bar charts, histograms, scatter plots, etc.

```python

import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)
plt.show()  # Display line plot


```

Listing 153: example of matplotlib

4. **SciPy**: SciPy is an open-source library used for scientific and technical computing. It builds on Numpy and provides additional functionality for optimization, integration, interpolation, eigenvalue problems, and other advanced math and statistics operations.

   Common Use: Scientific computing tasks such as optimization, signal processing, and statistics.

```python

from scipy import stats
data = [2, 3, 4, 5, 6]
mean = stats.tmean(data)
print(mean)  # Output: 4.0


```

Listing 154: EXample of scipy

5. **Scikit-Learn**: Scikit-Learn is a machine learning library that provides simple and efficient tools for data mining and data analysis. It supports various machine learning algorithms for classification, regression, clustering, dimensionality reduction, and more.

   Common Use: Building and deploying machine learning models (e.g., decision trees, linear regression, k-means).

```python

from sklearn.linear_model import LinearRegression
model = LinearRegression()
X = [[1], [2], [3], [4]]
y = [1, 2, 3, 4]
model.fit(X, y)
print(model.predict([[5]]))  # Predict for input 5


```

Listing 155: example of scikit-learn

5.3 Interact with database

5.3.1 Python Mysql commands

Below are the key commands and steps for interacting with a MySQL database using Python. The mysql-connector-python library is commonly used to connect to MySQL databases from Python.

1. **Install MySQL Connector**: To interact with MySQL in Python, you need to install the MySQL connector library. You can install it using **pip**:

```

pip install mysql-connector-python


```

Listing 156: Installing mysql connector

2. **Test MySQL Connector**: Ensure the MySQL connector is installed correctly by testing the connection.

```python

import mysql.connector

try:
    connection = mysql.connector.connect(
        host="localhost",
        user="root",  # Replace with your MySQL username
        password="your_password"  # Replace with your MySQL password
    )
    if connection.is_connected():
            print("Connected to MySQL
    server")
    except mysql.connector.Error as err:
        print(f"Error: {err}")
```

Listing 157: testing connector

3. **Create Connection**: You can create a connection to the MySQL server by providing the required parameters such as host, user, password, and database name.

```
    connection = mysql.connector.connect
    (
    host="localhost",
    user="root",
    password="your_password",
    database="test_db"  # Optional, if
    you want to connect to a specific
    database)
```

Listing 158: Creating connection

4. **Create Database**: To create a new database:

```
    cursor = connection.cursor()
    cursor.execute("CREATE DATABASE
    my_database")
    cursor.close()
```

Listing 159: Creating database

5. **Create Table**: You can create a table inside a database.

```
    cursor = connection.cursor()
    cursor.execute("""
        CREATE TABLE users (
            id INT AUTO_INCREMENT
    PRIMARY KEY,
            name VARCHAR(100),
            age INT
        )
    """)
    cursor.close()
```

Listing 160: Creating a table

6. **Insert Data**: Insert data into a table.

```
    cursor = connection.cursor()
    cursor.execute("INSERT INTO users (
    name, age) VALUES (%s, %s)", ("John
    Doe", 30))
    connection.commit()  # Commit
    changes to the database
    cursor.close()
```

Listing 161: Inserting data

7. **Select Data**: To retrieve data from a table.

```
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM users"
    )
    result = cursor.fetchall()
    for row in result:
        print(row)
    cursor.close()
```

Listing 162: Selecting data

8. **Delete Data**: To delete data from a table.

```
    cursor = connection.cursor()
    cursor.execute("DELETE FROM users
    WHERE id = %s", (1,))
    connection.commit()
    cursor.close()
```

Listing 163: Deleting data

9. **Where Condition**: Using a WHERE condition to filter data.

```
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM users
    WHERE age > %s", (25,))
    result = cursor.fetchall()
    for row in result:
        print(row)
    cursor.close()
```

Listing 164: WHERE condition

10. **Order By**: To order the results by a column.

```
    cursor = connection.cursor()
    cursor.execute("SELECT * FROM users
    ORDER BY name ASC")
    result = cursor.fetchall()
    for row in result:
        print(row)
    cursor.close()
```

Listing 165: Order by condition

11. **Drop Table**: To drop (delete) a table.

```
    cursor = connection.cursor()
    cursor.execute("DROP TABLE IF EXISTS
     users")
    cursor.close()
```

Listing 166: Droping table

12. **Update Data**: To update an existing record in the table.

```
cursor = connection.cursor()
cursor.execute("UPDATE users SET age = %s WHERE name = %s", (35, "John Doe"))
connection.commit()  # Commit changes to the database
cursor.close()
```

Listing 167: Updating table

13. **Limit**: Use LIMIT to restrict the number of rows returned.

```
cursor = connection.cursor()
cursor.execute("SELECT * FROM users LIMIT 2")
result = cursor.fetchall()
for row in result:
    print(row)
cursor.close()
```

Listing 168: LIMIT condition

14. **Join**: Perform a join between two tables.

```
cursor = connection.cursor()
cursor.execute("""
    SELECT orders.id, users.name, orders.amount
    FROM orders
    JOIN users ON orders.user_id = users.id
""")
result = cursor.fetchall()
for row in result:
    print(row)
cursor.close()
```

Listing 169: Join operation

15. Make sure to close the cursor and the connection after executing operations::

```
cursor.close()
connection.close()
```

Listing 170: closing connection

These commands allow you to perform various operations like creating databases and tables, inserting, updating, deleting, and selecting data, as well as more advanced queries like using conditions and joins.

5.3.2 MongoDB

MongoDB is a NoSQL database that stores data in flexible, JSON-like documents. Below are the key commands and operations for working with MongoDB in Python using the **pymongo** library.

1. **Install PyMongo Driver**: To interact with MongoDB using Python, install the **pymongo** library

```
pip install pymongo
```

Listing 171: Installing pymango

2. **Create Connection to MongoDB**: Connect to a local MongoDB server:

```
import pymongo

client = pymongo.MongoClient("mongodb://localhost:27017/")
```

Listing 172: creating connection

You can also connect to a remote MongoDB server by replacing "**localhost:27017**" with the server's address

3. **Create Database**: MongoDB does not require explicitly creating a database. It is created when you first store data in it.

```
db = client["my_database"]
```

Listing 173: connecting to remote mongodb

4. **Create Collection (Equivalent to Table in SQL)**: Collections are created automatically when you insert the first document (record).

```
collection = db["users"]
```

Listing 174: creating collection

5. **Insert Data**

   (a) **Single document**:

```
    user = {"name": "John Doe", "age": 30}
    collection.insert_one(user)
```

Listing 175: single document

(b) **Multiple documents**:

```
    users = [
    {"name": "Alice", "age":
25},
    {"name": "Bob", "age": 28},
]
collection.insert_many(users)
```

Listing 176: multiple document

6. **Select Data (Retrieve Documents)**:

(a) **Find all documents**:

```
    for user in collection.find
():
        print(user)
```

Listing 177: Retrieving document

(b) **Find specific fields**:

```
    for user in collection.find
({}, {"_id": 0, "name": 1}):
        print(user)
```

Listing 178: finding specific field

7. **Delete Data**

(a) **Delete one document**:

```
    collection.delete_one({"
name": "John Doe"})
```

Listing 179: Deleting one document

(b) **Delete multiple documents:**

```
    collection.delete_many({"
age": {"$lt": 30}})  # Delete
users younger than 30
```

Listing 180: Delete multiple documents

8. **Where Condition (Query with Filters)**: Find users older than 25

```
for user in collection.find({"age":
{"$gt": 25}}):
    print(user)
```

Listing 181: Using WHERE condition

9. **Order By (Sorting Data)**: Sort users by age in ascending order

```
    for user in collection.find().
sort("age", 1):  # 1 for ascending,
-1 for descending
    print(user)
```

Listing 182: using ORDER BY condition

10. **Drop Collection (Equivalent to Drop Table in SQL)**

```
collection.drop()  # Deletes the
entire "users" collection
```

Listing 183: Droping collection

11. **Update Data**

(a) **Update a single document**:

```
    collection.update_one({"
name": "Alice"}, {"$set": {"age
": 26}})
```

Listing 184: updating single document

(b) **Update multiple documents**:

```
    collection.update_many({"
age": {"$lt": 30}}, {"$set": {"
status": "young"}})
```

Listing 185: Updating multiple document

12. **Limit Results**: Limit the number of retrieved documents

```
for user in collection.find().limit
(2):
    print(user)
```

Listing 186: Using LIMIT condition

13. **Join (Aggregation in MongoDB)**: MongoDB does not support SQL-style joins, but you can achieve similar results using aggregation:

```
    db.orders.insert_many([
    {"user_id": 1, "product": "Laptop"},
    {"user_id": 2, "product": "Phone"}
])

pipeline = [
    {
        "$lookup": {
            "from": "users",
            "localField": "user_id",
            "foreignField": "_id",
            "as": "user_info"
        }
    }
]

for order in db.orders.aggregate(pipeline):
    print(order)
```

Listing 187: Using join operation

6 Develop AI based applications

6.1 Introduce AI

6.1.1 Definitiona of Key terms:

1. **What is AI**
   Artificial Intelligence (AI) refers to the simulation of human intelligence in machines that are programmed to think, learn, and solve problems. AI encompasses various subfields, including machine learning, natural language processing, computer vision, and robotics. It enables systems to perform tasks such as decision-making, pattern recognition, and automation.

2. **History of AI**
   The history of AI dates back to the 1950s when Alan Turing introduced the concept of machine intelligence. The field formally began in 1956 at the Dartmouth Conference, where researchers explored ways to develop machines that could simulate human intelligence. Over the decades, AI has evolved through different phases: early symbolic AI (1950s-1970s), expert systems (1980s), statistical machine learning (1990s-2000s), and deep learning (2010s-present). Today, AI is driven by advancements in big data, computing power, and neural networks[1].

3. **Benefits of AI**
   AI offers numerous benefits across various industries, including:

   (a) **Automation**: AI automates repetitive tasks, improving efficiency and reducing human effort.

   (b) **Improved Decision-Making**: AI-powered analytics enhance data-driven decision-making in fields like healthcare and finance.

[1]You can read more about the history of AI by clicking here

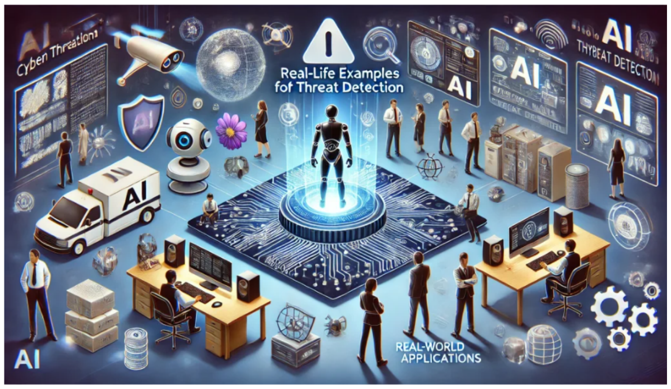

Figure 1: AI in every sector of life

Figure 2: AI atmosphere

   (c) **Personalization**: AI enables personalized recommendations in services like e-commerce and entertainment.

   (d) **Healthcare Advancements**: AI assists in medical diagnosis, drug discovery, and robotic surgeries.

   (e) **Enhanced Security**: AI-driven surveillance and cybersecurity systems detect threats in real time.

   (f) **Economic Growth**: AI boosts productivity, innovation, and job opportunities in technology-driven sectors.

   AI continues to revolutionize industries by increasing efficiency, reducing costs, and enabling new technological advancements.

6.1.2 Types of AI

1. **Weak AI (Narrow AI)**
   Weak AI, also known as Narrow AI, refers to artificial intelligence systems that are designed to perform specific tasks without possessing general intelligence. These AI models operate under predefined rules and cannot think, reason, or understand beyond their programming. Examples include:

   (a) Virtual Assistants (e.g., Siri, Alexa, Google Assistant)

   (b) Recommendation Systems (e.g., Netflix, YouTube, Amazon)

   (c) Autonomous Vehicles (self-driving cars use AI for navigation)
   (d) Chatbots (used in customer service)

2. **Strong AI (General AI)**
   Strong AI, also known as Artificial General Intelligence (AGI), refers to AI systems that possess human-like cognitive abilities, including reasoning, problem-solving, and learning across multiple domains. Unlike Weak AI, Strong AI can understand, learn, and adapt to new situations without specific programming. AGI remains theoretical, as no system has yet achieved true human-level intelligence. If developed, Strong AI could:
   (a) Perform any intellectual task that a human can
   (b) Exhibit reasoning and self-awareness
   (c) Adapt to new problems without retraining

   While Weak AI is widely used today, Strong AI is still a subject of research and remains a future goal in AI development.

6.1.3 Real life example of AI

1. **Self-Driving Cars**: AI-powered autonomous vehicles use computer vision, machine learning, and sensor data (LiDAR, cameras) to navigate roads, detect obstacles, and make real-time driving decisions. Example: **Tesla Autopilot, Waymo**

Figure 3: Selving Driving Car

2. **Navigation Systems**: AI enhances GPS-based navigation by analyzing traffic patterns, road conditions, and user preferences to provide optimal routes. Example: **Google Maps, Waze**

3. **Chatbots** : AI-driven chatbots assist users in customer service, answering queries, and automating conversations using natural language processing (NLP). Example: **ChatGPT, Siri, Alexa, Google Assistant, banking chatbots**

Figure 4: Navigation Systems

Figure 5: Chatbot

4. **Human vs. Computer Games**: AI competes against humans in video games, learning strategies and adapting gameplay. Example: **DeepMind's AlphaGo (beat human Go champions), OpenAI Five (Dota 2), IBM Deep Blue (chess vs. Garry Kasparov)**

5. **Sophia Robot**: A humanoid robot developed by Hanson Robotics that uses AI for facial recognition, speech processing, and human-like interactions. Sophia can engage in conversations and has been granted honorary citizenship in Saudi Arabia.

6. **Turing Machine**: Proposed by **Alan Turing**, this theoretical AI model laid the foundation for modern computing and artificial intelligence by demonstrating how machines could process and simulate human decision-making.

7. **More AI Examples in Daily Life**
   (a) **Facial Recognition**: Used in security systems, smartphones (Face ID).
   (b) **Healthcare AI**: AI detects diseases, assists in diagnosis (IBM Watson, AI-powered X-ray analysis).
   (c) **Financial AI**: AI predicts stock markets, detects fraud (AI trading algorithms, fraud detection in banks).

Figure 6: facial and Key points detection

Figure 7: Face detection

(d) **E-commerce AI**: Personalized recommendations (Amazon, Netflix).

(e) **Smart Home Devices**: AI-powered home automation (Google Nest, Amazon Echo).

AI continues to revolutionize industries by improving efficiency, accuracy, and automation across various sectors.

6.1.4 Future of AI

1. **Multary Bots (Advanced AI Assistants)**: AI-powered bots will become more advanced, capable of performing complex tasks across multiple fields, from military operations to scientific research and daily assistance. These bots will integrate **natural language processing (NLP), robotics, and deep learning** for seamless human interaction.

2. **The Perfect Lawyer** : AI will revolutionize the legal industry by analyzing vast amounts of case law, automating legal research, predicting case outcomes, and assisting in contract drafting. AI-powered systems will provide legal advice, reducing the need for human intervention in routine legal matters.

Figure 8: Keypoints detection

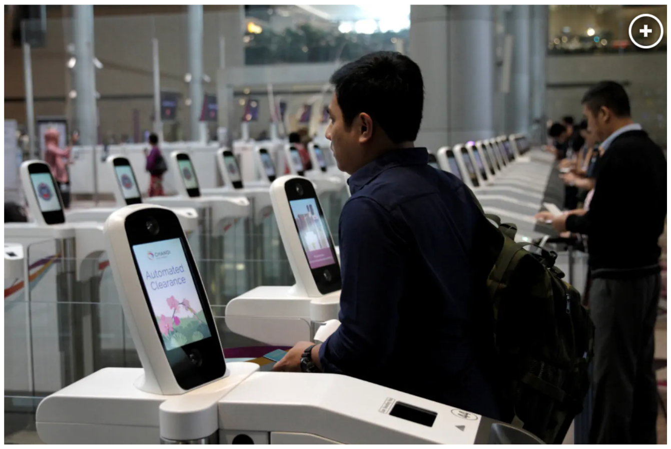

Figure 9: Changi Airport Singapore facial recognition system

3. **AI in Music**: AI will compose music, enhance sound production, and personalize user experiences. Tools like **AI-generated lyrics, virtual musicians, and real-time music customization** will become mainstream. AI-powered algorithms will also help detect copyright infringements. Example: **AIVA (AI music composer), OpenAI's Jukebox**.

4. **AI in Business**: Businesses will leverage AI for **decision-making, automation, fraud detection, personalized marketing, and supply chain optimization**. AI-driven chatbots and virtual assistants will enhance customer service, while predictive analytics will improve financial forecasting.

5. **AI in Healthcare**: AI will transform healthcare through AI-powered diagnosis, robotic surgeries, drug discovery, and personalized treatment plans. AI-driven systems will detect diseases (like cancer and Alzheimer's) at early stages, leading to better outcomes and lower healthcare costs. Example: Google's DeepMind, IBM Watson Health.

The future of AI is set to revolutionize multiple industries, making systems smarter, more efficient, and

Figure 10: AI in Health care

deeply integrated into human life.

6.2 Implement Machine Learning

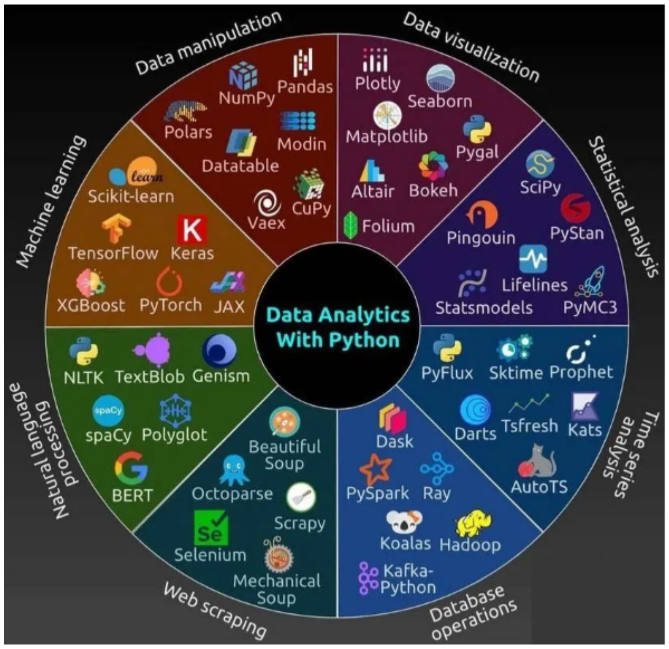

Figure 11: Essential Python Libraries for Data Analytics

6.2.1 Definition of Key Terms:

1. **Data Splitting**: The process of dividing a dataset into training, validation, and testing sets to evaluate machine learning models. Common splits include **80-20 (train-test) or 70-20-10 (train-validation-test)**.

2. **Numerical Data**: Data represented in numbers, used for calculations and analysis. It includes discrete data (e.g., number of students) and continuous data (e.g., height, weight, temperature).

3. **Categorical Data**: Data that represents distinct groups or categories, often non-numeric. Example: Gender (Male/Female), Colors (Red, Blue, Green), Types of animals (Dog, Cat, Bird).

4. **Ordinal Data**: A type of categorical data with a meaningful order but without fixed intervals between values. Example: Education levels (High School ¡ Bachelor's ¡ Master's ¡ PhD), Customer satisfaction ratings (Low ¡ Medium ¡ High).

5. **Percentile** : A statistical measure indicating the relative position of a value within a dataset. Example: A student scoring in the 90th percentile performed better than 90% of others.

6. **Data Distribution**: The way data values are spread or arranged in a dataset. Common distributions include normal distribution, skewed distribution, and uniform distribution.

7. **Algorithm** : A set of rules or instructions used by a computer to solve a problem or perform a task. Machine learning algorithms include decision trees, neural networks, and support vector machines.

8. **Model** : A trained machine learning system that learns patterns from data to make predictions or classifications. Example: A spam filter model that detects spam emails.

9. **Scatter Plot**: A graphical representation of two numerical variables, showing their relationship. It helps visualize patterns, trends, and correlations.

10. **Importance of Machine Learning**: Machine learning automates decision-making, improves efficiency, enhances predictions, and enables personalized experiences in industries like healthcare, finance, and business.

11. **Data Slicing**: The process of dividing data into smaller subsets based on specific conditions or attributes for better analysis and training. Example: Filtering customer data by region or age group.

These concepts are fundamental in data science and machine learning, helping in better data analysis, model building, and decision-making.

6.2.2 Define Variables and data

1. **Predictor Variable (Independent Variable)**: A variable used to predict or explain the response variable. It is also called an input variable or feature in machine learning. Example: In predicting house prices, features like square footage, number of rooms, and location are predictor variables.

2. **Response Variable (Dependent Variable)**: The outcome or target variable that the model tries to predict based on predictor variables. Example: In house price prediction, the price of the house is the response variable.

3. **Training Data**: A subset of the dataset used to train a machine learning model. It helps the model learn patterns and relationships between predictor and response variable

4. **Testing Data**: A separate subset of the dataset used to evaluate the model's performance after training. It ensures that the model generalizes well to new, unseen data

5. **Data Scraping (Web Scraping)**: The process of extracting large amounts of data from websites using automated tools or scripts. It is commonly used in data collection for machine learning, research, and analysis. Example: Scraping product prices from e-commerce sites.

These concepts are essential in building and evaluating machine learning models effectively.

6.2.3 Machine Learning processes

Machine learning (ML) is a subset of artificial intelligence (AI) that enables computers to learn from data without explicit programming, allowing them to improve their performance and make predictions or decisions over time[2]

1. **Define Objective**: Identify the problem to be solved and the goal of the machine learning model. Example: Predicting customer churn, detecting fraud, or classifying emails as spam or not.

2. **Data Gathering**: Collect relevant data from sources like databases, APIs, sensors, or web scraping. High-quality data is crucial for model performance.

3. **Preparing Data**: Clean and preprocess the data by handling missing values, removing duplicates, normalizing numerical data, encoding categorical variables, and splitting data into training and testing sets.

4. **Data Exploration**: Analyze data using visualizations and statistical techniques to understand distributions, relationships, and potential biases before training the model. Example: Using scatter plots, histograms, or correlation matrices.

5. **Building Model**: Select an appropriate machine learning algorithm (e.g., decision trees, neural networks, support vector machines) and train the model using the training dataset.

6. **Model Evaluation**: Assess the model's performance using metrics like accuracy, precision, recall, F1-score, and RMSE (Root Mean Squared Error) on the testing dataset.

7. **Predictions** : Use the trained model to make predictions on new, unseen data and apply it to real-world scenarios. Example: Recommending movies based on user preferences.

This structured process ensures the development of accurate, reliable, and efficient machine learning models.

6.2.4 Types of Machine Learning

Each type of machine learning serves different applications, helping to solve a variety of real-world problems.

1. **Supervised Learning**: The model learns from labeled data, where both input (features) and output (target variable) are provided.
   (a) **Example**: Spam email detection (emails labeled as spam or not spam), house price prediction.
   (b) **Algorithms**: Linear regression, decision trees, support vector machines (SVM), neural networks.

2. **Unsupervised Learning**: The model learns patterns from unlabeled data without predefined outputs. It identifies structures, clusters, or associations within the data.
   (a) **Example**: Customer segmentation in marketing, anomaly detection in fraud detection.
   (b) **Algorithms**: K-Means clustering, hierarchical clustering, principal component analysis (PCA), autoencoders.

3. **Reinforcement Learning**: The model learns by interacting with an environment and receiving rewards or penalties based on actions taken
   (a) **Example**: Self-driving cars, game-playing AI like AlphaGo, robotic automation.
   (b) **Algorithms**: Q-learning, deep Q-networks (DQN), policy gradient methods.

6.2.5 Machine Learning Algorithm

Machine learning algorithms are used to make predictions, classify data, and identify patterns. Below is an in-depth explanation of key algorithms: (consider figure [13], [14], and [15])

1. **Linear Regression**

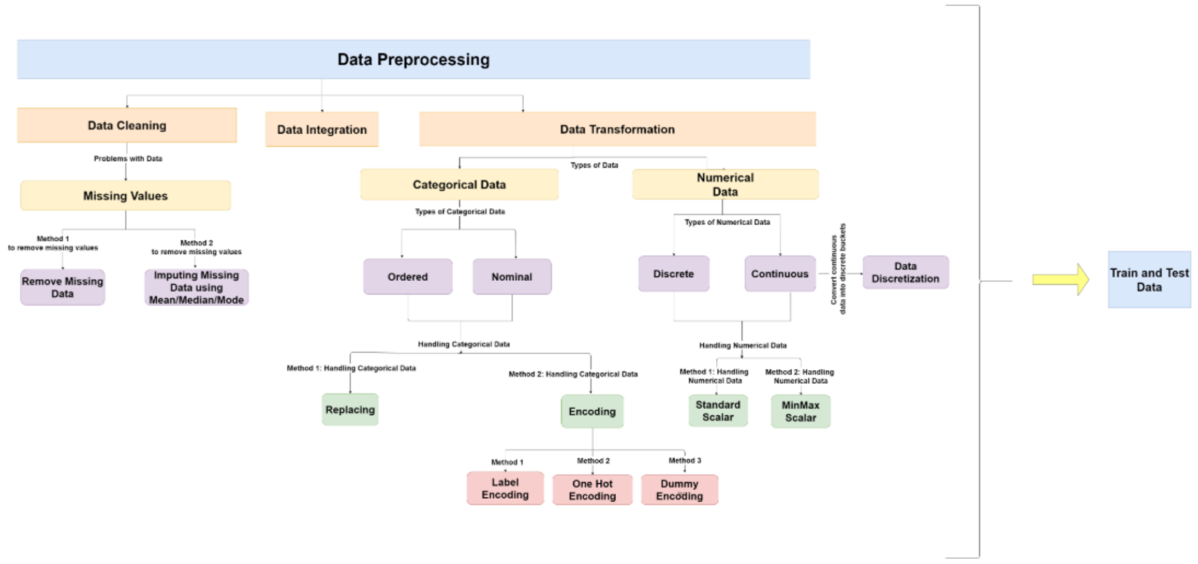

Figure 12: Data Preprocessing Workflow

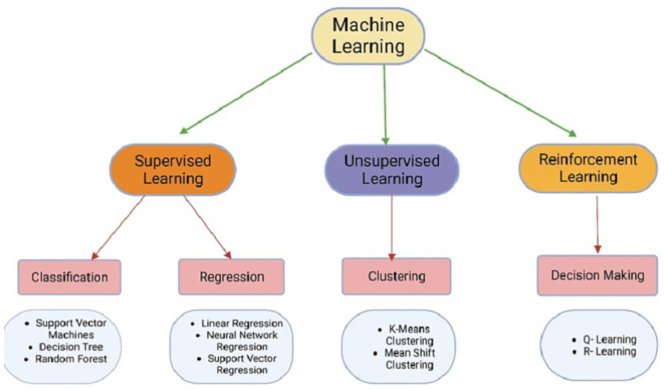

Figure 13: Machine learning types and its algorithms

(a) **Definition**
Linear regression is a supervised learning algorithm used for predicting continuous numerical values based on the relationship between independent (predictor) and dependent (response) variables. It assumes a linear relationship between input features and the target variable.

(b) **Mathematical Representation**: The equation for a simple linear regression model is:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where: Where:

- Y = Predicted output (dependent variable)
- X = Input feature (independent variable)
- β_0 = Intercept (bias term)
- β_1 = Slope (weight assigned to X)

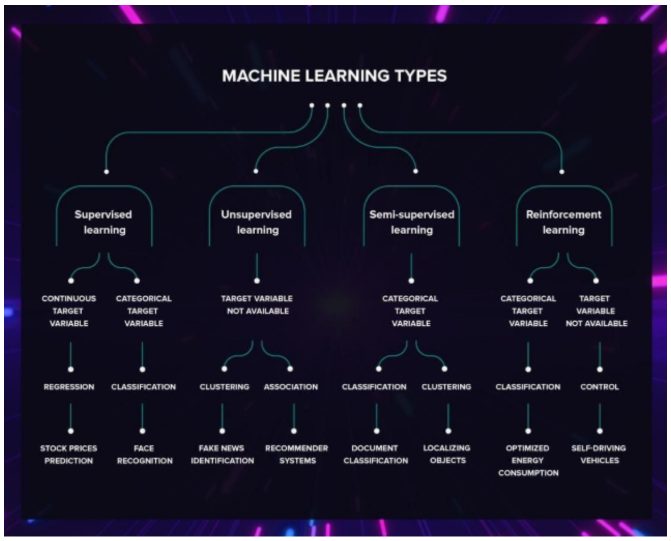

Figure 14: Machine learning types and its target nature

- ϵ = Error term

(c) **Use Cases**: House price prediction based on size and location, Forecasting sales revenue, Predicting temperature changes

(d) **Advantages**
Simple and easy to interpret, and Works well for linearly related data.

(e) **Disadvantages**
Poor performance on non-linear relationships, and Sensitive to outliers

Examples: Simple Linear Regression (One Feature)

```python

    import numpy as np
```

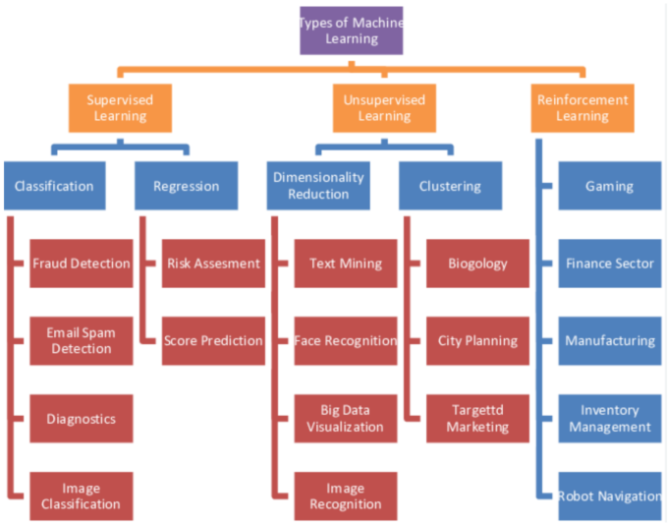

Figure 15: Machine learning types with Learning Tasks and examples

```
import matplotlib.pyplot as plt
from sklearn.linear_model import
LinearRegression

# Generate sample data
X = np.array([1, 2, 3, 4, 5]).
reshape(-1, 1)
y = np.array([2, 4, 5, 4, 5])

# Create and train the model
model = LinearRegression()
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)

# Plot results
plt.scatter(X, y, color='blue',
label="Actual Data")
plt.plot(X, y_pred, color='red',
linewidth=2, label="Regression Line"
)
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()

# Print model coefficients
print(f"Intercept: {model.intercept_
}")
print(f"Coefficient: {model.coef_
[0]}")
```

Listing 188: With one features

Multiple Linear Regression (Multiple Features)

```
import numpy as np
import pandas as pd
from sklearn.linear_model import
LinearRegression
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
mean_squared_error

# Sample dataset
data = {
    'Feature1': [1, 2, 3, 4, 5, 6,
7, 8, 9, 10],
    'Feature2': [10, 9, 8, 7, 6, 5,
4, 3, 2, 1],
    'Target': [2.2, 3.8, 5.5, 7.1,
8.7, 10.1, 11.8, 13.4, 15.0, 16.5]
}

df = pd.DataFrame(data)

# Split dataset
X = df[['Feature1', 'Feature2']]
y = df['Target']
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size
=0.2, random_state=42)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Model evaluation
mse = mean_squared_error(y_test,
y_pred)
print(f"Mean Squared Error: {mse}")
print(f"Intercept: {model.intercept_
}")
print(f"Coefficients: {model.coef_}"
)
```

Listing 189: With Multiple features

Linear Regression with a Real Dataset (Boston Housing)

```
from sklearn.datasets import
load_diabetes
from sklearn.linear_model import
LinearRegression
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
mean_absolute_error, r2_score

# Load dataset
data = load_diabetes()
X = data.data
y = data.target

# Split dataset
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size
=0.2, random_state=42)

# Train model
model = LinearRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Model evaluation
```

```
mae = mean_absolute_error(y_test,
y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error: {mae}")
print(f"R-squared Score: {r2}")
print(f"Intercept: {model.intercept_
}")
print(f"Coefficients: {model.coef_}"
)

```

Listing 190: With Real dataset

2. **Logistic Regression**

   (a) **Definition**
   Logistic regression is a classification algorithm used when the output variable is categorical (binary or multi-class). Instead of predicting a continuous value like linear regression, it predicts probabilities for class labels[3].

   (b) **Mathematical Representation**
   Logistic regression uses the sigmoid function (logistic function) to map predictions between 0 and 1.

$$P(Y = 1 \mid X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

   Where:

   - $P(Y = 1 \mid X)$ = Probability of the positive class
   - e = Euler's number (≈ 2.718)
   - β_0, β_1 = Coefficients

   (c) **Use Cases**
   Spam detection (Spam or Not Spam), Medical diagnosis (Disease or No Disease), and Credit risk assessment (Default or No Default)

   (d) **Advantages**
   Works well for binary classification, and Provides probability scores for predictions

   (e) **Disadvantages**
   Assumes a linear relationship between features and log-odds, and Not suitable for complex, non-linear problems

Examples: Binary Classification using Logistic Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import
LogisticRegression
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
accuracy_score,
classification_report

# Generate sample data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  #
Features (100 samples)
y = (X > 5).astype(int).ravel()  #
Labels (Binary classification: 0 or
1)

# Split data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size
=0.2, random_state=42)

# Train logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test,
y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n",
classification_report(y_test, y_pred
))

# Plot decision boundary
X_range = np.linspace(0, 10, 100).
reshape(-1, 1)
y_prob = model.predict_proba(X_range
)[:, 1]

plt.scatter(X_train, y_train, color=
'blue', label="Training Data")
plt.scatter(X_test, y_test, color='
red', label="Testing Data")
plt.plot(X_range, y_prob, color='
black', linewidth=2, label="Decision
 Boundary")
plt.xlabel("Feature")
plt.ylabel("Probability")
plt.legend()
plt.show()


```

Listing 191: Binary Classification

Multiclass Classification using Logistic Regression (Iris Dataset)

```
from sklearn.datasets import
load_iris
from sklearn.linear_model import
LogisticRegression
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
accuracy_score,
classification_report

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target  #
Features and labels

# Split data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size
=0.2, random_state=42)

```

```
    # Train logistic regression model
    model = LogisticRegression(
    multi_class='ovr', solver='lbfgs',
    max_iter=200)
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Evaluate model
    accuracy = accuracy_score(y_test,
    y_pred)
    print(f"Accuracy: {accuracy:.2f}")
    print("Classification Report:\n",
    classification_report(y_test, y_pred
    ))


```

Listing 192: Using Iris dataset

Logistic Regression on a Real Dataset (Breast Cancer Detection)

```
    from sklearn.datasets import
    load_breast_cancer
    from sklearn.linear_model import
    LogisticRegression
    from sklearn.model_selection import
    train_test_split
    from sklearn.metrics import
    confusion_matrix,
    classification_report,
    accuracy_score

    # Load dataset
    data = load_breast_cancer()
    X, y = data.data, data.target

    # Split data
    X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size
    =0.2, random_state=42)

    # Train model
    model = LogisticRegression(max_iter
    =5000)
    model.fit(X_train, y_train)

    # Predictions
    y_pred = model.predict(X_test)

    # Evaluate model
    accuracy = accuracy_score(y_test,
    y_pred)
    conf_matrix = confusion_matrix(
    y_test, y_pred)

    print(f"Accuracy: {accuracy:.2f}")
    print("Confusion Matrix:\n",
    conf_matrix)
    print("Classification Report:\n",
    classification_report(y_test, y_pred
    ))


```

Listing 193: Using Breast Cancer Detection

3. **Decision Tree**

   (a) **Definition**
   A decision tree is a rule-based algorithm that splits data into branches based on feature values, creating a tree-like structure for decision-making.

   (b) **Working Mechanism**

   - The algorithm starts from a root node and splits data based on conditions at each node
   - Splitting is done using metrics like Gini Impurity or Entropy (Information Gain)
   - The tree grows until a stopping criterion is met (e.g., maximum depth or minimum samples per leaf). consider figure [16]

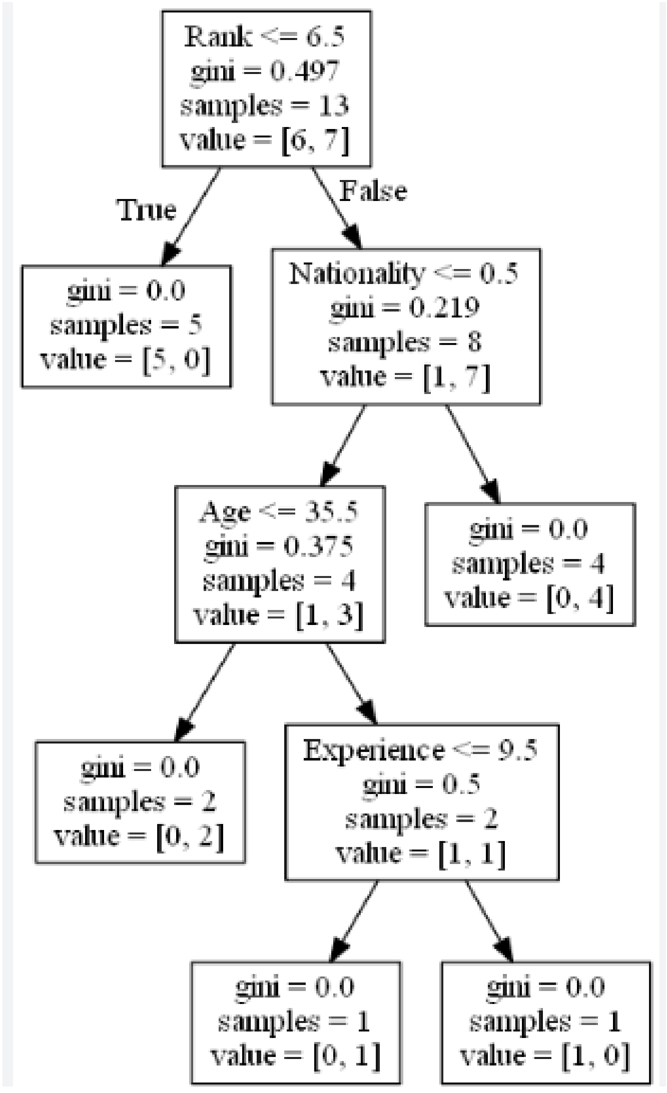

Figure 16: Decision Tree

   (c) **Use Cases**
   Customer segmentation, Fraud detection, and Medical diagnosis

   (d) **Advantages**

Easy to understand and interpret, and Handles both numerical and categorical data

(e) **Disadvantages**
Prone to overfitting (deep trees can be too complex), and Unstable (small changes in data can change the tree structure)

Examples:Decision Tree for Classification (Binary Classification)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Generate sample data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # Features
y = (X > 5).astype(int).ravel()  # Labels (0 or 1)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train decision tree model
model = DecisionTreeClassifier(max_depth=3)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))

# Plot decision tree
plt.figure(figsize=(10, 6))
plot_tree(model, filled=True, feature_names=["Feature"])
plt.show()

```

Listing 194: Binary classification

Decision Tree for Multiclass Classification (Iris Dataset)

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train decision tree model
model = DecisionTreeClassifier(max_depth=4)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))

```

Listing 195: Multiclass classification using Iris dataset

Decision Tree for Regression

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# Generate sample data
np.random.seed(42)
X = np.sort(5 * np.random.rand(80, 1), axis=0)
y = np.sin(X).ravel() + np.random.randn(80) * 0.1  # Adding noise

# Train decision tree regressor
model = DecisionTreeRegressor(max_depth=4)
model.fit(X, y)

# Predict
X_test = np.linspace(0, 5, 100).reshape(-1, 1)
y_pred = model.predict(X_test)

# Plot results
plt.scatter(X, y, color="blue", label="Training Data")
plt.plot(X_test, y_pred, color="red", linewidth=2, label="Decision Tree Prediction")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.show()

```

Listing 196: Decision tree for Regression

4. **Random Forest**

(a) **Definition**
Random forest is an ensemble learning algorithm that builds multiple decision trees and combines their outputs for better accuracy

and stability. It reduces overfitting by averaging multiple predictions.

(b) **Working Mechanism**

- Creates multiple decision trees using bootstrap sampling (random sampling with replacement).
- Uses random feature selection to reduce correlation between trees.
- Aggregates predictions using majority voting (classification) or averaging (regression).

(c) **Use Cases**
Loan approval prediction, Image classification, and Disease diagnosis

(d) **Advantages**
High accuracy compared to single decision trees, and Works well with missing data and imbalanced datasets

(e) **Disadvantages**
Computationally expensive for large datasets, and Less interpretable than a single decision tree

5. **K-Nearest Neighbors (KNN)**

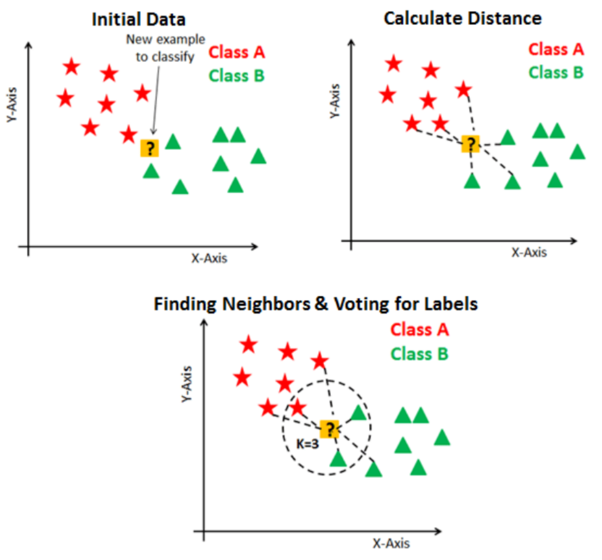

Figure 17: K-Nearest Neighbor

(a) **Definition**
KNN is a non-parametric, instance-based learning algorithm that classifies new data points based on the majority class of their K nearest neighbors in the training data. Refer to figure [17]

(b) **Working Mechanism**

i. Select a value for K (number of neighbors).
ii. Measure the distance between the new data point and all training points using Euclidean distance or Manhattan distance
iii. Identify the K nearest neighbors.
iv. Assign the most common class label (classification) or compute the average (regression).

(c) **Use Cases**
Handwriting recognition, Recommender systems, and Fraud detection

(d) **Advantages**
Simple and intuitive, and Works well for small datasets

(e) **Disadvantages**
Computationally expensive for large datasets (requires storing all data points), and Sensitive to irrelevant features and noisy data

Examples: [2] **KNN for Binary Classification**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import
KNeighborsClassifier
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
accuracy_score,
classification_report

# Generate sample data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  #
Feature
y = (X > 5).astype(int).ravel()  #
Labels (Binary classification: 0 or
1)

# Split data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size
=0.2, random_state=42)

# Train KNN model
model = KNeighborsClassifier(
n_neighbors=5)  # Using k=5
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test,
y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n",
classification_report(y_test, y_pred
))
```

[2]K-Nearest Neighbor(KNN) is a very simple, easy-to-understand, versatile, and one of the topmost machine learning algorithms. KNN used in a variety of applications such as finance, healthcare, political science, handwriting detection, image recognition, and video recognition.

For more information and explained codes clicki here machinelearningeek.com and mlarchive.com

```

```

Listing 197: KNN for Binary Classification

KNN for Multiclass Classification (Iris Dataset)

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train KNN model
model = KNeighborsClassifier(n_neighbors=5)  # Using k=5
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n", classification_report(y_test, y_pred))

```

Listing 198: KNN for Multiclass classification

KNN for Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor

# Generate sample data
np.random.seed(42)
X = np.sort(5 * np.random.rand(80, 1), axis=0)
y = np.sin(X).ravel() + np.random.randn(80) * 0.1  # Adding noise

# Train KNN Regressor
model = KNeighborsRegressor(n_neighbors=5)  # Using k=5
model.fit(X, y)

# Predict
X_test = np.linspace(0, 5, 100).reshape(-1, 1)
y_pred = model.predict(X_test)

# Plot results
plt.scatter(X, y, color="blue", label="Training Data")
plt.plot(X_test, y_pred, color="red", linewidth=2, label="KNN Prediction")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.show()

```

Listing 199: KNN for regression

6. **Support Vector Machine (SVM)**

(a) **Definition**
SVM is a powerful classification algorithm that finds the optimal hyperplane to separate classes with the maximum margin. It works well for both linear and non-linear classification by using kernel tricks[3].

(b) **Working Mechanism**

- Finds the hyperplane that best separates different classes.
- Maximizes the margin (distance between support vectors and hyperplane).
- Uses kernels (linear, polynomial, radial basis function - RBF) for handling non-linear data.

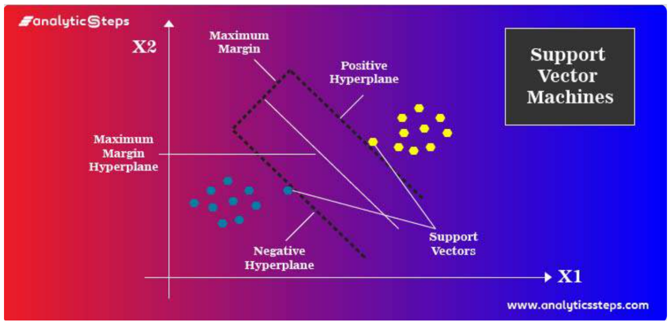

Figure 18: Support Vector Machine

(c) **Use Cases**
Face recognition, Spam email detection, and Medical diagnosis

(d) **Advantages**
Works well for high-dimensional data, and Effective in cases where the number of dimensions is greater than the number of samples

(e) **Disadvantages**
Computationally intensive for large datasets, and Requires careful tuning of kernel parameters

Examples: SVM for Binary Classification

[3]scikit-learn.org

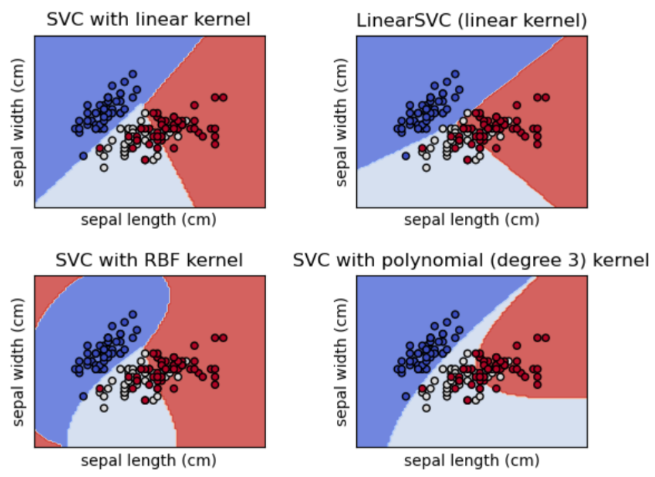

Figure 19: Support Vector Machine with Kernels

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
accuracy_score, classification_report

# Generate sample data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  #
Feature
y = (X > 5).astype(int).ravel()  #
Labels (Binary classification: 0 or 1)

# Split data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
random_state=42)

# Train SVM model
model = SVC(kernel='linear', C=1.0)  #
Linear kernel
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred
)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n",
classification_report(y_test, y_pred))
```

Listing 200: SVM for Binary Classification

SVM for Multiclass Classification (Iris Dataset)

```python
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.model_selection import
train_test_split
from sklearn.metrics import
accuracy_score, classification_report

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
random_state=42)

# Train SVM model
model = SVC(kernel='rbf', C=1.0, gamma='
scale')  # RBF kernel for non-linear
classification
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred
)
print(f"Accuracy: {accuracy:.2f}")
print("Classification Report:\n",
classification_report(y_test, y_pred))
```

Listing 201: SVM for Multiclass Classification (Iris Dataset)

SVM for Regression

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVR

# Generate sample data
np.random.seed(42)
X = np.sort(5 * np.random.rand(80, 1),
axis=0)
y = np.sin(X).ravel() + np.random.randn
(80) * 0.1  # Adding noise

# Train SVM Regressor
model = SVR(kernel='rbf', C=1.0, epsilon
=0.1)  # RBF kernel for smooth
regression
model.fit(X, y)

# Predict
X_test = np.linspace(0, 5, 100).reshape
(-1, 1)
y_pred = model.predict(X_test)

# Plot results
plt.scatter(X, y, color="blue", label="
Training Data")
plt.plot(X_test, y_pred, color="red",
linewidth=2, label="SVM Prediction")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.show()
```

Listing 202: SVM for Regression

6.3 Building data model

6.3.1 Artificial Neural networks

1. **Definition**
   An Artificial Neural Network (ANN) is a computational model inspired by the way biological

neural networks in the brain process information. It consists of layers of interconnected nodes, also known as neurons. ANNs are used for various machine learning tasks, such as classification, regression, pattern recognition, and function approximation.

(a) **Input Layer:** Receives the input features. Refer to figure [22]

(b) **Hidden Layers:** Perform computations to learn complex patterns.

(c) **Output Layer:** Produces the final prediction or classification.

(d) **Weights & Biases:** Parameters that determine the strength and influence of connections between neurons.

(e) **Activation Function:** Non-linear functions (e.g., ReLU, Sigmoid, Tanh) used to introduce non-linearity and allow the network to learn complex patterns.

2. **Use Case Implementation Steps**
The typical steps for implementing an artificial neural network are as follows. Refer to figure [20], and [21]

(a) **Problem Definition**: Define the problem you want the neural network to solve (e.g., image classification, sentiment analysis, sales prediction).

(b) **Data Collection**: Gather and preprocess the data needed for training the model. This may involve tasks like data cleaning, normalization, and splitting into training and test sets.

(c) **Network Architecture Design**: Decide on the structure of the neural network:

- Number of layers (input, hidden, and output layers)
- Number of neurons per layer
- Choice of activation function (e.g., ReLU for hidden layers, Softmax for output in classification tasks)

(d) **Forward Propagation**: Pass the input data through the network. Each neuron performs a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.

(e) **Loss Function**: Define a loss function (e.g., Mean Squared Error for regression, Cross-Entropy for classification) that quantifies the difference between predicted and actual outputs.

(f) **Backpropagation**: Update the weights and biases of the network based on the gradient of the loss function with respect to the weights. This process uses optimization techniques such as Gradient Descent.

(g) **Training**: Train the network by feeding the training data and adjusting weights over several iterations (epochs) to minimize the loss function.

(h) **Evaluation**: Evaluate the trained model on unseen test data to assess its performance (e.g., accuracy, precision, recall).

(i) **Model Deployment**: Once the model is trained and evaluated, deploy it to make real-time predictions on new data.

Typical Example[4]**:**

```python
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

# Load MNIST dataset
(X_train, y_train), (X_test, y_test)
 = keras.datasets.mnist.load_data()

# Normalize pixel values to range
[0,1]
X_train, X_test = X_train / 255.0,
X_test / 255.0

# Flatten images (28x28 to 784
features)
X_train = X_train.reshape(-1, 28 *
28)
X_test = X_test.reshape(-1, 28 * 28)

# Define Neural Network model
model = keras.Sequential([
    keras.layers.Dense(128,
activation='relu', input_shape
=(784,)),  # Input layer
    keras.layers.Dense(64,
activation='relu'),  # Hidden layer
    keras.layers.Dense(10,
activation='softmax')  # Output
layer (10 classes)
])

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

# Train model
model.fit(X_train, y_train, epochs
=10, batch_size=32, validation_data
=(X_test, y_test))

# Evaluate model
test_loss, test_acc = model.evaluate
(X_test, y_test, verbose=2)
print(f"Test accuracy: {test_acc:.2f
}")
```

[4] This model achieves around 97-98% accuracy on the MNIST test set

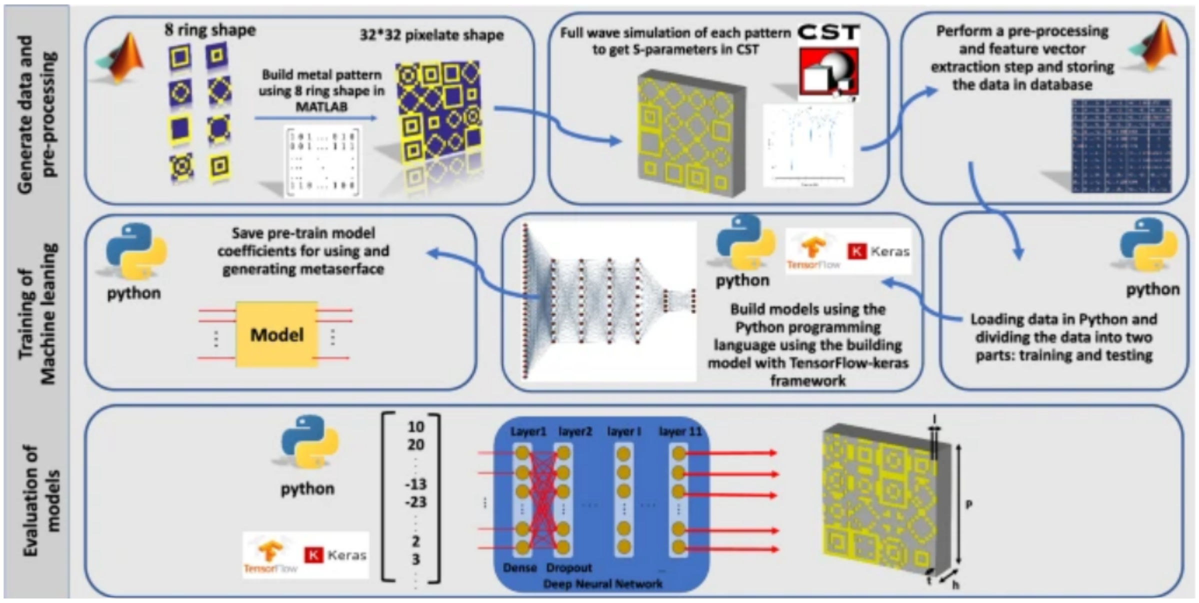
Generate data and pre-processing
8 ring shape
32*32 pixelate shape
Build metal pattern using 8 ring shape in MATLAB
Full wave simulation of each pattern to get S-parameters in CST
CST
Perform a pre-processing and feature vector extraction step and storing the data in database
Training of Machine leaning
python
Save pre-train model coefficients for using and generating metaserface
Model
python
TensorFlow
Keras
Build models using the Python programming language using the building model with TensorFlow-keras framework
python
Loading data in Python and dividing the data into two parts: training and testing
Evaluation of models
python
TensorFlow
Keras
Layer1 layer2 layer l layer 11
Dense Dropout
Deep Neural Network
p
h

Figure 20: Most of Artificial Neural Network flow

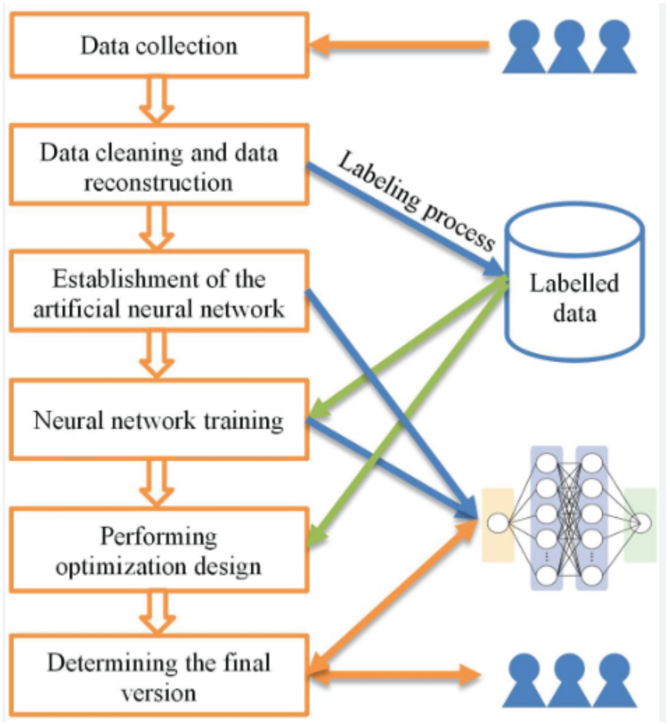
Data collection
Data cleaning and data reconstruction
Labeling process
Labelled data
Establishment of the artificial neural network
Neural network training
Performing optimization design
Determining the final version

Figure 21: Summarized ANN workflow

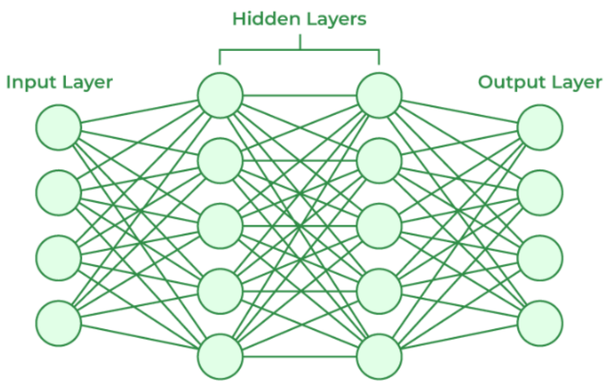

Figure 22: Neural Network Architecture

```
    # Make a prediction
    predictions = model.predict(X_test)

    # Display an example
    index = np.random.randint(0, len(X_test))
    plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
    plt.title(f"Predicted: {np.argmax(predictions[index])}, Actual: {y_test[index]}")
    plt.show()
```

Listing 203: MNIST Classification

(a) **Data Preprocessing**

- The MNIST dataset contains **28x28 grayscale images** of handwritten digits (0-9). Refer to figure [23]
- The pixel values are **normalized to [0,1]**.
- The images are **flattened** from **(28,28) - (784,)** for input into the neural network.

(b) **Model Architecture**

- **Input Layer**: 784 neurons (one for each pixel).

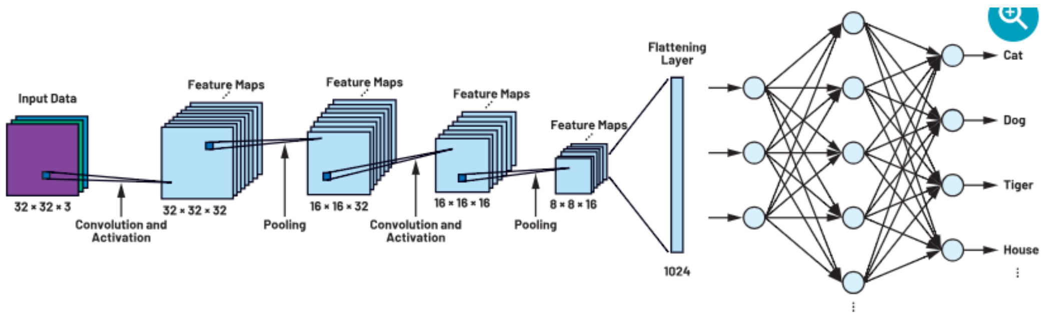

Figure 1. CIFAR CNN architecture.

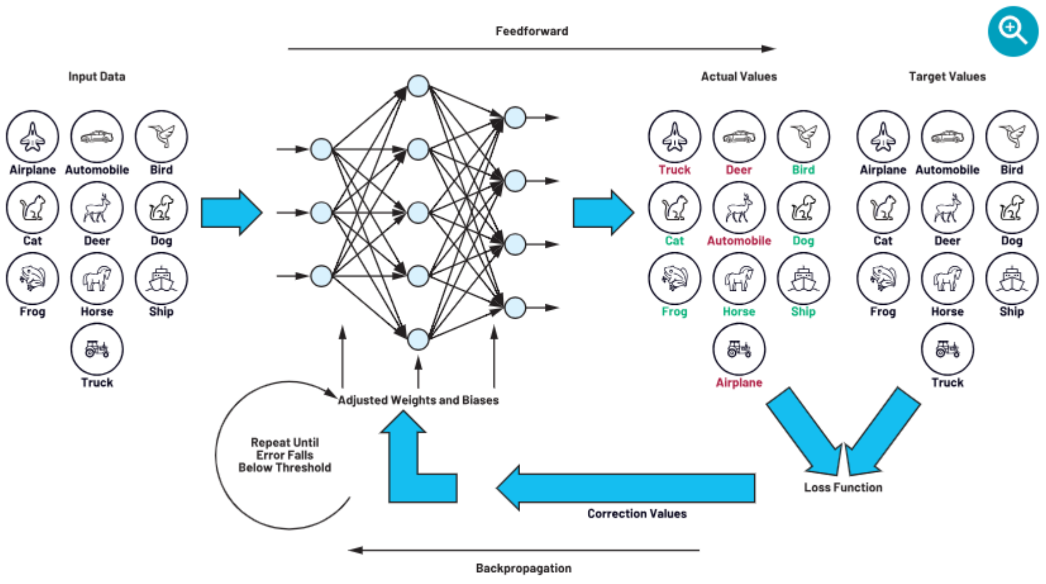

Figure 23: Convolution Neural Network Illustration (CNN)

- **Hidden Layers**: First hidden layer with **128 neurons** (ReLU activation). Second hidden layer with **64 neurons** (ReLU activation).
- **Output Layer**: 10 neurons (one for each digit 0-9), using **softmax activation**.

(c) **Compilation & Training**

- **Loss Function**: *sparse_categorical_crossentropy* (since labels are integers).
- **Optimizer**: *adam* (Adaptive Moment Estimation).
- **Metrics**: Accuracy is used to evaluate the model.
- The model is trained for **10 epochs** with a batch size of **32**.

(d) **Prediction & Visualization**

- A **random test image** is displayed with its **predicted and actual label**.

3. **Neural Network Examples**

(a) **Image Classification (CNNs)**

- **Problem**: Classifying images into categories (e.g., dog vs. cat).
- **Example**: A Convolutional Neural Network (CNN) can be used to process image data by extracting spatial features using convolutional layers, followed by fully connected layers for classification. CNNs are highly effective for image-related tasks.
- **Use case**: Recognizing handwritten digits (MNIST dataset).

(b) **Natural Language Processing (RNNs & LSTMs)**

- **Problem**: Predicting the next word in a sentence or generating text.
- **Example**: A Recurrent Neural Network (RNN) or Long Short-Term Memory (LSTM) network can be used to

process sequential data, such as text. LSTMs are especially good at capturing long-range dependencies in the text.

- **Use case**: Sentiment analysis of reviews or text generation using models like GPT (Generative Pretrained Transformer).

Examples[5]**:**

RNN for Text Classification (Sentiment Analysis on IMDB Dataset)

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# Load IMDB dataset
vocab_size = 10000  # Use top 10,000 words
max_length = 200  # Max length of a review
(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data(num_words=vocab_size)

# Pad sequences to ensure uniform input size
X_train = pad_sequences(X_train, maxlen=max_length, padding='post', truncating='post')
X_test = pad_sequences(X_test, maxlen=max_length, padding='post', truncating='post')

# Define RNN Model
model = keras.Sequential([
    keras.layers.Embedding(input_dim=vocab_size, output_dim=128, input_length=max_length),
    keras.layers.SimpleRNN(64, return_sequences=False),  # Basic RNN layer
    keras.layers.Dense(1, activation='sigmoid')  # Binary classification
])

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

# Train Model
model.fit(X_train, y_train, epochs=5, batch_size=64, validation_data=(X_test, y_test))

# Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")

```

Listing 204: Sentiment Analysis on IMDB Dataset

Explanation

- **Dataset**: IMDB Movie Reviews (binary sentiment classification)
- **Preprocessing**: Padding ensures all reviews have a uniform length.
- **Embedding Layer**: Converts words into dense vectors
- **RNN Layer**: Processes the sequences.
- **Output Layer**: Uses sigmoid activation for binary classification.

LSTM for Text Classification (Sentiment Analysis on IMDB)

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Load IMDB dataset
vocab_size = 10000
max_length = 200
(X_train, y_train), (X_test, y_test) = keras.datasets.imdb.load_data(num_words=vocab_size)

# Pad sequences
X_train = pad_sequences(X_train, maxlen=max_length, padding='post', truncating='post')
X_test = pad_sequences(X_test, maxlen=max_length, padding='post', truncating='post')

# Define LSTM Model
model = keras.Sequential([
    keras.layers.Embedding(input_dim=vocab_size, output_dim=128, input_length=max_length),
    keras.layers.LSTM(64, return_sequences=False),  # LSTM Layer
    keras.layers.Dense(1, activation='sigmoid')  # Binary classification
])

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

# Train Model
model.fit(X_train, y_train, epochs=5, batch_size=64, validation_data=(X_test, y_test))

# Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
```

[5]These examples cover RNN-based and LSTM-based NLP tasks: **Text Classification** (Sentiment Analysis on IMDB) **Text Generation** (Next-word prediction)

Listing 205: Sentiment Analysis

Why LSTM?

- Unlike a standard RNN, **LSTM** handles long-range dependencies better due to **gates (input, forget, output)**.
- It helps prevent **vanishing gradients**.

LSTM for Text Generation (Shakespeare-style Text)

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

# Sample text data
text = """To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,"""

# Tokenization
tokenizer = Tokenizer()
tokenizer.fit_on_texts([text])
word_index = tokenizer.word_index
total_words = len(word_index) + 1

# Create sequences
input_sequences = []
for line in text.split("\n"):
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        input_sequences.append(token_list[:i + 1])

# Pad sequences
max_length = max(len(seq) for seq in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_length, padding='pre')

# Split into input (X) and output (y)
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = keras.utils.to_categorical(y, num_classes=total_words)

# Define LSTM Model
model = keras.Sequential([
    keras.layers.Embedding(total_words, 50, input_length=max_length - 1),
    keras.layers.LSTM(100),
    keras.layers.Dense(total_words, activation='softmax')  # Multi-class classification
])

# Compile Model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train Model
model.fit(X, y, epochs=500, verbose=1)

# Generate text
def generate_text(seed_text, next_words=10):
    for _ in range(next_words):
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_length - 1, padding='pre')
        predicted = np.argmax(model.predict(token_list), axis=-1)
        for word, index in tokenizer.word_index.items():
            if index == predicted:
                seed_text += " " + word
                break
    return seed_text

# Example prediction
print(generate_text("To be", next_words=10))
```

Listing 206: Using Shakespeare-style Text

Explanation

- **Data Preparation**: Tokenizes text and creates sequences of increasing length.
- **LSTM Model**: Predicts the next word given previous words.
- **Text Generation**: Uses trained model to predict words sequentially.

4. **Speech Recognition (Deep Neural Networks)**

   - **Problem**: Converting spoken language into text.
   - **Example**: A Deep Neural Network (DNN) can be used to process audio features like spectrograms and transcribe them into text. Often, CNNs and RNNs are combined for better accuracy in speech recognition.
   - **Use case**: Voice assistants like Siri, Alexa, or Google Assistant.

5. **Autonomous Driving (Deep Learning)**

   - **Problem**: Enabling self-driving cars to recognize road signs, pedestrians, and obstacles.

- **Example**: Deep Neural Networks (DNNs) and CNNs are employed to process images from cameras, LiDAR, and radar sensors for object detection, classification, and decision-making.
- **Use case**: Tesla's self-driving cars use deep learning models for real-time decision-making based on sensor inputs.

6.3.2 Building model steps

When building a machine learning model, the process is structured and involves several key steps to ensure that the model is effective in solving the problem at hand. Below are the steps for building a machine learning model

1. **Identify Business Problems**
   Before jumping into building a model, it is essential to define the problem clearly. This step involves:
   - Understanding the business objectives: What problem does the business need to solve?
   - Deciding on the type of model: Is it a classification problem, regression, recommendation system, or something else?
   - Setting specific goals: What outcome or prediction does the business need from the model?
   - Communicating the expected performance: What accuracy, precision, or other metrics are acceptable for the model to be considered useful?
2. **Identify and Understand the Data**
   Once the business problem is defined, you need to identify and understand the data that will be used to train the model.
   - **Data Source**: Identify where the data comes from (e.g., databases, APIs, spreadsheets).
   - **Data Relevance**: Understand if the available data is relevant to solving the business problem.
   - **Data Types**: Identify whether the data is numerical, categorical, or textual.
   - **Data Exploration**: Perform exploratory data analysis (EDA) to uncover patterns, trends, and relationships within the data (using visualizations, correlation analysis, etc.).
   - **Check for Missing Values**: Determine if there are any missing or incomplete data points that need handling.
3. **Collect and Prepare Data**
   Data preparation is one of the most critical steps in model development. Clean, preprocessed data is essential for accurate and reliable models. This includes:
   - **Data Cleaning**: Handle missing values, remove duplicates, and correct errors.
   - **Data Transformation**: Normalize or standardize numerical features and encode categorical features (e.g., one-hot encoding, label encoding).
   - **Feature Engineering**: Create new features or modify existing ones based on domain knowledge to improve model performance.
   - **Data Splitting**: Split the data into training and testing datasets (typically 80-20 or 70-30 split).
4. **Determine Models and Train Data**
   After preparing the data, it is time to choose the right machine learning model based on the problem type. This could be a:
   - **Supervised model**: For classification or regression tasks (e.g., Logistic Regression, Decision Trees, SVM).
   - **Unsupervised model**: For clustering or dimensionality reduction tasks (e.g., K-Means, PCA).
   - **Reinforcement learning**: For tasks like game-playing or robotic control (e.g., Q-learning)
   - **Deep learning model**: For more complex problems, like image or speech recognition (e.g., Neural Networks).

   Once you've selected a model:
   - Train the model using the training dataset.
   - Adjust the model's parameters and fit the model to the data.
5. **Evaluate Models**
   Model evaluation is crucial to assess its performance and ensure it meets the business requirements. Key steps in model evaluation include:
   - **Model Metrics**: Choose appropriate metrics based on the task (e.g., Accuracy, Precision, Recall, F1-Score for classification; Mean Squared Error for regression).
   - **Cross-validation**: Use techniques like k-fold cross-validation to assess how the model performs on different subsets of the data, ensuring robustness.

- **Confusion Matrix**: For classification tasks, analyze the confusion matrix to understand the true positives, false positives, true negatives, and false negatives.
- **Overfitting and Underfitting**: Check if the model generalizes well to unseen data or if it is overfitting to the training set.

6. **Experiment and Adjust the Model**
After evaluating the model, you may need to refine it further by experimenting with different parameters or techniques

   - **Hyperparameter Tuning**: Use methods like Grid Search or Random Search to tune hyperparameters and optimize the model's performance.
   - **Feature Selection**: Experiment with selecting the most relevant features to improve model accuracy.
   - **Model Comparison**: Try different algorithms (e.g., comparing Decision Trees, Random Forests, and Support Vector Machines) to see which one performs the best.
   - **Ensemble Methods**: Combine multiple models (e.g., using techniques like Bagging, Boosting, or Stacking) to improve accuracy.

   Finally, iterate on the process until you achieve an optimal model that meets the desired performance metrics.

6.3.3 Deep learning

1. **Definition**
Deep Learning is a subset of machine learning that uses neural networks with many layers (hence "deep") to model complex patterns and representations in large datasets. Unlike traditional machine learning algorithms, deep learning models automatically learn feature representations from raw data, reducing the need for manual feature engineering. These models are particularly effective in handling tasks like image recognition, natural language processing (NLP), and speech recognition.

   Deep learning involves training artificial neural networks with multiple layers, where each layer learns to transform the input data into more abstract and higher-level representations. The complexity and depth of the models allow them to learn from large amounts of data and improve over time.

2. **Natural Language Processing (NLP)**
Natural Language Processing (NLP) is a field of deep learning that focuses on the interaction between computers and human language. It allows machines to understand, interpret, and generate human language in a way that is valuable. Key tasks in NLP include:

   - **Text Classification**: Categorizing text into predefined categories (e.g., spam detection, sentiment analysis).
   - **Named Entity Recognition (NER)**: Identifying entities like names, dates, locations in text.
   - **Machine Translation**: Automatically translating text from one language to another (e.g., Google Translate).
   - **Text Generation**: Producing coherent and contextually appropriate text based on input prompts (e.g., GPT-3).

   Deep learning techniques, such as Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs) as shown in figure [27], and Transformers (e.g., BERT, GPT), have revolutionized NLP by achieving state-of-the-art performance across these tasks.

Example of Speech Recognition[6]

This example will show how to preprocess audio data, extract Mel Frequency Cepstral Coefficients (MFCCs), and use a Deep Neural Network to classify speech commands.

```python
import os
import numpy as np
import librosa
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load and process audio files to extract MFCC features
def extract_features(file_path):
    audio, sr = librosa.load(file_path, sr=None)  # Load the audio file
    mfcc = librosa.feature.mfcc(audio, sr=sr, n_mfcc=13)  # Extract MFCCs
    mfcc_scaled = np.mean(mfcc.T, axis=0)  # Average over time axis
    return mfcc_scaled

```

[6]We'll use the TensorFlow/Keras framework for building the neural network model, and Librosa for audio processing. For simplicity, we will use a pre-recorded speech dataset, such as Google's Speech Commands Dataset, you can download it directly from Google's Speech Commands Dataset

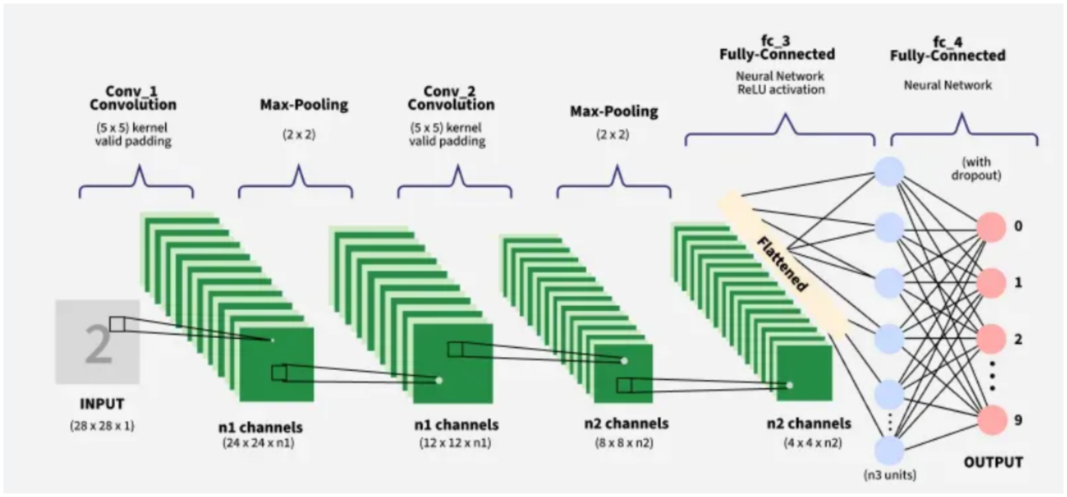

Figure 24: Convolution with Pooling

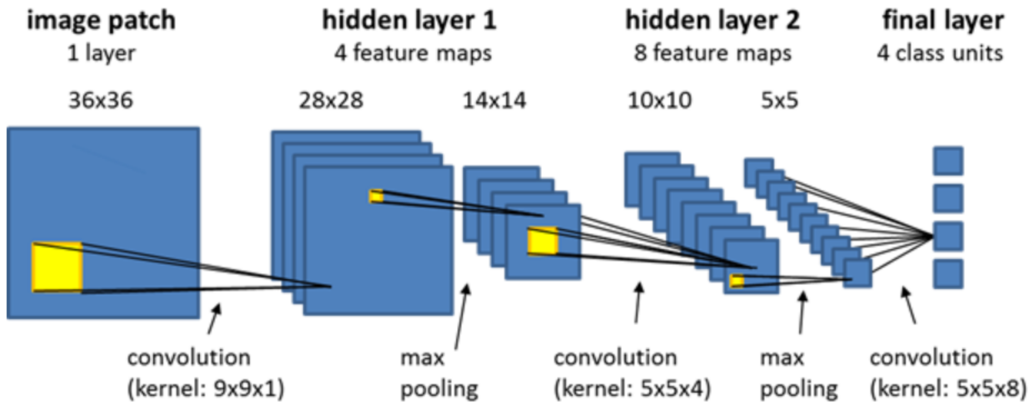

Figure 25: Convolve with Kernel Size

```python
# Load dataset (assuming you have your dataset ready in the folder 'data')
data_dir = 'data/speech_commands'
labels = []
features = []

for label in os.listdir(data_dir):
    label_path = os.path.join(data_dir, label)
    if os.path.isdir(label_path):
        for file in os.listdir(label_path):
            if file.endswith(".wav"):
                file_path = os.path.join(label_path, file)
                mfcc = extract_features(file_path)
                features.append(mfcc)
                labels.append(label)

# Convert to numpy arrays
X = np.array(features)
y = np.array(labels)

# Encode labels to numeric values
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

# Build the Deep Neural Network model
model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),  # Input layer
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dropout(0.2),
    Dense(len(np.unique(y_encoded)), activation='softmax')  # Output layer for multi-class classification
])

# Compile the model
```

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

# Train the model
model.fit(X_train, y_train, epochs
=20, batch_size=32, validation_data
=(X_test, y_test))

# Evaluate the model
loss, accuracy = model.evaluate(
X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f
}")

# Function for predicting a command
from audio
def predict_command(audio_path):
    mfcc = extract_features(
audio_path)
    mfcc = np.expand_dims(mfcc, axis
=0)  # Add batch dimension
    prediction = model.predict(mfcc)
    predicted_label = label_encoder.
inverse_transform([np.argmax(
prediction)])
    return predicted_label[0]

# Example usage
audio_file = 'data/speech_commands/
yes/0a7c2a8d_nohash_0.wav'  #
Example file path
predicted_command = predict_command(
audio_file)
print(f"Predicted command: {
predicted_command}")


```

Listing 207: Speech Recognition

Explanation of the Code[7]

(a) **Feature Extraction (MFCCs**[8]**)**

- The Librosa library is used to load and process the audio files.
- We extract MFCCs from the audio files, which are widely used in speech recognition tasks as features.
- MFCCs are averaged across the time axis to get a fixed-length feature vector for each audio sample.

(b) **Model Construction**

- We build a Deep Neural Network (DNN) with two hidden layers (Dense layers with ReLU activation).
- Dropout is applied to avoid overfitting.
- The output layer uses softmax activation to classify the audio into one of the predefined speech command labels.

[7]**Requirements**: TensorFlow: For building the neural network model. Librosa: For extracting MFCCs from the audio files. NumPy and Scikit-learn: For data processing and model evaluation. (***pip install tensorflow librosa numpy scikit-learn***)

[8]MFCCs stands for: Mel Frequency Cepstral Coefficients

(c) **Label Encoding**

- The labels (e.g., "yes," "no," etc.) are converted into numerical labels using LabelEncoder.

(d) **Training**

- The model is trained on the audio features (MFCCs) and the encoded labels.
- The sparse_categorical_crossentropy loss function is used because the labels are integers.

(e) **Prediction**

- The model predicts the command for a given audio file using the trained model.

(f) **Example**

- The audio file 0a7c2a8d_nohash_0.wav is used as an example. The predict_command function predicts the speech command by extracting MFCC features and passing them through the model

3. **Image and Object Recognition**
Image and Object Recognition are two of the most popular applications of deep learning, particularly using Convolutional Neural Networks (CNNs). CNNs are designed to automatically detect and recognize patterns in images.

- **Image Recognition**: The process of identifying the content of an image, such as recognizing objects, scenes, or faces.
- **Object Detection**: Not only recognizing objects in an image but also locating them (bounding box around an object).

As shown form figure 26 You can read more here [9] Deep learning models have been able to outperform traditional computer vision methods by automatically learning hierarchical features in images (edges, textures, shapes) from raw pixels. Some applications include:

[9]**a**: Development of breast cancer prognostic biomarkers.

b: Main strategies of biomarker development for disease prognosis.

c: Co-registered multi-biomarker spatial heterogeneity (IGNN) in primary breast tumor with biomarker-biomarker interactions unavailable from corresponding single-marker heterogeneity (TACS1-8).

d: Personalized TACS1-8 reginal distributions in co-registered images (H&E, MPM of second harmonic generation SHG, and MPM of two-photon excited fluorescence TPEF) from one exemplary patient (9 regions/nodes, each of which encoded with an 8-bit vector) that result in one graph structure input for IGNN model and another non-graph input for TACS1-8 model.

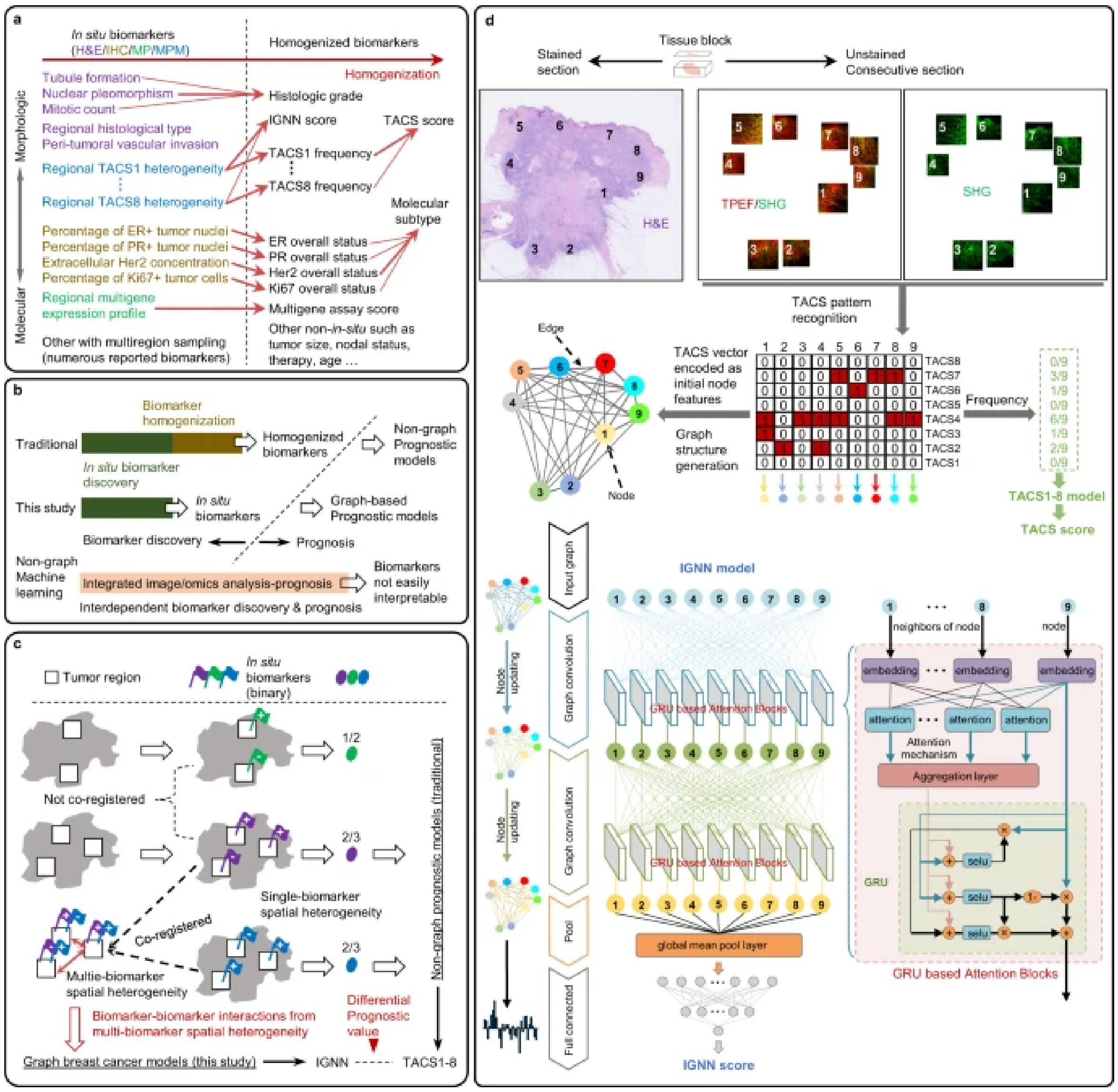

Figure 26: Breast Cancer Prognosis steps Illustration

- **Facial recognition**: Identifying individuals from images or videos.
- **Autonomous vehicles**: Recognizing road signs, pedestrians, and other vehicles.
- **Medical imaging**: Detecting abnormalities like tumors in radiographs or MRI scans.

4. **Prediction**
Deep learning is also widely used for predictive analytics, where the goal is to predict future outcomes based on historical data. Some areas of application include:

- **Stock market prediction**: Forecasting future stock prices or market trends using historical data.
- **Sales prediction**: Estimating future sales volumes based on past sales data and trends.
- **Demand forecasting**: Predicting demand for products or services, especially in supply chain management.

Deep learning models, especially Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, are well-suited for sequence prediction problems due to their ability to handle temporal dependencies in time-series data.

5. **Deep Learning Layers**
A deep learning model typically consists of several types of layers, each performing specific operations. Common layers in deep learning include:

- **Input Layer**: Receives the raw data (e.g., an image or text)
- **Convolutional Layers (CNN)**: Detect local patterns like edges and textures in im-

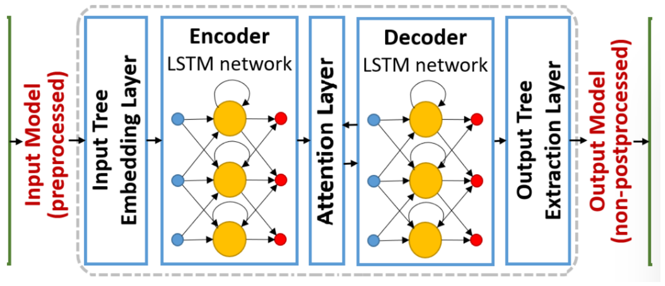
Input Model (preprocessed)
Input Tree Embedding Layer
Encoder
LSTM network
Attention Layer
Decoder
LSTM network
Output Tree Extraction Layer
Output Model (non-postprocessed)

Figure 27: A Generic LSTM-Based Neural Network Architecture to infer Heterogeneous Model Transformations

ages.

- **Recurrent Layers (RNN, LSTM)**: Process sequential data such as time-series or language.
- **Fully Connected Layers**: Each neuron is connected to every neuron in the previous layer, helping to make final predictions.
- **Activation Layers**: Introduces non-linearities, making it possible for the model to learn complex patterns (e.g., ReLU, Sigmoid).
- **Dropout Layers**: Prevent overfitting by randomly "dropping" neurons during training.
- **Output Layer**: Produces the final prediction or classification. For classification tasks, a Softmax or Sigmoid activation is used to produce probabilities.

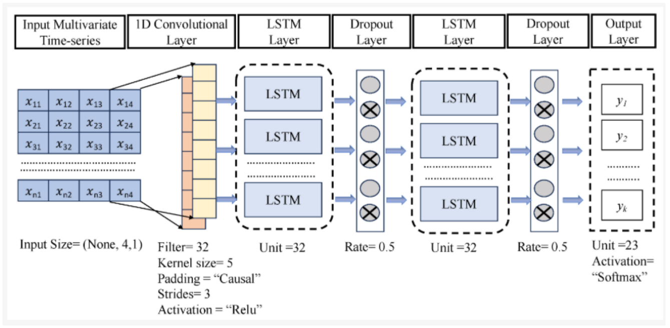

Figure 28: The proposed 1D CNN-LSTM architecture

6. **TensorFlow**
TensorFlow is an open-source deep learning framework developed by Google that allows developers to build and train deep learning models. It is highly flexible, scalable, and widely used for creating neural networks and large-scale machine learning models[10]. TensorFlow supports both CPU and GPU computations, which is crucial for training large models on vast datasets. Key features of TensorFlow include:

   - **Tensor**: The fundamental unit in TensorFlow, representing multi-dimensional arrays or matrices.
   - **Keras API**: A high-level neural networks API within TensorFlow, making it easier to build and train models with simple syntax.
   - **Model Deployment**: TensorFlow provides tools to deploy models in production environments across platforms, from mobile devices to large-scale servers.
   - **Distributed Computing**: TensorFlow supports distributed training, allowing models to be trained faster by leveraging multiple processors or machines.

   TensorFlow has become one of the most popular frameworks for deep learning, providing an efficient and comprehensive ecosystem for building, training, and deploying models.

-End-

[10] tensorflow.org TensorFlow makes it easy to create ML models that can run in any environment. Learn how to use the intuitive APIs through interactive code samples.

References

[1] Open AI. Chatgpt. Version 4, 2025.

[2] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012, 2012.

[3] Richard Szeliski. *Computer Vision: Algorithms and Applications*, volume 2nd Editional. 2022 Springer, 2021.