About Technoglobe

Technoglobe is Leading IT Training Company of India working for IT Training, Skilling & Placement

of Students since year 2001. Technoglobe has trained & placed a huge number of students in

various sectors like Digital Marketing, Graphic Designing, Accounting, Video Editing, Web

Development with Java Python & PHP, Data Analytics, Data Sciences, Adv Excel, Networking, Cyber Security, Devops, Generative AI & many more technologies.

It has been awarded more than 30 times for its Quality Education & Placements at National &

International platforms. It is one of the very few IT Training Companies in India that are awarded at **Oxford University UK**. Technoglobe has more than 100+ centers in India, UAE, UK, Canada & Singapore.

As part of its Strong Placement Support Technoglobe has done 500+ tie ups with various IT & Non

IT companies & adding more companies to it.

If you are not willing to learn, no one can help you. If you are determined to learn, no one can stop you.

Message from Team Technoglobe

Dear Students,

IT skilling is crucial for India as it significantly contributes to the nation's economic growth by powering the rapidly expanding IT sector, generating substantial employment opportunities, driving innovation, and enabling India to compete effectively in the global market, making it one of the key. pillars of the Indian economy

Skilled IT professionals are essential for driving innovation in various sectors, including IT,

healthcare, finance, Banking and manufacturing through technology adoption.

We at Technoglobe bridge the gap between the requirement of companies & skills of the students.

Our job oriented Training programs makes the students employable & industry ready.

About the Book

This book is a comprehensive self-learning guide created to empower aspiring and professional data science enthusiasts with the essential knowledge, tools, and techniques needed in today's rapidly growing data-driven world. Whether you are a beginner taking your first step into the world of data science or an experienced learner aiming to upgrade your analytical and modeling skills, this guide provides step-by-step learning enriched with practical exercises and real-world datasets.

Covering the complete data science lifecycle, the book explores core concepts such as data collection, data cleaning, exploratory data analysis (EDA), data visualization, statistics, machine learning, deep learning, and data engineering fundamentals. It also introduces advanced areas including natural language processing (NLP), computer vision, big data analytics, business intelligence, cloud-based ML platforms, and modern AI-driven solutions.

Each chapter is structured to gradually strengthen your understanding with hands-on labs, coding exercises, dataset-based projects, and real industry case studies. You will gain proficiency in using industry-standard tools and platforms such as Python, R, SQL, Jupyter Notebook, Pandas, NumPy, Scikit-learn, TensorFlow, PyTorch, Power BI, Tableau, Hadoop, Spark, and cloud platforms like AWS, Azure, and Google Cloud.

Aligned with the latest trends in artificial intelligence, predictive analytics, automation, and digital transformation, this book ensures you become job-ready, analytically strong, and capable of solving real-world problems through data. Whether your goal is to become a data analyst, data scientist, machine learning engineer, AI specialist, business intelligence analyst, or data engineer, this guide provides the strong foundation and advanced insights required to succeed in the data science domain.

This guide is developed by Technoglobe, a leading IT and multimedia training institute awarded for Quality Education and Placements, with over 100+ centers and a legacy of training thousands of students since 2001.

Index

Chapter 1: Introduction to Data Science

1.1 What is Data Science?

Data Science is the field of study that combines **mathematics**, **statistics**, **computer science**, and **domain knowledge** to extract meaningful insights from data. In today's digital world, vast amounts of data are generated every second — from social media posts and financial transactions to health records and sensor outputs. Data Science helps us make sense of this data to solve real-world problems.

1.2 Why is Data Science Important?

Data has become the "new oil." Organizations use data science to:

- Understand customer behaviour
- Improve business operations
- Detect fraud
- Recommend products (like Netflix or Amazon)
- Predict future trends
- Build smart systems (like self-driving cars or AI assistants)

1.3 Components of Data Science

1. **Data Collection**: Gathering data from various sources (web, databases, files, APIs).
2. **Data Cleaning**: Removing errors, duplicates, and missing values.
3. **Exploratory Data Analysis (EDA)**: Understanding data patterns using statistics and visualizations.
4. **Feature Engineering**: Creating new variables or features to improve model performance.
5. **Model Building**: Using algorithms (like regression, classification) to train models.
6. **Evaluation**: Testing model accuracy and performance.
7. **Deployment**: Putting the model into real-world applications.

1.4 Skills Required for Data Science

To become a data scientist, you typically need to know:

- **Programming**: Python or R
- **Statistics & Probability**
- **Data Visualization**: Using tools like Matplotlib, Seaborn, Tableau, or Power BI

- **Machine Learning**: Algorithms and models
- **SQL**: To interact with databases
- **Big Data Tools**: Hadoop, Spark (optional for advanced use cases)

1.5 Who Uses Data Science?

Industries that rely heavily on data science:

- **Healthcare**: Predict diseases, personalize treatments
- **Finance**: Risk modelling, fraud detection
- **Retail**: Recommendation engines, sales forecasting
- **Marketing**: Customer segmentation, campaign analysis
- **Manufacturing**: Quality control, predictive maintenance
- **Sports**: Performance analysis, game strategies

1.6 Real-World Examples

- **Netflix** uses data science to recommend shows and movies.
- **Google Maps** uses real-time data to suggest fastest routes.
- **Banks** use data science for loan approvals and fraud detection.
- **E-commerce websites** like Amazon use it to recommend products and manage inventory.

1.7 Career Roles in Data Science

- **Data Analyst**
- **Data Scientist**
- **Machine Learning Engineer**
- **Data Engineer**
- **Business Intelligence Analyst**

Chapter 2: Python Basics and Setup

Introduction

Before writing powerful programs in Python, it is essential to set up the environment properly and understand the basics. In this chapter, we will learn how to install Python, set up an editor, and explore some of the most fundamental concepts of Python programming: variables, data types, type casting, input/output operations, comments, and formatted strings (f-strings). By the end of this chapter, you will be able to write your first small Python program that interacts with the user.

2.1 Installing Python

Python is a free, open-source programming language.

Steps to install Python:

1. Go to the official website: https://www.python.org/downloads/.
2. Download the latest stable version for your operating system (Windows, Mac, or Linux).
3. During installation, make sure to **check the option "Add Python to PATH."**
4. After installation, open Command Prompt (Windows) or Terminal (Mac/Linux) and type:

python --version

If installation was successful, it will show the installed version, for example:

Python 3.12.2

2.2 Installing pip

pip is Python's package manager, used to install and manage external libraries.

- To check if pip is installed, run:

pip --version

- If not installed, download it from the pip documentation.
  Most of the time, pip is installed automatically with Python.

2.3 Setting Up an IDE

Although Python programs can be written in any text editor, it is better to use an **IDE (Integrated Development Environment)** or **code editor**.

- **Visual Studio Code (VS Code):**
  A lightweight but powerful editor. Install Python extension for best experience.
  Download: https://code.visualstudio.com/

- **Jupyter Notebook:**
  Very popular among beginners and data scientists because it allows running code step by step.
  Install it using:
- pip install notebook
- jupyter notebook

This will open in your browser.

2.4 Variables

A **variable** is a name that refers to a value stored in memory. You can think of it as a box where you keep information.

Example:

name = "Alice"

age = 25

Here:

- name stores a string "Alice".
- age stores an integer 25.

Rules for variable names:

1. Must begin with a letter or underscore (not a number).
   - ✅ age1
   - ❌ 1age
2. Case-sensitive: Name and name are different.
3. Use descriptive names (recommended: first_name, not fn).

2.5 Data Types

Python has different **data types** to represent different kinds of values.

Data Type	Example	Description
int (integer)	x = 10	Whole numbers
float	y = 3.14	Numbers with decimals
str (string)	name = "Alice"	Text data
bool (boolean)	is_active = True	True/False values

2.6 Type Casting

Sometimes, you need to convert one data type into another. This is called **type casting**.

```
x = "100"     # string
y = int(x)    # convert to integer
z = float(x)  # convert to float
```

2.7 Input and Output

- **Input:** Used to take data from the user.

```
name = input("Enter your name: ")
```

- **Output:** Used to display results.

```
print("Hello", name)
```

2.8 Comments

Comments are used to explain code. They are ignored by Python.

- **Single-line comment:**

```
# This is a comment
```

- **Multi-line comment:**

```
"""
This is
a multi-line
comment
"""
```

2.9 f-strings

f-strings are a modern and convenient way to format strings with variables.

```
name = "John"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

Output:

My name is John and I am 21 years old.

2.10 Practice Program

Let’s combine everything we have learned into a small program.

Problem: Ask the user for their name and age, and print a personalized message.

Code:

```
# Taking input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Printing personalized message
print(f"Hello {name}, you are {age} years old. Welcome to Python learning!")
```

Sample Output:

```
Enter your name: Neha
Enter your age: 22
Hello Neha, you are 22 years old. Welcome to Python learning!
```

Summary

- Python must be installed from the official website, and pip is used to install libraries.
- IDEs like VS Code and Jupyter Notebook make coding easier.
- Variables store data, and Python supports multiple data types such as int, float, str, and bool.
- Type casting converts data from one type to another.
- input() is used to take user input, and print() is used for output.
- Comments (# or """...""") are ignored by Python but help humans understand the code.
- f-strings make string formatting simple and powerful.

Chapter 3: Operators & String Functions in Python

3.1 Introduction

Every programming language provides tools to perform operations on data. In Python, these tools come in two powerful forms:

1. **Operators** – Special symbols or keywords that perform operations on variables and values.
2. **String Functions (Methods)** – Built-in functions that allow us to modify, analyze, and work with text.

Understanding operators and mastering string functions will give you strong problem-solving abilities in Python.

3.2 Operators in Python

Python supports **7 categories of operators**.

1. Arithmetic Operators

Used for mathematical operations.

Operator	Description	Example	Output
+	Addition	5 + 3	8
-	Subtraction	10 - 4	6
*	Multiplication	6 * 2	12
/	Division	15 / 2	7.5
//	Floor Division	15 // 2	7
%	Modulus (Remainder)	15 % 4	3
**	Exponent (Power)	2 ** 3	8

2. Comparison (Relational) Operators

Compare two values and return True or False.

Operator	Example	Output
==	5 == 5	True
!=	5 != 3	True

>	7 > 3	True
<	7 < 3	False
>=	5 >= 5	True
<=	5 <= 3	False

3. Logical Operators

Used to combine conditions.

Operator	Example	Output
and	(5 > 3 and 10 > 7)	True
or	(5 > 10 or 7 > 3)	True
not	not(5 > 3)	False

4. Bitwise Operators

Work at the **binary (bit) level**.

Operator	Example (a=5, b=3)	Binary Work	Result
&	5 & 3	101 & 011	1
`	`	`5	3`
^	5 ^ 3	101 ^ 011	6
~	~5	-(5+1)	-6
<<	5 << 1	101 → 1010	10
>>	5 >> 1	101 → 10	2

5. Assignment Operators

Assign values to variables.

Operator	Example	Equivalent To
=	x = 5	Assign value
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 2	x = x * 2

/=	x /= 2	x = x / 2
//=	x //= 2	x = x // 2
%=	x %= 2	x = x % 2
**=	x **= 3	x = x ** 3

6. Identity Operators

Check whether two objects are the same in memory.

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]


print(x is y)      # True (same object)
print(x is z)      # False (different objects, even if values same)
print(x is not z)  # True
```

7. Membership Operators

Check if a value exists in a sequence (string, list, tuple, etc.).

```
text = "python"
print("p" in text)      # True
print("z" not in text)   # True
```

3.3 String Functions in Python

Strings are widely used in programming. Python offers a **rich set of built-in string methods**.

Here are the **most commonly used string functions**:

Method	Description	Example	Output
upper()	Converts to uppercase	"hello".upper()	"HELLO"
lower()	Converts to lowercase	"HELLO".lower()	"hello"
title()	Capitalizes each word	"python programming".title()	"Python Programming"

strip()	Removes leading & trailing spaces	" python ".strip()	"python"
lstrip()	Removes spaces from left	" python".lstrip()	"python"
rstrip()	Removes spaces from right	"python ".rstrip()	"python"
split()	Splits into list (default space)	"a,b,c".split(",")	['a', 'b', 'c']
join()	Joins list into string	" ".join(['I','love','Python'])	"I love Python"
find()	Finds first occurrence of substring	"python".find("th")	2
replace()	Replaces substring	"python".replace("py","java")	"javathon"
startswith()	Checks if string starts with given value	"python".startswith("py")	True
endswith()	Checks if string ends with given value	"python".endswith("on")	True
count()	Counts occurrences	"banana".count("a")	3
isdigit()	Checks if all chars are digits	"123".isdigit()	True
isalpha()	Checks if all chars are alphabets	"abc".isalpha()	True
isalnum()	Checks if all chars are letters/numbers	"abc123".isalnum()	True
capitalize()	First letter uppercase, rest lowercase	"hello".capitalize()	"Hello"

3.4 Practice Exercise

Problem: Analyze a String

Write a Python program that:

1. Takes a string from the user.
2. Counts how many **vowels** (a, e, i, o, u) are present.
3. Counts how many **digits** are present.

Solution:

```
text = input("Enter a string: ")

vowels = "aeiouAEIOU"

vowel_count = 0

digit_count = 0


for char in text:

    if char in vowels:

        vowel_count += 1

    elif char.isdigit():

        digit_count += 1


print(f"Vowels: {vowel_count}")

print(f"Digits: {digit_count}")
```

Sample Run:

```
Enter a string: Python 3.10

Vowels: 2

Digits: 3
```

3.5 Summary

- Python provides **7 operator categories**: Arithmetic, Comparison, Logical, Bitwise, Assignment, Identity, Membership.
- Strings can be manipulated using powerful **built-in methods** such as upper(), lower(), title(), split(), join(), find(), replace(), and many more.
- Operators + String functions together allow us to process and analyze data effectively

Chapter 4: String Manipulation & Control Statements

4.1 Introduction

Programming is not just about writing instructions; it is about **making smart decisions** and **manipulating data**.
In this chapter, we will explore two important areas:

1. **String Manipulation** → Extracting, slicing, and rearranging text.
2. **Control Statements** → Making decisions in programs using conditions.

Together, these skills allow you to build interactive and logical Python programs.

4.2 String Manipulation

4.2.1 Indexing in Strings

- Strings are sequences of characters.
- Every character has an **index (position)**.
- Indexing in Python starts from **0 (left to right)**.
- Negative indexing starts from **-1 (right to left)**.

Example:

```
text = "PYTHON"
print(text[0])   # P  (first character)
print(text[3])   # H  (fourth character)
print(text[-1])  # N  (last character)
print(text[-3])  # H  (third from last)
```

Visualization:

```
 P  Y  T  H  O  N
 0  1  2  3  4  5
-6 -5 -4 -3 -2 -1
```

4.2.2 Slicing in Strings

Slicing extracts a **substring** from a string.

Syntax:

string[start:end:step]

- start → index where slice begins (inclusive).
- end → index where slice stops (exclusive).
- step → interval between characters (default = 1).

Examples:

```
word = "PYTHON"

print(word[0:4])    # PYTH
print(word[2:])     # THON
print(word[:4])     # PYTH
print(word[::2])    # PTO  (every 2nd character)

print(word[::-1])   # NOHTYP (reversed string)
```

☞ Slicing is widely used in tasks like reversing text, extracting substrings, and skipping characters.

4.2.3 Common String Manipulations

- **Concatenation (joining):** "Hello" + "World" → HelloWorld
- **Repetition:** "Hi" * 3 → HiHiHi
- **Membership test:** "a" in "apple" → True
- **Length:** len("Python") → 6

These operations make strings flexible to use in programs.

4.3 Control Statements

Control statements help you **make decisions** in a program.

4.3.1 The if Statement

Executes a block only when the condition is **True**.

```
x = 10
if x > 5:
    print("x is greater than 5")
```

4.3.2 The if-else Statement

Adds an alternative block if condition is **False**.

```
age = 15
if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible to vote")
```

4.3.3 The if-elif-else Statement

Used for multiple conditions.

```
marks = 72
if marks >= 90:
    print("Grade A")
elif marks >= 75:
    print("Grade B")
elif marks >= 50:
    print("Grade C")
else:
    print("Fail")
```

4.3.4 Nested if

An if inside another if.

```
num = 25
if num > 0:
    if num % 2 == 0:
        print("Positive Even")
    else:
        print("Positive Odd")
else:
    print("Negative Number")
```

4.3.5 Logical Operators in Conditions

Operator	Meaning	Example	Result
and	True if **both** conditions are true	(x>5 and x<10)	True
or	True if **at least one** condition is true	(x>5 or x<2)	True
not	Reverses the condition	not(x>5)	False

Example:

```
age = 20
citizen = True
if age >= 18 and citizen:
    print("Eligible to vote")
```

4.4 Indentation in Python

Unlike other languages that use { } for blocks, Python uses **indentation** (spaces or tabs). ☞ Incorrect indentation will give an **IndentationError**.

Correct Example:

```
if True:
    print("This is indented correctly")
```

Incorrect Example:

```
if True:
print("Error!")   # ✗ Wrong indentation
```

Best Practice: Always use **4 spaces** per indentation level.

4.5 Practice Problems

Problem 1: Tax Calculator (Using if-elif)

- Income ≤ 2,50,000 → No Tax
- Income ≤ 5,00,000 → 5% Tax
- Income ≤ 10,00,000 → 20% Tax
- Income > 10,00,000 → 30% Tax

Solution:

```
income = int(input("Enter annual income: "))
if income <= 250000:
    tax = 0
elif income <= 500000:
        tax = income * 0.05
    elif income <= 1000000:
    tax = income * 0.20
else:
    tax = income * 0.30
print("Tax to be paid:", tax)
```

Problem 2: Largest of Three Numbers

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if a >= b and a >= c:
                print("Largest:", a) elif b >= a and b >= c:
    print("Largest:", b)
else:
    print("Largest:", c)
```

Problem 3: Leap Year Checker

A leap year is divisible by **400**, OR divisible by **4 but not by 100**.

```
year = int(input("Enter year: "))
if (year % 400 == 0) or (year % 4 == 0 and year % 100 != 0):
    print("Leap Year")
else:
    print("Not a Leap Year")
```

Problem 4: Password Strength Checker

```
password = input("Enter password: ")
if len(password) < 6:
    print("Weak: Too short")
elif password.isalpha() or password.isdigit():
    print("Weak: Use both letters &
numbers") else:
    print("Strong password")
```

Problem 5: Movie Ticket Discount System

- Age < 12 → 50% discount
- Age between 12–18 → 25% discount
- Age > 18 → No discount

```
age = int(input("Enter your age: "))
if age < 12:
    print("You get 50% discount")
elif age <= 18:
    print("You get 25% discount")
else:
    print("No discount")
```

4.6 Summary

- **Indexing** → Access single characters (s[0], s[-1]).
- **Slicing** → Extract substrings (s[1:4], s[::-1]).
- **Control Statements** → if, if-else, if-elif-else, nested if.
- **Logical Operators** → Combine conditions (and, or, not).
- **Indentation** → Python uses indentation instead of braces.
- Practice Problems showed real-world applications like Tax Calculator, Leap Year, Largest Number, Password Strength Checker, Ticket Discounts.

Chapter 5 : Loops with Conditions and Control

5.1 Recap of Loops

- A **loop** executes a block of code multiple times.
- **for loop** → Best when we know the number of iterations.
- **while loop** → Best when we don't know how many times but depend on a condition.

Now let's combine them with **conditions** and **control statements** (break, continue), plus **nested loops**.

5.2 for Loop with Conditions

We can use **if-else conditions inside a for loop** to make decisions while iterating.

Example 1: Print Even Numbers

```
for i in range(1, 11):
    if i % 2 == 0:   # condition for even
        print(i, "is Even")
    else:
        print(i, "is Odd")
```

Output:

1 is Odd
2 is Even
3 is Odd
...
10 is Even

5.3 for Loop with break

The break statement **immediately exits the loop**. **Example 2: Stop Loop at 5**

```
for i in range(1, 11):
    if i == 5:
        print("Breaking the loop at", i)
        break
```

```
    print(i)
```

Output:

1

2

3

4

Breaking the loop at 5

5.4 for Loop with continue

The continue statement **skips the current iteration** and moves to the next one.

Example 3: Skip Multiples of 3

```
for i in range(1, 11):
    if i % 3 == 0:
        continue
    print(i)
```

Output:

1

2

4

5

7

8

10

5.5 Nested for Loops

A for loop can run inside another loop.

Example 4: Multiplication Table (1 to 3)

```
for i in range(1, 4):        # Outer loop
    for j in range(1, 6):    # Inner loop
```

```
        print(f"{i} x {j} = {i*j}")
    print("----")
```

Output:

```
1 x 1 = 1
1 x 2 = 2
...
1 x 5 = 5
----
2 x 1 = 2
...
3 x 5 = 15
```

5.6 while Loop with Conditions

The while loop continues **as long as the condition is true**. **Example 5: Print Numbers Until 10**

```
num = 1
while num <= 10:
    print(num)
    num += 1
```

Output:

```
1
2
...
10
```

5.7 while Loop with break

We can exit the loop before the condition naturally ends. **Example 6: Stop at 7**

```
num = 1
while num <= 10:
    if num == 7:
        print("Breaking at", num)
        break
    print(num)
```

```
    num += 1
```

Output:

```
1

2

3

4

5

6
```

Breaking at 7

5.8 while Loop with continue

We can skip certain iterations.

Example 7: Skip Even Numbers

```
num = 0

while num < 10:
    num += 1

    if num % 2 == 0:
        continue
    print(num)
```

Output:

```
1
3

5

7
9
```

5.9 Nested while Loops

Just like for, we can nest while loops.

Example 8: Pattern Printing

```
i = 1

while i <= 5:
```

```
j = 1
while j <= i:
    print("*", end=" ")
    j += 1
print()
i += 1
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

5.10 Comparison of for and while with Conditions

Feature	for Loop	while Loop
Iterations	Fixed/known	Unknown/depends on condition
Condition Placement	Often inside loop body	At loop header
Best Use	Tables, lists, strings	User input, waiting until a condition

5.11 Summary

- Both **for** and **while** loops allow repetition.
- **Conditions inside loops** make them flexible.
- **break** → exit loop immediately.
- **continue** → skip current iteration.
- **Nested loops** → solve patterns, tables, and multidimensional problems.
- Choose **for** when iterations are known, and **while** when they depend on a condition.

Chapter 6: Try-Except & Error Handling

6.1 Why Error Handling?

Imagine you are building a calculator program. A user tries to divide a number by zero. What happens? The program suddenly crashes with an error message. This is not a good experience for the user.

In real-world programs, such errors are common:

- A user types letters instead of numbers.
- A program tries to open a file that does not exist.
- An index outside the list's range is accessed.

These errors are called **exceptions**.
Python provides a mechanism called **exception handling** that allows us to deal with such situations gracefully, without crashing the program.

6.2 The try-except Block

The **basic building block** of error handling in Python is the try-except structure.

How It Works

- Python first runs the code inside the try block.
- If an error occurs, Python stops and jumps to the except block.
- If no error occurs, the except block is skipped.

Syntax

```
try:
    # Code that may raise an error
except SomeError:
    # Code to handle the error
```

Example 1: Division by Zero

```
try:
    result = 10 / 0
    print(result)
except ZeroDivisionError:
```

```
    print("You cannot divide by zero!")
```

Output:

You cannot divide by zero!

Without try-except, this program would crash. With handling, the program continues smoothly.

6.3 Handling Multiple Exceptions

A single program may face different types of errors. Python allows multiple except blocks to handle them individually.

Example 2: Handling Different Errors

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

Possible Scenarios

- If user enters 5 → Program prints Result: 2.0
- If user enters abc → Program prints Invalid input! Please enter a number.
- If user enters 0 → Program prints You cannot divide by zero!

6.4 Adding else

Python also provides an **else block** with try-except.

- The else block runs **only if no error occurs**.

Example 3: Try-Except-Else

```
try:
    num = int(input("Enter a number: "))
    print("You entered:", num)
except ValueError:
    print("Invalid input!")
```

```
else:
    print("Conversion successful!")
```

Input: 10
Output:

You entered: 10

Conversion successful!

6.5 The finally Block

Sometimes, we want a block of code to run **no matter what happens**.
That's where the finally block comes in. It is often used for cleanup tasks such as closing files or releasing resources.

Example 4: Try-Except-Finally

```
try:
    f = open("sample.txt", "r")
    data = f.read()
    print(data)
except FileNotFoundError:
    print("File not found!")
finally:
    print("Execution finished.")
```

If the file does not exist, the output will be:

File not found!

Execution finished.

Notice how finally runs regardless of the error.

6.6 Raising Exceptions

Sometimes, we want to **create an error on purpose** when certain conditions are not met.
For this, Python provides the **raise keyword**.

Example 5: Raise Custom Error

```
age = int(input("Enter your age: "))
if age < 18:
    raise ValueError("You must be at least 18 years old.")
```

```
else:
    print("Access granted!")
```

Input: 15
Output:

ValueError: You must be at least 18 years old.

6.7 Common Exceptions in Python

Exception	Cause	Example
ValueError	Wrong type of value	int("abc")
ZeroDivisionError	Dividing by zero	10 / 0
TypeError	Wrong data type operation	"abc" - 2
FileNotFoundError	File does not exist	open("xyz.txt")
IndexError	Accessing invalid list index	arr[10] when list has 5 items
KeyError	Accessing invalid dictionary key	mydict["unknown"]

6.8 Key Takeaways

- **Exceptions** are runtime errors that can crash a program.
- **try** → block of code to test.
- **except** → block of code to handle error.
- **else** → executes when no error occurs.
- **finally** → executes always, useful for cleanup.
- **raise** → used to throw custom exceptions.
- Good error handling makes a program **robust, user-friendly, and safe**.

Chapter 7: Data Containers in Python

7.1 Introduction

In real life, we often need to **store collections of items**. For example:

- A list of students in a class.
- Marks of subjects for each student.
- A phonebook with names and numbers.

Python provides us with **data containers** to manage such data easily.
The four most important containers are:

1. **List** – Ordered and changeable.
2. **Tuple** – Ordered but unchangeable.
3. **Set** – Unordered and unique.
4. **Dictionary** – Key-value pairs.

7.2 Lists

A **list** is an ordered, mutable collection of elements.
It can store numbers, strings, or even other lists.

Creating a List

fruits = ["apple", "banana", "mango", "grapes"]

Properties of Lists

- ✅ Ordered → items maintain their sequence.
- ✅ Mutable → items can be changed.
- ✅ Allows duplicates.

All List Methods

Method	Description	Example
append(x)	Add element at end	lst.append(10)
extend(iterable)	Add multiple elements	lst.extend([20, 30])
insert(i, x)	Insert at index i	lst.insert(1, 99)

remove(x)	Remove first occurrence of x	lst.remove(10)
pop(i)	Remove and return element at index i	lst.pop(2)
clear()	Remove all elements	lst.clear()
index(x)	Return index of first occurrence of x	lst.index(20)
count(x)	Count occurrences of x	lst.count(10)
sort(reverse=False)	Sort ascending (or descending if reverse=True)	lst.sort()
reverse()	Reverse order	lst.reverse()
copy()	Return shallow copy	new = lst.copy()

Example

```
numbers = [10, 20, 30, 10]


numbers.append(40)     # [10, 20, 30, 10, 40]
numbers.insert(1, 15)  # [10, 15, 20, 30, 10, 40]
numbers.remove(10)     # removes first 10 → [15, 20, 30, 10, 40]
print(numbers.pop())   # 40 (removed)
print(numbers.index(30))  # 2
print(numbers.count(10))  # 1
numbers.sort()         # [10, 15, 20, 30]
numbers.reverse()      # [30, 20, 15, 10]
```

7.3 Tuples

A **tuple** is like a list, but **immutable** (unchangeable).

Creating a Tuple

```
colors = ("red", "green", "blue")
```

Properties of Tuples

- ✅ Ordered.
- ❌ Immutable (cannot change).
- ✅ Allows duplicates.

Tuple Methods

Since tuples are immutable, only 2 methods exist:

Method	Description	Example
count(x)	Count occurrences	(1,2,2,3).count(2) → 2
index(x)	Index of first occurrence	(1,2,3).index(3) → 2

Example

```
days = ("Mon", "Tue", "Wed", "Tue")

print(days.count("Tue"))  # 2

print(days.index("Wed"))  # 2
```

7.4 Sets

A **set** is an unordered collection of **unique elements**.

Creating a Set

```
fruits = {"apple", "banana", "mango"}
```

Properties of Sets

- ❌ Unordered (no indexing).
- ❌ No duplicates allowed.
- ✅ Supports mathematical set operations.

All Set Methods

Method	Description	Example
add(x)	Add element	s.add(10)
update(iterable)	Add multiple elements	s.update([20,30])
remove(x)	Remove element; error if not found	s.remove(10)
discard(x)	Remove element; no error if not found	s.discard(50)
pop()	Remove & return random element	s.pop()

clear()	Remove all elements	s.clear()
union(s2)	Combine sets	s1.union(s2)
intersection(s2)	Common elements	s1.intersection(s2)
difference(s2)	Elements in s1 but not in s2	s1.difference(s2)
symmetric_difference(s2)	Elements in either but not both	s1.symmetric_difference(s2)
issubset(s2)	Check if subset	s1.issubset(s2)
issuperset(s2)	Check if superset	s1.issuperset(s2)
isdisjoint(s2)	True if no common items	s1.isdisjoint(s2)
copy()	Shallow copy	s.copy()

Example

a = {1, 2, 3}

b = {3, 4, 5}

a.add(6) # {1,2,3,6}

a.update([7,8]) # {1,2,3,6,7,8}

print(a.union(b)) # {1,2,3,4,5,6,7,8}

print(a.intersection(b)) # {3}

print(a.difference(b)) # {1,2,6,7,8}

print(a.symmetric_difference(b)) # {1,2,4,5,6,7,8}

7.5 Dictionaries

A **dicWionary** stores **key-value pairs**.

CreaWing a DicWionary

student = {"name": "Amit", "age": 21, "course": "Python"}

ProperWies of DicWionaries

- ✅ Keys are unique.
- ✅ Values can be duplicate.
- ✅ Mutable.

All Dictionary Methods

Method	Description	Example
get(key, default)	Return value (or default)	d.get("age", 0)
keys()	Return all keys	d.keys()
values()	Return all values	d.values()
items()	Return key-value pairs	d.items()
update({})	Update dictionary	d.update({"age":25})
pop(key)	Remove key-value	d.pop("age")
popitem()	Remove last inserted pair	d.popitem()
clear()	Clear dictionary	d.clear()
setdefault(key, default)	Insert if key not present	d.setdefault("city","Delhi")
copy()	Shallow copy	d.copy()
fromkeys(keys, value)	Create new dict	dict.fromkeys(["a","b"],0)

Example

```
student = {"name": "Amit", "age": 21}


print(student.get("name"))       # Amit
student.update({"city": "Delhi"}) # {"name":"Amit","age":21,"city":"Delhi"}
student.setdefault("course","Python")
print(student.keys())          # dict_keys(['name','age','city','course'])
```

7.7 Key Takeaways

- **List**: Ordered, mutable, allows duplicates.
- **Tuple**: Ordered, immutable, allows duplicates.
- **Set**: Unordered, unique, supports union & intersection.
- **Dictionary**: Key-value pairs, mutable, fast lookups.
- Choosing the right data container depends on the **type of problem**.

Chapter 8: Functions in Python (Advanced Concepts)

8.1 Introduction

Functions are one of the **building blocks** of Python programming.
They allow us to **organize code**, **avoid repetition**, and **make programs modular and reusable**.

Imagine a calculator. Instead of rewriting the logic for addition, subtraction, or multiplication again and again, we put them inside **functions**.
Then whenever we need them, we just **call the function**.

8.2 What is a Function?

Definition:
A function is a **block of code** that performs a specific task, can take inputs, and may return an output.

Syntax:

```
def function_name(parameters):

    """Optional docstring"""

    # function body

    return value
```

- def → keyword to define function
- function_name → unique name for the function
- parameters → inputs (optional)
- return → output (optional)

Example 1: Simple Function

```
def greet():

    print("Hello, welcome to Python functions!")
```

Dry Run:

1. Function defined with name greet.

2. Call greet() → executes function body.
3. Prints → "Hello, welcome to Python functions!".

8.3 Why Functions?

1. **Code Reusability** – Write once, use many times.
2. **Modularity** – Break big program into smaller parts.
3. **Readability** – Makes code easier to understand.
4. **Debugging** – Easier to test smaller parts.

Without functions = Messy code
With functions = Clean, professional code

8.4 Types of Functions

1) Built-in Functions

Already available in Python (e.g., print(), len(), sum()).

2) User-defined Functions

Created by programmers using def.

3) Lambda Functions

Small anonymous (nameless) functions using lambda.

8.5 Function with Parameters

Functions can take **inputs** (parameters/arguments).

Example 2: Function with Parameters

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


print(add(5, 3))   # Output: 8
```

Dry Run:

1. Call add(5,3) → assigns a=5, b=3.
2. Executes return a+b → gives back 8.
3. print() displays 8.

8.6 Default Arguments

Python allows assigning **default values** to parameters.

```
def greet(name="User"):
    print(f"Hello, {name}!")


greet()         # Hello, User!
greet("Alice")   # Hello, Alice!
```

8.7 Return Statement

The return keyword **sends a value back** to the function caller.

If no return, the function returns None.

```
def square(x):
    return x * x


result = square(4)
print(result)   # Output: 16
```

8.8 Difference Between return and print

Many beginners confuse return and print.

Feature	return	print
Purpose	Sends the result back to caller	Displays value on console
Reusability	Value can be stored, reused, passed further	Value cannot be reused directly
Stops Function?	Yes, ends function immediately	No, function continues after print
Where Used?	Inside functions (commonly)	Anywhere in code

Example with return

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


result = add(5, 3)
print(result)   # Output: 8
```

✅ **Example with print**

```
def add(a, b):
    print(a + b)


result = add(5, 3)   # Output: 8
print(result)        # Output: None
```

Key Point:

- Use **return** when you need the result for further processing.
- Use **print** when you just want to display something.

8.9 Variable Scope

1. **Local Variables** – Declared inside a function, used only there.
2. **Global Variables** – Declared outside, accessible everywhere.

```
x = 10   # Global variable


def test():
    x = 5   # Local variable
    print("Inside:", x)


test()         # Inside: 5
print("Outside:", x)  # Outside: 10
```

8.10 Nested Functions

A function can be defined **inside another function**.

Example

```
def outer():
    def inner():
        print("Inner function executed")
    print("Outer function executed")
    inner()
outer()
```

Dry Run:

- м. 2dzűŜNóú ŏl·ffŜŘ Γђ LJNλyűă þhdzűŜN ŦdzyŎűλ2y ŜĖŜŏdzűŜŘþ.
- н. / l·ffă λyŷŜNóú Γђ LJNλyűă þLyŷŜN ŦdzyŎűλ2y ŜĖŜŏdzűŜŘþ.

8.11 Lambda Functions

Definition:
! fl·YōŘl· ŦdzyŎűλ2yλă l· **small anonymous function** ØNλűűŜyλy2yŜ flyŜ.

Syntax:

fl·YōŘl· l·N∃dzYŜyűăY ŜĖLJNŜăăλ2y

Example

ăljdzl·NŜ Γ fl·YōŘl· ĖY ĖFĖ

LJNλyűóăljdzl·NŜópúú l hdzűLJdzűY нp

8.12 Nested Lambda Functions

¸2dz ŏl·yLJfl·ŎŜ **one lambda inside another.**

l·ŘŘψűKŜyψăljdzl·NŜ Γ fl·YōŘl· l·Σ ōY ófl·YōŘl· ĖY ĖFĖúól·ƀōú

LJNλyűól·ŘŘψűKŜyψăljdzl·NŜóнΣ oúú l hdzűLJdzűY нp

Explanation:

- м. hdzűŜN fl·YōŘl· űl·ʅŜă l· l·yŘ ō.
- н. LyŷŜN fl·YōŘl· ăljdzl·NŜă ól·ƀōú.
- o. óнƀoúΓp Γђ pFpΓнp.

8.13 High-Level Examples

Example 1: Calculator using Functions

ŘŜŦ l·ŘŘól·Σōúy NŜűdzNyl·ƀō

ŘŜŦ ădzōól·Σōúy NŜűdzNyl·πō

ŘŜŦ Ydzfól·Σōúy NŜűdzNyl·Fō

ŘŜŦ ŘλØól·Σōúy NŜűdzNyl·κō λŦ ōHΓл ŜfăŜ þ5λØλăλ2yōè ÎŜN2Hþ

LJNλyűól·ŘŘómлΣpúú l мp

LJNλyűóădzōómлΣpúú l p

LJNλyűóYdzfómлΣpúú l рл

[illegible]

Example 2: Using Lambda + Map, Filter, Reduce

[illegible]

[illegible]

[illegible]

[illegible]

Example 3: Nested Function for Authentication

[illegible]

[illegible]

8.14 Functional Programming Tools in Python

tèűK2ƴLJN2ØλŘŜă ă2ϒŜ **powerful functional programming utilities**Y

- ϒl·LJóú Γђ ¢Nl·yã₮2NϒŘl·ű̇l·
- ₮λtűŜNóú Γђ {ŜtŜŎű Řl·ű̇l·
- NŜŘdzŎŜóú Γђ !∃∃NŜ∃l·űŜ Řl·ű̇l·

¢KŜăŜ l·ff2ø è2dz ű2 øNλűŜ **cleaner, shorter, and more efficient code**φ

8.14.1 The map() Function

Definition:
¢KŜ ϒl·LJóú ₮dzyŎűλ2ƴl·LJLJtλŜă l· ∃λØŜyϒ₮dzyŎűλ2ƴű2 **each element** 2₮ l·ƴϒλűŜNl·ōfŜ ófλ⌊Ŝ tλăű 2N ûdzLJtŜú l·yŘ NŜűdzNyã l· ySø λűŜNl·ōfŜ óϒl·LJ 2ō?ŜŎűúφ

Syntax:

ϒl·LJó₮dzyŎűλ2yΣ λűŜNl·ōfŜú

- ₮dzyŎűλ2ƴΓђ ŘŜ₮λySă øKl·ű ű2 Ř2 øλűK Ŝl·ŎK ŜfŜYŜyű
- λűŜNl·ōfŜ Γђ ăŜljdzŜyŎŜ 2₮ Øl·tdzŜă

Example 1: Square numbers using a normal function

ŘŜ₮ ăljdzl·NŜóÈúY

NŜűdzNƴÈ ₣ È

ydzϒōŜNă Γ ώмΣ нΣ oΣ пϐ

NŜădztű Γ ϒl·LJóăljdzl·NŜΣ ydzϒōŜNăú

LJNλyűófλăűóNŜădztűúú l ώмΣ пΣ фΣ мсϐ

Dry Run:

- ăljdzl·NŜómúΓм
- ăljdzl·NŜóнúΓп
- ăljdzl·NŜóoúΓф
- ăljdzl·NŜópúΓмс

Example 2: Using lambda with map

ydzϒōŜNă Γ ώмΣ нΣ oΣ пϐ

NŜădztű Γ ϒl·LJófl·ϒōŘl· ÈY ÈЪмлΣ ydzϒōŜNăú

```
print(list(result))   # [11, 12, 13, 14]
```

8.14.2 The filter() Function

Definition:
The filter() function filters elements of an iterable by applying a condition (function). It keeps only those elements for which the function returns True.

Syntax:

```
filter(function, iterable)
```

Example 1: Filter even numbers

```
def is_even(x):
    return x % 2 == 0


numbers = [1, 2, 3, 4, 5, 6]
result = filter(is_even, numbers)
print(list(result))   # [2, 4, 6]
```

Dry Run:

- is_even(1) → False → removed
- is_even(2) → True → keep
- is_even(3) → False → removed
- is_even(4) → True → keep
- Final → [2,4,6]

Example 2: Filter words longer than 4 letters

```
words = ["cat", "elephant", "dog", "tiger"]
result = filter(lambda w: len(w) > 4, words)
print(list(result))   # ['elephant', 'tiger']
```

8.14.3 The reduce() Function

Unlike map() and filter(), reduce() is in the **functools** module.

Definition:
The reduce() function applies a function **cumulatively** to items of an iterable, reducing it to a **single value**.

Syntax:

```
from functools import reduce
```

NJŜŘdzŎŜóŦdzyŎűλƨyΣ λűŜNJl·ōfŜú

✓ Example 1: Sum of all numbers

ŦNJƨY ŦdzyŎűƨƨfă λYLJƨNJú NJŜŘdzŎŜ

yűzYōŜNJă Ґ ώмΣ нΣ oΣ пΣ pϐ

NJŜădztű Ґ NJŜŘdzŎŜóťl·YōŘl· ÈΣ èY È Ь èΣ yűzYōŜNJăú

LJNJλyűóNJŜădztűú I мр

Dry Run:

- {ű̋ŜLJ мY óмЂнú Ґ o
- {ű̋ŜLJ нY óoЬoú Ґ c
- {ű̋ŜLJ oY ócЪпú Ґ мл
- {ű̋ŜLJ пY óмлЬрú Ґ мр

CλyΙ·f NJŜădztű ГЂ мр

Example 2: Find maximum value

ŦNJƨY ŦdzyŎűƨƨfă λYLJƨNJú NJŜŘdzŎŜ

yűzYōŜNJă Ґ ώмлΣ прΣ нΣ ффΣ нoϐ

NJŜădztű Ґ NJŜŘdzŎŜóťl·YōŘl· l·Σ ōY l· λŦ l· ƀ ō ŜťăŜ ōΣ yűzYōŜNJăú

LJNJλyűóNJŜădztűú I фф

8.14.4 Comparison Table

Function	Purpose	Output
Yl·LJóú	! LJLJfè ŦdzyŎűλƨyűƨ ŜØŜNJè ŜfŜYŜyű	¢NJl·yãŦƨNJYŜŘ fλăű
ŦλťűŜNJóú	YŜŜLJ ŜfŜYŜyűă YŜŜűλyƎ ŎƨyŘλűλƨy	CλťűŜNJŜŘ fλăű
NJŜŘdzŎŜóú	/ƨYōλyŜ ŜfŜYŜyűă λyűƨ ƨyŜ Øl·fdzŜ	{λyƎfŜ NJŜădztű

8.14.5 High-Level Example: Combine all three

ŦNJƨY ŦdzyŎűƨƨfă λYLJƨNJú NJŜŘdzŎ

yűzYōŜNJă Ґ ώмΣ нΣ oΣ пΣ pΣ cϐ

l {űŜLJ мY {ljdzl·ŊŜ Ŝl·ŎK yʠzϒōŜŊ óϒl·LJú

ăljdzl·ŊŜă Ґ fλăűóϒl·LJófl·ϒōŘl· ÈϒÈƑÈΣ yʠzϒōŜŊăúú

l {űŜLJ нY YŜŜLJ 2yƒè ŜØŜyˇăljdzl·ŊŜă óŦλfűŜŊú

ŜØŜyψăljdzl·ŊŜă Ґ fλăűóŦλfűŜŊófl·ϒōŘl· ÈϒȨ̂ нҐҐлΣ ăljdzl·ŊŜăúú

l {űŜLJ oY adzfűλLJfè űKŜϒ ű2ƎŜűKŜŊ óŊŜŘdzŎŜú

LJŊ2ŘdzŎű Ґ ŊŜŘdzŎŜófl·ϒōŘl· ÈΣèϒ ÈƑèΣ ŜØŜyψăljdzl·ŊŜăú

LJŊλyűóþ{ljdzl·ŊŜăϒþΣ ăljdzl·ŊŜăú l ώмΣпΣфΣмcΣнpΣocб

LJŊλyűóþ9ØŜyˇ{ljdzl·ŊŜăϒþΣ ŜØŜyψăljdzl·ŊŜăú l ώпΣмcΣocб

LJŊλyűóþ tŊ2ŘdzŎűϒþΣ LJŊ2ŘdzŎűú l нолп

Explanation:

- мΦ bdzϒōŜŊă ăljdzl·ŊŜŘ Гђ ώмΣпΣфΣмcΣнpΣocб
- нΦ hyƒè ŜØŜyˇГђ ώпΣмcΣocб
- oΦ wŜŘdzŎŜ Гђ пƑмcƒoc Ґ нолп

8.16 Summary of Chapter

- Cdzyŏűλ2yã ϒl·ʅŜ Ŏ2ŘŜ **modular, reusable, and readable**Φ
- [Ŝl·ŊyŜŘ ōl·ăλŎă Гђ ŘŜŦλyλyƎΣ LJl·ŊJl·ϒŜűŜŊăΣ ŊŜúdzŊyˇØă LJŊλyűΣ ăŎ2LJŜΦ
- 9ÈLJf2ŊŜŘ l·ŘØl·yŎŜŘ Гђ ŘŜŦl·dzfű l·ŊƎăΣ ySăűŜŘ Ŧdzyŏűλ2yãΣ fl·ϒōŘl·ăΦ
- ¦ yŘŜŊăű22Ř **map, filter, reduce** Ŧ2Ŋ Ŧdzyŏűλ2yl·f LJŊ2ƎŊl·ϒYˇλyƎΦ
- tŊl·ŎűλŎŜŘ KλƎKπfŜØŜf ŜÈl·ϒLJfŜă fλʅŜ Ŏl·fŎdzfl·ű2ŊΣ l·dzűKŜyűλŎl·ûλ2yΣ l·yŘ Řl·ûl· LJŊ2ŎŜăăλyƎΦ

.è y2øΣ è2dz ăK2dzfŘ ōŜ l·ōfŜ ű2 øŊλűŜ **professional-grade functions** λyˇ tèűK2yΦ

Chapter 9: Object-Oriented Programming (OOP) in Python

Object-Oriented Programming (OOP) is one of the most powerful ways to organize and write Python programs. Instead of writing scattered functions and variables, OOP allows you to group them into **objects** that model real-world entities.

9.1 Classes and Objects

Definition:

- A **class** is a blueprint or template for creating objects.
- An **object** is an instance of a class, representing a real-world entity.

Explanation:
Think of a **class as a recipe** and an **object as the cake** made from that recipe. The recipe can be reused to make many cakes (objects).

Example:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

car1 = Car("Honda", "City")
car2 = Car("Toyota", "Fortuner")

print(car1.brand, car1.model)
print(car2.brand, car2.model)
```

Dry Run:

1. Class Car defined with brand and model.
2. car1 created → brand=Honda, model=City.
3. car2 created → brand=Toyota, model=Fortuner.
4. Output →

```
Honda City
```

¢2è2ű I· C2ǋűdzƴŜǋ

9.2 __init__ and self

Definition:

- ψψλƴλűψψ λă l· ăLJŜŎλl·ƭ ϓŜűK2Ř óŎ2ƴăűǋdzŎű2ǋú l·dzű2ϓ l·űλŎl·ƭƭè Ŏl·ƭƭŜŘ øKŜƴ l·ƴ 2ōƨŜŎű λă ŎǋŜl·űŜŘφ
- ăŜƭŦ ǋŜLJǋŜăŜƴűă űKŜ 2ōƨŜŎű λűăŜƭƒ Ŝƴădzǋλƴƌ Ŝl·ŎK 2ōƨŜŎű ƪŜŜLJă λűă 2øƴ Řl·űl·φ

Example:

Ŏƭl·ăă {űdzŘŜƴűϓ

ŘŜƒ ψψλƴλűψψóăŜƭƒΣ ƴl·ϓŜΣ l·ƌŜúϓ

ăŜƭƒφƴl·ϓŜ Ґ ƴl·ϓŜ

ăŜƭƒφl·ƌŜ Ґ l·ƌŜ

ăм Ґ {űdzŘŜƴűóϦwl·KdzƭϦΣ нлú

ăн Ґ {űdzŘŜƴűóϦtǋλèl·ϦΣ ннú

LJǋλƴűóăмφƴl·ϓŜΣ ămφl·ƌŜú

LJǋλƴűóăнφƴl·ϓŜΣ ăнφl·ƌŜú

Dry Run:

- ăм Ћ ƴl·ϓŜҐwl·KdzƭΣ l·ƌŜҐнлφ
- ăн Ћ ƴl·ϓŜҐtǋλèl·Σ l·ƌŜҐннφ
- hdzűLJdzűϓ

wl·Kdzƭ нл

tǋλèl· нн

Analogy: ăŜƭƒ ø2ǋƪă ƭλƪŜ ϦűKλăϦ λƴ 9ƴƌƭλăK ăŜƴűŜƴŎŜă Ћ *"This student's name is Rahul."*

9.3 Inheritance

Definition:
LƴKŜǋλűl·ƴŎŜ l·ƭƭ2øă 2ƴŜ Ŏƭl·ăă óŎKλƭŘú ű2 dzăŜ űKŜ LJǋ2LJŜǋűλŜă l·ƴŘ ϓŜűK2Řă 2ƒ l·ƴ2űKŜǋ Ŏƭl·ăă óLJl·ǋŜƴűúφ

Example:

```
class Animal:
    def eat(self):
        print("Animal eats")


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


d = Dog()
d.eat()
d.bark()
```

Dry Run:

- Dog inherits eat() from Animal.
- d.eat() → “Animal eats”.
- d.bark() → “Dog barks”.

Analogy: Children inherit traits (like eye color) from parents.

9.4 Method Overriding

Definition:
If a child class defines a method with the **same name** as the parent class, the child’s version is used.

Example:

```
class Animal:
    def sound(self):
        print("Animal makes sound")


class Cat(Animal):
    def sound(self):
        print("Cat meows")


c = Cat()
```

c.sound()

Dry Run:

- Cat overrides sound().
- Output → "Cat meows".

Analogy: Parent says "I'll drive," child says "No, I'll drive my way."

9.5 Encapsulation

Definition:
Encapsulation is **data hiding**. Attributes can be made private using __. Access is given only through methods.

Example:

```
class Bank:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

account = Bank(5000)
print(account.get_balance())
```

Dry Run:

- __balance is private.
- Direct access fails → account.__balance gives error.
- get_balance() → returns 5000.

Analogy: Like an ATM – you cannot open the bank's vault but can withdraw money through the correct channel.

9.6 Polymorphism

Definition:
Polymorphism means "many forms." A single function/method name behaves differently depending on the object.

Example:

```
class Bird:
```

```
    def fly(self):
        print("Bird flies in the sky")


class Airplane:
    def fly(self):
        print("Airplane flies in the air")


for obj in (Bird(), Airplane()):
    obj.fly()
```

Dry Run:

- Loop 1 → Bird.fly() → “Bird flies in the sky”.
- Loop 2 → Airplane.fly() → “Airplane flies in the air”.

Analogy: “Fly” has different meanings for a sparrow and an airplane.

9.7 Overloading (Dunder Methods)

Definition:
Operator overloading allows using operators (+, -, *) with objects by defining special dunder methods like __add__, __sub__.

Example:

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y


    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)


v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2
print(v3.x, v3.y)
```

Dry Run:

- v1 + v2 calls __add__.
- New Vector(6, 8) created.
- Output → 6 8.

Analogy: Just like + works for both numbers and strings differently.

9.8 Abstraction

Definition:
Abstraction hides internal details and shows only the necessary features. In Python, it is achieved using abc module.

Example:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):

    @abstractmethod

    def start(self): pass

class Car(Vehicle):

    def start(self):

        print("Car starts with key")

c = Car()

c.start()
```

Dry Run:

- Abstract class Vehicle → cannot be used directly.
- Car provides its own start().
- Output → “Car starts with key”.

Analogy: You use a phone to call but don’t need to know the microchip’s working

9.9 High-Level Example: E-Commerce System

```
from abc import ABC, abstractmethod

class User(ABC):

    def __init__(self, name):

        self.name = name

    @abstractmethod

    def access(self): pass
```

```
class Customer(User):
    def access(self):
        print(f"{self.name} can browse and buy products")

class Admin(User):
    def access(self):
        print(f"{self.name} can add or remove products")

u1 = Customer("Rahul")
u2 = Admin("Priya")

u1.access()
u2.access()
```

Dry Run:

- Rahul → Customer → access = “browse and buy”.
- Priya → Admin → access = “add or remove”.

Concepts Used: Classes, __init__, self, Inheritance, Overriding, Polymorphism, Abstraction.

9.10 Chapter Summary

- **Class** = blueprint, **Object** = instance.
- **__init__ & self** initialize attributes.
- **Inheritance** = reuse parent features.
- **Overriding** = child redefines parent method.
- **Encapsulation** = hide sensitive data.
- **Polymorphism** = same method, different behavior.
- **Overloading** = redefine operators using dunder methods.
- **Abstraction** = hide implementation, show essentials.

Chapter 10: File Handling and CSV Operations in Python

CλfŜ Kl·yŘfλyƎ λă l·y˘ŜăăŜyűλl·f LJl·Nű 2Ŧ LJN2ƎNl·Y˘Yλy∃¢ Lű l·ff2ǿă LJN2ƎNl·Y˘ă ű2 **store, retrieve, and manipulate data** λy˘ŦλfŜăΣ Y˘l·ʅλy∃ űKŜY Řèyl·Y˘λŎ l·yŘ Y˘2NŜ dzăŜŦdzf¢ tèűK2y˘LJN2ØλŘŜă ăλY˘LJfŜ l·yŘ λyűdzλűλØŜ ǿl·èă ű2 ǿ2Nʅ ǿλűK ŦλfŜăΣ ō2űK űŜĚű l·yŘ /{±¢

10.1 Opening and Closing Files

Definition:
¢KŜ 2LJŜyǿú Ŧdzyŏűλ2y˘λă dzăŜŘ ű2 **open a file** l·yŘ NŜűdzNyã l· **file object** ű2 λyűŜNl·Ŏű ǿλűK űKŜ ŦλfŜ¢ !ŦűŜN ŦλyλăKλyƎΣ ǿŜ Y˘dzăű **close** űKŜ ŦλfŜ ű2 ăl·ØŜ ŎKl·yƎŜă l·yŘ ŦNŜŜ ăèăúŜY˘ NŜă2dzNŎŜă¢

Syntax:

ŦλfŜψ2ō²ŜŎű Γ 2LJŜyǿhŦλfŜyl·Y˘Ŝh∑ hY˘2ŘŜh ú

- hŦλfŜyl·Y˘Ŝh Γh bl·Y˘Ŝ 2Ŧ űKŜ ŦλfŜ óŜ¢Ǝ¢∑ hŘl·ű¡·¢ÚĚűhú¢
- hY˘2ŘŜh Γh a2ŘŜ 2Ŧ 2LJŜNl·ûλ2y˘óN∑ ǿ∑ l·∑ NЪ∑ ŜűŎ¢ú¢

Example:

Ŧ Γ 2LJŜyǿhŘl·ű¡·¢ÚĚűh∑ hNhú l 2LJŜy˘λy˘NŜl·Ř Y˘2ŘŜ

Ŏ2yűŜyű Γ Ŧ¢NŜl·Řóú

LJNλyűóŎ2yűŜyűú

Ŧ¢Ŏf2ăŜóú

Dry Run:

м¢ CλfŜ Řl·ű¡·¢ÚĚű 2LJŜySŘ λy˘hNh Y˘2ŘŜ¢

н¢ NŜl·Řóú NŜl·Řă ŜyűλNŜ Ŏ2yűŜyű¢

о¢ LJNλyűóú ŘλăLJfl·èă Ŏ2yűŜyű¢

п¢ CλfŜ Ŏf2ăŜŘ dzăλyƎ Ŧ¢Ŏf2ăŜóú¢

Analogy: hLJŜyλy∃ l· ō2ƨʅ ű2 **read or write notes**∑ űKŜy˘Ŏf2ăλy∃ λű l·ŦűŜN dzăŜ¢

10.2 File Modes

tèűK2y˘Kl·ă ŘλŦŦŜNŜyű Y˘2ŘŜă ű2 λyűŜNl·Ŏű ǿλűK ŦλfŜăY

Mode	Meaning
"r"	Read-only; file must exist
"w"	Write; overwrites if file exists, creates if not
"a"	Append; adds data at the end
"r+"	Read and write; file must exist

Examples:

Read Mode:

```
f = open("data.txt", "r")
print(f.read())
f.close()
```

Write Mode:

```
f = open("data.txt", "w")
f.write("Hello Python!")
f.close()
```

Append Mode:

```
f = open("data.txt", "a")
f.write("\nWelcome to Python")
f.close()
```

Dry Run:

- "r" → reads existing file.
- "w" → overwrites content.
- "a" → adds new content at the end.

10.3 File Methods

1) read()

Reads **entire file** as a string.

2) readline()

Reads **one line at a time**.

3) readlines()

wŜűdzNyã l·ff fλySă l·ă l· **list of strings**ф

4) write()

² NλűŜă ăűNλy∃ Řl·űl· ű2 l· ŦλfŜф

Example:

ǿλűK 2LJŜyŏþŘl·űl·ф̈ÊűþΣ þǿþú l·ă ŦY

Ŧ₹ǿNλűŜóþ[λySмξy[λyŜнþú

ǿλűK 2LJŜyŏþŘl·űl·ф̈ÊűþΣ þNþú l·ă ŦY

LJNλyűó₹₹NŜl·Řóúú

Dry Run:

- CλfŜ ǿNλűűŜy˙Γþ þ[λySмξy[λyŜнþф
- hdzűLJdzű Γþ

[λySм

[λySн

10.4 Using with open (Recommended)

Definition:
ǿλűK 2LJŜyŏú l·dzű2Y˙l·űλŎl·ffè Ŏf2ăŜă űKŜ ŦλfŜ l·ŦűŜN űKŜ ōf2Ŏ]Σ ŜØŜy˙λŦ l·y˙ŜNN2N 2ŎŎdzNăф

Example:

ǿλűK 2LJŜyŏþŘl·űl·ф̈ÊűþΣ þNþú l·ă ŦY

Ŏ2yűŜyű Γ ₹₹NŜl·Řóú

LJNλyűóŎ2yűŜyűú

Analogy: [λ]Ŝ dzăλy∃ l· **locker that locks automatically** ǿKŜy˙è2dz ǿl·f] l·ǿl·èф

10.5 Working with CSV Files

Definition:
/{± ó/2Y˙Y˙l·π{ŜLJl·Nl·űŜŘ ±l·fdzŜăú ŦλfŜă ăű2NŜ **table-like data**ф 9l·ŎK N2ǿ NŜLJNŜăŜyűă l· NŜŎ2NŘΣ l·yŘ Ŏ2Y˙Y˙l·ă ăŜLJl·Nl·űŜ Ŏ2fdzY˙yãф tèűK2yΩă ŎăØ Y˙2Řdzfŝ ăλY˙LJfλŦλŜă NŜl·Řλy∃ l·yŘ ǿNλűλy∃ /{±ăф

10.5.1 Reading CSV Files

Example: ăűdzŘŜyűăфŎăØ

bl·Y·Ŝ! ∃ŜDNl·ŘŜ

wl·Kdzf£нл!

tNλèl·£нн.

λY·LJ2Nű Ŏă∅

ø λűK 2LJŜyŏþăűdzŘŜyűă¢ŎăØþ ÞNþú l·ă ₹Y

NŜl·ŘŜN Γ Ŏă∅¢NŜl·ŘŜNó₹ú

₹2N N2ø λy·NŜl·ŘŜNY

LJNλyűó¢N2øú

Dry Run:

- wŜl·Řă N2ø ōè N2ø Γþ

ω4bl·Y·Ŝ4£ 4!∃Ŝ4£ 4DNl·ŘŜ4ß

ω4wl·Kdzf4£ 4нл4£ 4!4ß

ω4tNλèl·4£ 4нн4£ 4.4ß

Analogy: wŜl·Řλy∃ l· **table from Excel row by row¢**

10.5.2 Writing CSV Files

λY·LJ2Nű Ŏă∅

ø λűK 2LJŜyŏþăűdzŘŜyűă¢ŎăØþ Þøþ£ ySøf‡ySΓþþú l·ă ₹Y

øNλűŜN Γ Ŏă∅¢øNλűŜNó₹ú

øNλűŜN¢øNλűŜN2øówbl·Y·Ŝþ£ þ!∃Ŝþ£ þDNl·ŘŜþßú

øNλűŜN¢øNλűŜN2øówwl·Kdzfþ£ нл£ þ!þßú

øNλűŜN¢øNλűŜN2øówtNλèl·þ£ нн£ þ.þßú

Dry Run:

- /NŜl·űŜă /{± ₹λfŜ øλűK KŜl·ŘŜN ь н Řl·űl· N2øă¢

10.5.3 Reading CSV as Dictionary

ø λűK 2LJŜyŏþăűdzŘŜyűă¢ŎăØþ Þøþ£ l·ă ₹Y

```
NSI·RSN Γ ÖäØ⁰5λÖűwSI·RSNó₮ú

₮2NJ N2ø λYˇNSI·RSNY

    LJNJλyűóN2øώbI·YˇSh6Σ N2øώbDNJI·RSh6ú
```

Output:

```
wl·Kdzf !

tNJλèl· .
```

Advantage: /ƨfdzYˇyã Ölˇ·yˇōS l·ÖÖŜäâŜŘ **by name**Σ Yˇl·ʅλy∃ Ö2ŘŜ NSI·Řl·ōfŜΦ

10.5.4 Writing CSV as Dictionary

```
øλűK ƨLJSyˇōhăűdzŘSyˇűăΦÖäØhΣ høhΣ ySøfλySΓhhú l·ă ₮Y

    ₮ΛŜfŘyl·YˇSă Γ ώbI·YˇShΣh ! ∃ShΣhDNJI·RSh6

    øNJλűŜNJ Γ ÖäØ⁰5λÖű ² NJλűŜNó₮Σ ₮ΛŜfŘyl·YˇSăΓ₮ΛŜfŘyl·YˇSăú

    øNJλűŜN⁰øNJλűŜKŜI·RSNóú

    øNJλűŜN⁰øNJλűŜN2øóYhbI·YˇShYhwl·Kdzfh∑h ! ∃ShYнлΣhDNJI·RŜhYh ! hYú

    øNJλűŜN⁰øNJλűŜN2øóYhbI·YˇShYh tNJλèl·h∑h ! ∃ShYннΣhDNJI·RŜhYh . hYú
```

Dry Run:

- {l·YˇS /{± ÖNJSI·űŜŘ ōdzű Yˇ2NJS NJSI·Řl·ōfŜ l·yŘ ₮fŜÈλōfŜ dzăλy∃ ŘλÖűλ2yl·NJλŜăΦ

10.6 High-Level Example: Sales Report

```
λYˇLJ2Nű ÖäØ


I /NJSI·űŜ /{± ₮λfŜ

øλűK ƨLJSyˇōhăl·fŜăΦÖäØhΣ høhΣ ySøfλySΓhhú l·ă ₮Y

    øNJλűŜNJ Γ ÖäØ⁰øNJλűŜNó₮ú

    øNJλűŜN⁰øNJλűŜN2øóώh tN2ŘdzÖűhΣh tNJλÖŜhΣh vdzl·yűλűèh6ú

    øNJλűŜN⁰øNJλűŜN2øóώh[l·LJű2LJhΣpллллΣo6ú

    øNJλűŜN⁰øNJλűŜN2øóώh a 2dzăŜhΣpллΣмл6ú

    øNJλűŜN⁰øNJλűŜN2øóώhYŜèō2l·NJŘhΣмpллΣp6ú
```

I wŜI·Ř /{± I·yŘ ŎI·fÔdzfI·ű$ ű2ű$I·f ăI·fŜă

ǿλűK 2LJŜyŏђăI·fŜăΦŎăØђΣ ђNЉú I·ă ŦY

NJŜI·ŘŜN Ґ ŎăØΦ5λŎűwŜI·ŘŜNJóŦú

ű2ű$I·f Ґ л

Ŧ2N NJ2ǿ λyΥNJŜI·ŘŜNJY

ű2ű$I·f ЬҐ λyΥűóN2ǿώђtNJλÔŜђßú ₣ λyΥűóN2ǿώђvdzI·yΥűλűèђßú

LJNJλyΥűóђ¢2ű$I·f {I·fŜăYђΣ ű2ű$I·fú

Dry Run:

- [I·LJű2LJ Гђ рллллғоҐмрллллл
- a 2dzăŜ Гђ роллғмлҐрллл
- YŜѐō2I·NJŘ Гђ мрллғрҐтрлл
- ¢2ű$I·f Ґ мснрлл

Output:

¢2ű$I·f {I·fŜăY мснрлл

Concepts Used: CλfŜ ǿNJλűλyƎΣ NJŜI·ŘλyƎΣ /{±Σ 5λŎűwŜI·ŘŜNΣ f22LJăΣ I·NJλűKYˇŜűλÔ 2LJŜNJI·ű$λ2yΥãΦ

✓ Chapter 10 Summary

- **File Modes:** ђNђ Гђ NJŜI·ŘΣ ђǿђ Гђ ǿNJλűŜΣ ђI·ђ Гђ I·LJLJŜyŘΣ ђNЬђ Гђ NJŜI·Ř ɜ ǿNJλűŜΦ
- **Methods:** NJŜI·ŘóúΣ NJŜI·ŘfλyŜóúΣ NJŜI·ŘfλyŜăóúΣ ǿNJλűŜóúΦ
- **with open:** {I·ŦŜNΣ I·dzű2πŎf2ăŜă ŦλfŜăΦ
- **CSV Module:** ŎăØΦNJŜI·ŘŜNΣ ŎăØΦǿNJλűŜNΣ ŎăØΦ5λŎűwŜI·ŘŜNΣ ŎăØΦ5λŎű ² NJλűŜNΦ
- **High-Level Application:** /NJŜI·űŜ NJŜLJ2Nűă$ ăű2NJŜ ŘI·űI·Σ ÔI·fÕdzfI·ûŜ ű2ű$I·fãΦ

Chapter 11: Regular Expressions (Regex) in Python

Regular Expressions, or **Regex**, are **patterns that describe sets of strings**. They allow programmers to **search, match, extract, and manipulate text** efficiently. Python provides the **re module** to work with regex.

Why Regex is important:

- Validate input data like emails, phone numbers, and dates.
- Extract meaningful information from large text.
- Replace or clean unwanted text.
- Log analysis, web scraping, and data preprocessing.

Analogy: Regex is like a **high-powered microscope for text** — it can zoom into patterns that humans would take hours to find.

11.1 Python re Module

Importing Regex:

```
import re
```

Core Functions:

Function	Purpose
search()	Returns first match anywhere in the string
match()	Checks match only at the **start** of string
findall()	Returns **all matches** as a list
sub()	Replaces matched patterns with new string
split()	Splits string based on pattern

Syntax:

```
result = re.function(pattern, string)
```

- pattern → Regex pattern
- string → Text to search

11.2 Basic Regex Patterns

Pattern	Meaning	Example
\d	Any digit [0-9]	\d{2} → 23
\D	Non-digit	\D+ → "abc"
\w	Alphanumeric + underscore [a-zA-Z0-9_]	\w+ → "Python3"
\W	Non-alphanumeric	\W → "!"
.	Any character except newline	. → "a" matches
+	One or more repetitions	\d+ → 123
*	Zero or more repetitions	\w* → "abc"
{n}	Exactly n repetitions	\d{4} → 2025
{n,m}	Between n and m repetitions	\d{2,4} → 25 or 2025
[]	Any character inside	[abc] → a or b or c
^	Start of string	^Hello
$	End of string	end$

11.3 search() Function

Definition: Searches **anywhere** in the string and returns a **Match object**.

Example:

```
text = "My phone number is 9876543210"
result = re.search(r"\d{10}", text)
print(result.group())
```

Dry Run:

- Pattern \d{10} → 10 consecutive digits
- Finds "9876543210"
- Output → 9876543210

Note: Returns only **first match**.

11.4 match() Function

Definition: Checks if **pattern matches at the start** of string.

```
text = "9876543210 is my phone number"
result = re.match(r"\d{10}", text)
print(result.group())
```

Dry Run:

- Pattern matches beginning → 9876543210
- If string starts differently, match() returns None.

11.5 findall() Function

Definition: Returns **all matches** as a list.

```
text = "Call 12345 or 67890"
numbers = re.findall(r"\d+", text)
print(numbers)
```

Dry Run:

- \d+ → matches one or more digits
- Finds [12345, 67890]

11.6 sub() Function

Definition: Replaces **all occurrences of a pattern** with a new string.

```
text = "My number is 9876543210"
new_text = re.sub(r"\d", "*", text)
print(new_text)
```

Dry Run:

- All digits replaced by *
- Output → "My number is **********"

High-Level Usage: Mask sensitive info like phone numbers or passwords.

11.7 Using Character Sets and Quantifiers

- [abc] → Matches a, b, or c
- [a-z] → Matches lowercase letters
- [A-Z] → Matches uppercase letters
- [0-9] → Matches digits
- + → One or more repetitions
- * → Zero or more repetitions
- {n} → Exactly n repetitions
- {n,m} → Between n and m repetitions

Example:

```
text = "My code is ABC123"
print(re.findall(r"[A-Z]+", text))  # ['ABC']
print(re.findall(r"\d+", text))     # ['123']
```

11.8 Practical Validation Examples

11.8.1 Email Validation

```
text = "Contact me at example@gmail.com"
pattern = r"[a-zA-Z0-9._]+@[a-zA-Z]+\.[a-zA-Z]{2,}"
result = re.search(pattern, text)
print(result.group())
```

Dry Run:

- Username → [a-zA-Z0-9._]+
- Domain → [a-zA-Z]+
- Extension → \.[a-zA-Z]{2,}
- Output → example@gmail.com

11.8.2 Phone Number Validation

```
text = "Call 9876543210"
pattern = r"\b\d{10}\b"
```

print(re.search(pattern, text).group())

Dry Run:

- 10-digit number matched
- Output → 9876543210

11.8.3 Date Validation (DD/MM/YYYY)

text = "Event on 15/09/2025"

pattern = r"\b\d{2}/\d{2}/\d{4}\b"

print(re.search(pattern, text).group())

- Output → 15/09/2025

11.9 High-Level Example: Extract Emails and Phones from Text

```
import re

text = """
John, john@example.com, 9876543210
Priya, priya@example.com, 9123456789
"""

emails = re.findall(r"[a-zA-Z0-9._]+@[a-zA-Z]+\.[a-zA-Z]{2,}", text)
phones = re.findall(r"\b\d{10}\b", text)

print("Emails:", emails)
print("Phone Numbers:", phones)
```

Dry Run:

- Emails → ['john@example.com', 'priya@example.com']
- Phone Numbers → ['9876543210', '9123456789']

11.10 Tips for Beginners

1. Always use **raw strings** (r"pattern") to avoid escape character issues.
2. Test patterns with **simple examples first**.
3. Use **online regex testers** to debug complex patterns.
4. Combine functions: search, findall, sub for advanced tasks.
5. Understand **quantifiers** (+, *, {}) and **character sets** ([]) first.

✅ Chapter 11 Summary

- Regex is used for **pattern matching and text processing**.
- re module functions: search(), match(), findall(), sub(), split().
- **Patterns:** \d, \w, +, *, {n}, {n,m}, [], ^, $.
- **Practical Uses:** Validate emails, phone numbers, dates, extract information.
- High-level applications: Parsing text, cleaning data, automating reports.

Chapter 12: Modules, Packages, and Libraries in Python

🔷 12.1 Introduction

When we write small Python programs, we can manage everything in one file. But as programs grow larger, **code organization** becomes critical.

- Instead of writing **everything in one file**, we break our code into **smaller reusable files (modules)**.
- We can then group multiple modules into **packages**.
- A collection of packages is known as a **library**.

👉 This structure makes code **reusable, manageable, and professional**.

🔷 12.2 What is a Module?

- **Definition:** A module is a **Python file (.py)** that contains variables, functions, or classes.
- **Analogy:** A module is like a **recipe card**. Each card tells you how to cook one dish. Instead of writing all recipes in one giant notebook, you keep separate cards.

✅ Example 1: Creating and Using a Module

Create a file named math_utils.py:

```
# math_utils.py

def add(a, b):
    return a + b


def subtract(a, b):
    return a - b
```

Now create another file main.py:

```
import math_utils
```

```
print(math_utils.add(10, 5))      # 15
print(math_utils.subtract(20, 8)) # 12
```

Dry Run:

1. Python looks for math_utils.py.
2. Imports its functions into memory.
3. Calls math_utils.add(10, 5) → returns 15.
4. Calls math_utils.subtract(20, 8) → returns 12.

✅ **Benefit:** Reusability – once defined, we can use it in multiple programs.

🔷 12.3 Built-in Modules

Python ships with **hundreds of built-in modules**. Some popular ones:

Module	Purpose	Example Usage
math	Mathematical functions	math.sqrt(16)
random	Random number generation	random.randint(1, 10)
datetime	Working with dates/times	datetime.date.today()
os	File system operations	os.listdir()
sys	System-specific functions	sys.version

✅ Example 2: Using Built-in Modules

```
import math, random, datetime

print(math.factorial(5))      # 120
print(random.choice([1,2,3])) # Random element
print(datetime.date.today())  # Prints today's date
```

Dry Run:

- math.factorial(5) → multiplies numbers 5×4×3×2×1 = 120
- random.choice randomly picks a list element.

- date.today() fetches today's date.

✅ Saves time since you don't need to reinvent the wheel.

🔷 12.4 Importing Modules

There are **3 ways** to import:

```
import math
print(math.sqrt(25))    # Method 1

from math import sqrt
print(sqrt(25))        # Method 2

import math as m
print(m.sqrt(25))      # Method 3 (alias)
```

👉 import brings the whole module,
👉 from ... import ... brings specific functions,
👉 as allows renaming.

🔷 12.5 What is a Package?

- **Definition:** A package is a **collection of modules**, stored in a directory with a special file __init__.py.
- **Analogy:** If a **module is a recipe card**, a **package is a recipe book** (collection of cards).

✅ Example 3: Creating a Package

Folder structure:

```
my_package/
   __init__.py
   math_utils.py
   string_utils.py
```

math_utils.py

```
def square(n):
```

```
    return n * n
```

string_utils.py

```
def reverse(s):
    return s[::-1]
```

main.py

```
from my_package import math_utils, string_utils


print(math_utils.square(4))   # 16
print(string_utils.reverse("Python")) # nohtyP
```

Dry Run:

- Python sees my_package/ with __init__.py → recognizes as a package.
- Imports required modules inside it.

✅ Packages help in **better organization** for big projects.

🔷 12.6 What is a Library?

- **Definition:** A library is a **collection of packages and modules** designed to solve specific problems.
- **Examples:**
    - **NumPy** → Numerical computing.
    - **Pandas** → Data analysis.
    - **Matplotlib** → Graphs and charts.
    - **TensorFlow** → Machine Learning.

Analogy: A **library is like a whole warehouse** containing many books (packages) and pages (modules).

🔷 12.7 Creating Your Own Library

Let's create a **utility library**.

```
utility_lib/
    __init__.py
    calc.py
```

text_ops.py

calc.py

```
def cube(n): return n**3
```

text_ops.py

```
def is_palindrome(s): return s == s[::-1]
```

main.py

```
from utility_lib import calc, text_ops

print(calc.cube(3))          # 27
print(text_ops.is_palindrome("madam"))  # True
```

✅ You just built your **own Python library**!

🔷 12.8 High-Level Example: Data Cleaning Library

Create a package data_cleaner/.

text_cleaner.py

```
import re

def clean_text(s):
    return re.sub(r'[^a-zA-Z0-9 ]', '', s).lower()
```

num_cleaner.py

```
def remove_negatives(lst):
    return [x for x in lst if x >= 0]
```

main.py

```
from data_cleaner import text_cleaner, num_cleaner

print(text_cleaner.clean_text("Hello@2025!!"))  # hello2025
print(num_cleaner.remove_negatives([-3, 4, -1, 6])) # [4, 6]
```

✅ This is how real-world data preprocessing libraries work.

12.9 Key Takeaways

- **Module** → A .py file with reusable functions/classes.
- **Package** → A folder containing multiple modules.
- **Library** → A collection of packages (like NumPy, Pandas).
- import, from ... import ..., and alias are ways to bring modules into a program.
- You can build your **own libraries** to make projects reusable and professional.

✅ Chapter 12 Summary

- You learned **Modules, Packages, Libraries**.
- You practiced **built-in and custom modules**.
- You created and imported a **package**.
- You built a **real-world data cleaning library**.
- You understood the difference between **module vs package vs library**.

Chapter 13: Introduction to NumPy

🔷 13.1 Introduction

What is NumPy?

NumPy (**Numerical Python**) is the **foundation of data science and scientific computing in Python**. It provides:

- A **fast, compact, N-dimensional array object (ndarray)**.
- **Mathematical functions** for linear algebra, random numbers, statistics.
- Support for **vectorization** (apply operations on entire arrays without loops).

👉 **Dry Run Example:**

```
import numpy as np

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

print(arr * 2)   # [2 4 6 8]
```

🔎 Python list version would need a loop:

```
nums = [1,2,3,4]

print([x*2 for x in nums])   # slower
```

So, NumPy is **faster and easier**.

Importance of NumPy (vs Python lists)

- **Speed**: NumPy is written in **C** under the hood.
- **Memory Efficiency**: NumPy stores data in **contiguous blocks**, unlike Python lists that store object references.
- **Convenience**: Easy to reshape, slice, and perform **matrix operations**.
- **Foundation**: Libraries like **pandas, scikit-learn, TensorFlow, PyTorch** all use NumPy internally.

Installation

```
pip install numpy
```

Importing

```
import numpy as np
```

Version

```
print(np.__version__)
```

13.2 NumPy Arrays

NumPy's **ndarray** is like a super-powered list.

Creating Arrays

np.array([1,2,3]) # from list

np.zeros((2,3)) # 2x3 zeros

np.ones((3,3)) # 3x3 ones

np.empty((2,2)) # uninitialized

np.full((2,2), 7) # filled with 7

np.arange(0,10,2) # like range()

np.linspace(0,1,5) # divide into 5 equal parts

np.logspace(1,3,3) # log scale [10,100,1000]

np.eye(3) # identity matrix

Array Dimensions

- **1D**: [1 2 3]
- **2D**: [[1 2 3],[4 5 6]]
- **3D**: "array of arrays"

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

print(arr.ndim) # 3

Array Attributes

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

print(arr.ndim) # 2 → dimensions
print(arr.shape) # (2,3) → rows x
cols print(arr.size) # 6 → total
elements

print(arr.dtype) # int32

print(arr.itemsize) # size of each element

print(arr.nbytes) # total bytes used

Changing dtype

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

print(arr)

print(arr.astype(int))

13.3 Indexing and Slicing

```
arr = np.array([10,20,30,40,50])


print(arr[0])       # 10
print(arr[-1])      # 50
print(arr[1:4])     # [20 30 40]
```

2D Indexing

```
matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(matrix[1,2])      # 6
print(matrix[0:2,1:3])  # [[2 3] [5 6]]
```

Fancy Indexing

```
arr = np.array([10,20,30,40,50])
print(arr[[0,2,4]])   # [10 30 50]
```

Boolean Indexing

```
arr = np.array([1,2,3,4,5,6])
print(arr[arr > 3])   # [4 5 6]
```

13.4 Array Operations

Element-wise

```
arr = np.array([1,2,3])
print(arr + 5)   # [6 7 8]
print(arr * 2)   # [2 4 6]
print(arr ** 2)  # [1 4 9]
```

Universal Functions (ufuncs)

```
arr = np.array([1,4,9])
print(np.sqrt(arr))   # [1. 2. 3.]
print(np.power(arr,2)) # [1 16 81]
```

Comparison

```
arr = np.array([1,2,3,4])
print(arr > 2)   # [False False True True]
```

13.5 Mathematical Functions

```
arr = np.array([0, np.pi/2, np.pi])


print(np.sin(arr))   # [0. 1. 0.]
print(np.exp([1,2])) # [2.718 7.389]
print(np.log([1, np.e])) # [0. 1.]
print(np.round([1.2,1.7])) # [1. 2.]
```

13.6 Aggregate Functions

```
arr = np.array([10,20,30,40])
print(arr.min())    # 10
print(arr.max())    # 40
print(arr.sum())    # 100
print(arr.mean())   # 25.0
print(arr.std())    # standard deviation
print(np.cumsum(arr)) # [10 30 60 100]
```

13.7 Shape Manipulation

```
arr = np.arange(1,13)
print(arr.reshape(3,4))   # reshape
print(arr.ravel())        # flatten
print(arr.T.reshape(4,3)) # transpose
```

13.8 Joining and Splitting

```
a = np.array([1,2,3])
b = np.array([4,5,6])

print(np.hstack((a,b)))   # [1 2 3 4 5 6]
print(np.vstack((a,b)))   # [[1 2 3] [4 5 6]]
Splitting:
arr = np.array([1,2,3,4,5,6])
print(np.split(arr,3))   # [ [1 2], [3 4], [5 6] ]
```

13.9 Copying and Sorting

```
arr = np.array([3,1,2])
print(np.sort(arr))    # [1 2 3]
print(np.argsort(arr)) # [1 2 0]
print(np.unique([1,2,2,3])) # [1 2 3]
```

13.10 Random Module

```
print(np.random.rand(3))      # random floats [0-1)
print(np.random.randint(1,10,5)) # 5 random ints
print(np.random.choice([1,2,3], size=5))
np.random.seed(42)            # reproducible
```

13.11 Linear Algebra

```
A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

print(np.dot(A,B))
print(np.linalg.det(A))   # determinant
print(np.linalg.inv(A))   # inverse
vals, vecs = np.linalg.eig(A)
print(vals)  # eigenvalues
```

13.12 Statistics

```
arr = np.array([10,20,30,40,50])

print(np.mean(arr))   # 30
print(np.median(arr)) # 30
print(np.var(arr))    # 200
print(np.std(arr))    # 14.14
print(np.percentile(arr,50)) # 30
```

🔷 13.13 File Handling

```
arr = np.array([1,2,3,4,5])
np.save('mydata.npy', arr)
print(np.load('mydata.npy'))

np.savetxt('data.txt', arr)
print(np.loadtxt('data.txt'))
```

🔷 13.14 Advanced Indexing

```
arr = np.arange(10)
mask = arr % 2 == 0
print(arr[mask])   # even numbers
```

🔷 13.15 Memory and Performance

- **Vectorization** → Avoids loops
- **Views vs Copies**

```
arr = np.array([1,2,3])
view = arr.view()
copy = arr.copy()
```

🔷 13.16 Advanced Array Ops

```
arr = np.array([1,2])
print(np.tile(arr, 3))     # [1 2 1 2 1 2]
print(np.repeat(arr, 3))   # [1 1 1 2 2 2]

x, y = np.meshgrid([1,2,3],[4,5])
print(x)
print(y)
```

◈ High-Level Examples

Example 1: Normalize Data

```
arr = np.array([10,20,30,40,50])
normalized = (arr - arr.min()) / (arr.max() - arr.min())
print(normalized)
```

Example 2: Image Simulation

```
import matplotlib.pyplot as plt
image = np.random.randint(0,255,(10,10))
plt.imshow(image, cmap="gray")
plt.show()
```

Example 3: Solve Equations

```
# Solve 2x + y = 5, x - y = 1
A = np.array([[2,1],[1,-1]])
b = np.array([5,1])
solution = np.linalg.solve(A,b)
print(solution)  # [2. 1.]
```

Chapter 14: Pandas in Python

---

1. Introduction to Pandas

What is Pandas?

Pandas is an **open-source Python library** used for **data manipulation and data analysis**.

- It is built on top of **NumPy**, which provides fast mathematical operations.
- Think of it as a combination of **Excel + SQL + Python**.
- It provides easy-to-use structures like:
    - **Series** → One-dimensional labeled data (like a column in Excel).
    - **DataFrame** → Two-dimensional labeled data (like an Excel table or SQL table).

☞ In short, Pandas makes working with structured data **fast, easy, and powerful**.

Why Pandas?

- **Speed** → Handles millions of rows faster than pure Python lists.
- **Readability** → Excel/SQL-like operations with simple syntax.
- **Flexibility** → Works with CSV, Excel, SQL, JSON, and more.
- **Integration** → Works seamlessly with NumPy, Matplotlib, Scikit-learn.

Installation

pip install pandas

Importing Pandas

```
import pandas as pd

print(pd.__version__)
```

---

2. Pandas Data Structures

(a) Series – One-dimensional labeled array

Definition:
A Pandas **Series** is like a single column in Excel. It has:

- **Index** (row labels)

- **Values** (data stored)

Example:

```
import pandas as pd


s = pd.Series([10, 20, 30, 40], index=['a','b','c','d'])
print(s)
```

Output:

```
a  10
b  20
c  30
d  40
dtype: int64
```

🔷 **Dry Run:**

- Pandas created a labeled array with index a,b,c,d.
- Values are integers.
- Now, s['a'] → 10 like dictionary-style lookup.

(b) DataFrame – Two-dimensional labeled data

Definition:
A Pandas **DataFrame** is like a table in Excel/SQL → Rows & Columns.

Example:

```
data = {
   "Name": ["Amit", "Riya", "Kunal"],
   "Age": [25, 30, 22],
   "City": ["Delhi", "Mumbai", "Pune"]
}
df = pd.DataFrame(data)
print(df)
```

Output:

```
   Name  Age    City
0   Amit   25   Delhi
1   Riya   30  Mumbai
2  Kunal   22    Pune
```

🔷 **Dry Run:**

- Keys of dictionary became **columns**.
- Values of dictionary became **rows**.
- Pandas automatically assigned row indexes 0,1,2.

---

3. Creating DataFrames

Pandas allows multiple ways:

1. **From List of Lists**

```
data = [[1, "Amit"], [2, "Riya"], [3, "Kunal"]]
df = pd.DataFrame(data, columns=["ID", "Name"])
```

2. **From Dictionary**

```
df = pd.DataFrame({"ID": [1,2,3], "Name": ["Amit","Riya","Kunal"]})
```

3. **From CSV File**

```
df = pd.read_csv("data.csv")
```

4. **From Excel File**

```
df = pd.read_excel("data.xlsx")
```

---

4. Inspecting Data

When working with large datasets, inspection is critical:

```
df.head()      # first 5 rows
df.tail(3)     # last 3 rows
df.shape       # (rows, columns)
df.info()      # summary: datatypes, null values
df.describe()  # statistical summary
df.columns     # list of column names
df.index       # row index
```

---

5. Indexing and Selection

Column Selection

df["Name"] # single column

df[["Name","Age"]] # multiple columns

Row Selection

df.loc[0] # by label

df.iloc[1] # by integer position

Conditional Filtering

df[df["Age"] > 25]

---

6. Data Cleaning

Data in the real world is messy. Pandas provides tools:

Detect Missing Values

df.isnull().sum()

Drop Missing Values

df.dropna(inplace=True)

Fill Missing Values

df["Age"].fillna(df["Age"].mean(), inplace=True)

☞ Real-life: If salaries have missing values, you can replace them with the **average salary**.

---

7. Data Manipulation

Adding a New Column

df["Salary"] = [50000, 60000, 45000]

Deleting a Column

df.drop("Salary", axis=1, inplace=True)

Renaming Columns

df.rename(columns={"Name":"Full_Name"}, inplace=True)

---

8. Aggregation & Grouping

GroupBy Example:

```
data = {
   "Department": ["IT", "IT", "HR", "HR", "Finance"],
   "Employee": ["A", "B", "C", "D", "E"],
   "Salary": [50000, 60000, 40000, 45000, 70000]
}
df = pd.DataFrame(data)

print(df.groupby("Department")["Salary"].mean())
```

Output:

```
Department
Finance    70000.0
HR         42500.0
IT         55000.0
```

Dry Run:

- Pandas grouped rows by department.
- For each department, calculated average salary.

---

9. Sorting

```
df.sort_values("Salary", ascending=False)
```

---

10. Merging and Joining

Merge (like SQL JOIN)

```
df1 = pd.DataFrame({"ID":[1,2], "Name":["Amit","Riya"]})
df2 = pd.DataFrame({"ID":[1,2], "City":["Delhi","Mumbai"]})

pd.merge(df1, df2, on="ID")
```

---

11. Concatenation

```
df1 = pd.DataFrame({"Name":["Amit","Riya"]})
df2 = pd.DataFrame({"Name":["Kunal","Neha"]})

pd.concat([df1, df2])
```

---

12. Pivot Tables

```
df.pivot_table(values="Salary", index="Department", aggfunc="mean")
```

---

13. Statistics with Pandas

```
df["Salary"].mean()
df["Salary"].median()
df["Salary"].std()
df.corr()  # correlation matrix
```

---

14. File Handling in Pandas

Save to CSV/Excel

```
df.to_csv("output.csv", index=False)
df.to_excel("output.xlsx", index=False)
```

Read CSV/Excel

```
pd.read_csv("output.csv")
pd.read_excel("output.xlsx")
```

---

15. High-Level Example: Sales Analysis

```
sales = {
   "Product": ["Laptop","Laptop","Mobile","Mobile","Tablet"],
   "Month": ["Jan","Feb","Jan","Feb","Jan"],
   "Sales": [200,250,300,280,150]
```

```
}
df = pd.DataFrame(sales)

# Monthly Sales
print(df.groupby("Month")["Sales"].sum())

# Product Wise Sales
print(df.groupby("Product")["Sales"].sum())

# Pivot Table
print(df.pivot_table(values="Sales", index="Product", columns="Month", aggfunc="sum"))
```

Output:

```
Month
Feb  530
Jan  650
Product
Laptop  450
Mobile  580
Tablet  150
Month   Feb  Jan
Product
Laptop  250  200
Mobile  280  300
Tablet   NaN 150
```

👉 This shows:

- Total sales per month.
- Total sales per product.
- Pivot table showing product sales across months.

Chapter 15: Exploratory Data Analysis (EDA)

15.1 What is EDA? (Introduction)

Definition:
Exploratory Data Analysis (EDA) is the process of exploring datasets to discover patterns, detect anomalies, test hypotheses, and summarize important insights—often using both **statistics** and **visualizations**.

Why Important?

- Detects **data quality issues** (missing, duplicates, wrong formats).
- Reveals **hidden relationships** (e.g., age vs. survival).
- Guides **feature selection** for machine learning.

Example (Titanic dataset):

- Question: Did women survive more than men?
- EDA answers: Yes → 74% women vs. 19% men survived.

15.2 The Workflow of EDA

EDA follows a structured pipeline:

1. **Load Data** – import dataset using pandas.
2. **Inspect Data** – shape, types, preview with .head().
3. **Clean Data** – handle missing values, duplicates, inconsistent formats.
4. **Summarize Data** – .describe() for numerical summary.
5. **Visualize Data** – histograms, boxplots, scatterplots, heatmaps.
6. **Detect Patterns** – correlations, group comparisons.
7. **Report Insights** – write down findings for decision-making.

📌 *Remember*: EDA is **iterative** (you may loop back to cleaning after visualization).

15.3 Loading and Inspecting Data

We first **import libraries** and load the dataset.

```
import pandas as pd

import seaborn as sns
```

```
import matplotlib.pyplot as plt


df = pd.read_csv("titanic.csv")
```

Inspection:

```
print(df.shape)   # (891, 12)
print(df.columns) # ['PassengerId','Survived','Pclass','Name','Sex','Age','Fare'...]
print(df.head(3)) # First 3 rows
print(df.info())  # Summary with dtypes
```

Dry Run:

- Titanic dataset → 891 rows, 12 columns.
- Age has missing values.
- Fare is float, Pclass is integer, Sex is object (string).

15.4 Data Cleaning

15.4.1 Handling Missing Values

```
df.isnull().sum()
df['Age'].fillna(df['Age'].median(), inplace=True)
df.dropna(subset=['Embarked'], inplace=True)
```

Dry run:

- Missing Age filled with median (28).
- Rows with missing Embarked removed.

15.4.2 Removing Duplicates

```
df.drop_duplicates(inplace=True)
```

15.4.3 Converting Data Types

```
df['Fare'] = df['Fare'].astype(float)
```

15.5 Descriptive Statistics

Numerical summary:

```
df['Age'].describe()
```

Output: mean ≈ 29.7, min = 0.42, max = 80.

Categorical summary:

df['Sex'].value_counts()

male: 577, female: 314

15.6 Univariate Analysis

Study **one variable** at a time.

Numerical Example (Age)

sns.histplot(df['Age'], bins=30, kde=True)

plt.show()

Dry run: Most passengers aged 20–40.

Categorical Example (Sex)

sns.countplot(x='Sex', data=df)

Dry run: More males than females.

15.7 Bivariate Analysis

Study **two variables together**.

Gender vs Survival

sns.barplot(x='Sex', y='Survived', data=df)

Dry run: Women survival ≈ 74%, Men ≈ 19%.

Class vs Survival

sns.barplot(x='Pclass', y='Survived', data=df)

Dry run: 1st class > 2nd class > 3rd class survival.

15.8 Multivariate Analysis

Study **three or more variables**.

sns.pairplot(df[['Age','Fare','Survived']], hue='Survived')

Dry run: Young passengers with higher fares had better survival.

15.9 Correlation Analysis

```
corr = df.corr()
sns.heatmap(corr, annot=True, cmap="coolwarm")
```

Dry run:

- Pclass and Fare are negatively correlated.
- Age has weak correlation with survival.

15.10 Outlier Detection

```
sns.boxplot(x=df['Fare'])
```

Dry run: Some fares > 500 are outliers (VIP tickets).

15.11 Grouping and Aggregation

```
df.groupby('Sex')['Survived'].mean()
# female: 0.74, male: 0.19
df.groupby(['Pclass','Sex'])['Survived'].mean()
```

Dry run: 1st class females had the highest survival.

15.12 Data Visualization in EDA

- **Histograms** → distributions.
- **Boxplots** → outliers.
- **Barplots** → category comparison.
- **Heatmaps** → correlations.
- **Pairplots** → multi-variable study.

15.13 Case Study 1 – Titanic Survival

Insights:

- Women survived more than men.
- 1st class survival > 3rd class.
- Children had higher chances than adults.

15.14 Case Study 2 – Sales Data

Dataset: Monthly sales across regions.

- Find top-selling region: df.groupby('Region')['Sales'].sum().
- Visualize monthly trends: sns.lineplot(x='Month', y='Sales', data=df).
- Detect anomalies: sudden spikes in sales.

15.15 Best Practices in EDA

- Always check **dtypes**.
- Treat **missing values logically**.
- Don’t remove outliers blindly—sometimes they’re important.
- Use **visualizations to confirm statistics**.
- Document findings step by step.

15.16 Summary

EDA = **Data Cleaning + Summarization + Visualization → Insights**.

It’s the **first and most important step** in any data science project. Without EDA, building ML models is like **flying blind**.

Chapter 16: Feature Engineering in Machine Learning

16.1 Introduction

Feature Engineering is the backbone of any machine learning project. You can think of raw data as **uncut diamonds** – valuable but unusable in their raw form. Feature engineering is the process of **cutting, polishing, and preparing** those diamonds so they shine in a machine learning model.

Even the most advanced algorithms (like XGBoost, Random Forests, or Neural Networks) will fail if features are messy, inconsistent, or irrelevant. On the other hand, even a simple **Logistic Regression** can achieve high accuracy if features are well engineered.

Rule of Thumb: "Garbage in, Garbage out" – poor features → poor model, no matter the algorithm.

16.2 What is Feature Engineering?

Definition:
Feature engineering is the process of **transforming raw data into meaningful input features** that improve the predictive power of machine learning models.

Example:

- Raw feature: "Date of Birth" → Engineered feature: "Age"
- Raw feature: "Text Review" → Engineered feature: "Sentiment Score"
- Raw feature: "City" → Engineered feature: "City Encoded (0/1)"

16.3 Train-Test Split Before Feature Engineering

Before starting any feature engineering, it's crucial to **split the dataset** into **training and testing sets**.

Why?
If you calculate imputation values, scaling parameters, or encodings using the **whole dataset**, your model unintentionally learns about the test set (data leakage).

Example:

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

```
import pandas as pd

df = pd.DataFrame({
    'Age': [25, 27, None, 30, 28, None, 35],
    'Salary': [50000, 54000, 60000, None, 58000, 62000, 64000],
    'Purchased': [0, 1, 0, 1, 0, 1, 1]
})

X = df[['Age', 'Salary']]
y = df['Purchased']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
```

Now, all preprocessing must be **fitted only on training data** and later **applied to test data**.

16.4 Handling Missing Values (Imputers)

Why Handle Missing Values?

- ML models cannot handle NaN.
- Missing values can bias the dataset.

Strategies:

1. **Dropping rows/columns** → Risky if too much data lost.
2. **Imputation** → Better, keeps dataset intact.

16.4.1 SimpleImputer in scikit-learn

```
from sklearn.impute import SimpleImputer
import numpy as np

# Mean imputation
```

```
mean_imputer = SimpleImputer(strategy='mean')
X_train[['Age']] = mean_imputer.fit_transform(X_train[['Age']])
X_test[['Age']] = mean_imputer.transform(X_test[['Age']])

# Median imputation
median_imputer = SimpleImputer(strategy='median')
X_train[['Salary']] = median_imputer.fit_transform(X_train[['Salary']])
X_test[['Salary']] = median_imputer.transform(X_test[['Salary']])
```

Dry Run Example:

- Age: [25, 27, NaN, 30] → mean = 27.3 → NaN replaced with 27.3.
- Salary: [50000, 54000, 60000, NaN] → median = 57000 → NaN replaced with 57000.

16.5 Encoding Categorical Data

Categorical variables like "Gender", "City", "Color" must be encoded into numbers.

16.5.1 Label Encoding (OrdinalEncoder)

- Converts categories to integers.
- Use for **ordered categories** (e.g., "Small" < "Medium" < "Large").

```
from sklearn.preprocessing import OrdinalEncoder

df = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Medium']})
encoder = OrdinalEncoder(categories=[['Small', 'Medium', 'Large']])
df['Size_encoded'] = encoder.fit_transform(df[['Size']])
```

Output:
['Small'=0, 'Medium'=1, 'Large'=2]

16.5.2 One-Hot Encoding

- Creates binary columns for each category.
- Use for **nominal data** (no order).

```
from sklearn.preprocessing import OneHotEncoder
```

df = pd.DataFrame({'Color': ['Red', 'Blue', 'Green', 'Blue']})

encoder = OneHotEncoder(sparse=False, drop='first')

encoded = encoder.fit_transform(df[['Color']])

Dry Run:

- Colors: ['Red', 'Blue', 'Green']
- drop='first' → keep only ['Green', 'Red'].
- Blue → [0,0], Green → [1,0], Red → [0,1].

16.6 Feature Scaling

Why?

If one feature is in thousands (Salary) and another in tens (Age), models like KNN, SVM, and Neural Nets get biased.

16.6.1 Standardization (Z-Score Normalization)

z=X−μσz = \frac{X - \mu}{\sigma}z=σX−μ

- Mean = 0, Std = 1.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)

16.6.2 Min-Max Normalization

X′=X−XminXmax−XminX' = \frac{X - X_{min}}{X_{max} - X_{min}}X′=Xmax−XminX−Xmin

- Scales between 0 and 1.

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

X_train_scaled = scaler.fit_transform(X_train)

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

16.6.3 Robust Scaling

- Uses median and IQR.
- Best when dataset has **outliers**.

```
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

16.7 Pipelines

Pipelines chain preprocessing + model training in **one step**.

Example: Pipeline with Imputer + Scaler + Logistic Regression

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])

pipeline.fit(X_train, y_train)
print("Accuracy:", pipeline.score(X_test, y_test))
```

- Training set: fit imputer, scaler, model.
- Test set: only transform → avoids leakage.

16.8 Transformers in scikit-learn

Transformers modify data using fit and transform.

16.8.1 PolynomialFeatures

Expands features to include powers and interactions.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X_train)
```

16.8.2 PowerTransformer

Reduces skewness in features.

```
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer(method='yeo-johnson')
X_train_transformed = pt.fit_transform(X_train)
```

16.8.3 FunctionTransformer

Custom transformations.

```
from sklearn.preprocessing import FunctionTransformer
import numpy as np

ft = FunctionTransformer(np.log1p, validate=True)
X_train_log = ft.fit_transform(X_train[['Salary']])
```

16.9 Putting It All Together (End-to-End Example)

```
from sklearn.compose import ColumnTransformer

numeric_features = ['Age', 'Salary']
categorical_features = ['City']
```

```
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(drop='first'))
])
preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('model', LogisticRegression())
])

pipeline.fit(X_train, y_train)
print("Accuracy:", pipeline.score(X_test, y_test))
```

✅ **Key Takeaways**

- Always split train-test **before preprocessing**.
- Handle missing values using **imputers**.
- Encode categorical data with **OneHotEncoder** or **OrdinalEncoder**.
- Scale features using **Standardization, MinMax, or Robust Scaling**.
- Use **Pipelines** for cleaner workflows and leakage prevention.
- Transformers like **PolynomialFeatures, PowerTransformer, FunctionTransformer** enhance feature representation.

Chapter 17: Machine Learning

17.1 Introduction to Machine Learning

What is Machine Learning?

Machine Learning (ML) is a subfield of **Artificial Intelligence (AI)** that enables computers to **learn patterns from data** without explicit programming. Instead of writing rules manually, ML systems **discover rules and relationships** automatically.

☞ Example:

- **Traditional programming:** To detect spam, you might hardcode rules like "if email contains the word lottery, mark as spam."
- **ML approach:** Provide thousands of spam and non-spam emails → algorithm learns the distinguishing patterns.

This makes ML powerful for handling **complex, large-scale, and noisy data** where rules cannot be predefined.

Why Machine Learning Matters?

1. **Handles Big Data** – Can process millions of rows quickly.
2. **Self-Learning** – Improves automatically as more data is provided.
3. **Pattern Recognition** – Finds hidden structures humans can't see.
4. **Broad Applications** – From Netflix recommendations 🎬 to medical diagnosis 🏥.

ML vs Traditional Programming

Traditional Programming	Machine Learning
Rules are written manually	Rules are learned automatically
Fixed logic	Adaptive, data-driven
Works well for simple problems	Works well for complex, data-heavy problems
Example: Calculator	Example: Face recognition

Real-Life Examples

- **Healthcare:** Predicting disease risk from patient history.

- **Finance:** Credit scoring, fraud detection.
- **Retail:** Recommendation systems.
- **Transport:** Self-driving cars.
- **NLP:** Chatbots, language translation.

17.2 Types of Machine Learning

Machine Learning is broadly divided into **three primary types** and **two advanced types**:

1. **Supervised Learning** (with labels).
2. **Unsupervised Learning** (without labels).
3. **Reinforcement Learning** (learning by feedback/rewards).
4. **Semi-Supervised Learning** (mix of labeled + unlabeled).
5. **Self-Supervised Learning** (advanced, used in deep learning).

We'll now explore each in detail.

17.3 Supervised Learning

Definition

Supervised Learning uses **labeled data**.

- Input (X) → Features.
- Output (Y) → Labels.
- Goal: Learn a mapping function **f(X) → Y**.

☞ Analogy: A teacher provides both **questions and answers** during practice.

Types of Supervised Learning

1. **Regression** – Output is continuous.
2. **Classification** – Output is categorical.

17.3.1 Regression Algorithms

Regression is about predicting **continuous values**.

a) Linear Regression

- Equation:

y=mx+cy = mx + cy=mx+c

where m = slope, c = intercept.

- Example: Predicting house price 🏠 based on area.

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

X = np.array([[1000], [1500], [2000], [2500]])   # area
y = np.array([200000, 300000, 400000, 500000])  # price

model = LinearRegression()
model.fit(X, y)
print(model.predict([[1800]]))  # Predict price of 1800 sq ft house
```

👉 Dry Run:

- Fit line through (1000, 200000) … (2500, 500000).
- At 1800 → ~360000 predicted.

```
# Linear Regression Example - Predicting House Price based on Area

import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Step 1: Prepare training data
# X -> Area of houses (input features)
# y -> Price of houses (target/output)
X = np.array([[1000], [1500], [2000], [2500]])   # in sq ft
y = np.array([200000, 300000, 400000, 500000])   # in rupees
```

```
# Step 2: Create Linear Regression model
model = LinearRegression()

# Step 3: Train (fit) the model on data
model.fit(X, y)

# Step 4: Predict price for a new area (say 1800 sq ft)
predicted_price = model.predict([[1800]])

# Step 5: Print predicted value
print("Predicted price for 1800 sq ft house:", predicted_price[0])

# ---------- DRY RUN EXPLANATION ----------
# The model finds best-fit line: y = m*x + c
# Based on training data:
# (1000, 200000), (1500, 300000), (2000, 400000), (2500, 500000)
#
# The pattern is clear:
# When area increases by 500 sq ft → price increases by 100000
# So slope (m) ≈ 100000 / 500 = 200
# Intercept (c) ≈ 0 (since at 0 area, predicted price is near 0)
#
# Hence equation becomes: y = 200*x + 0
# For x = 1800 → y = 200 * 1800 = 360000
#
# ✅ Final Output: Predicted price ≈ ₹360,000
```

---

b) Polynomial Regression

- Adds non-linearity:

y=a+b1x+b2x2+...y = a + b_1x + b_2x^2 + ...y=a+b1x+b2x2+...

- Example: Predict crop yield vs rainfall (curved trend).

---

c) Ridge & Lasso Regression

- **Ridge (L2 penalty):** Shrinks coefficients, avoids overfitting.
- **Lasso (L1 penalty):** Shrinks some coefficients to 0 (feature selection).

---

d) Decision Tree Regression

- Splits data into intervals, predicts average in each leaf.
- Example: Predict electricity consumption by temperature.

---

e) Random Forest Regression

- Collection of decision trees.
- Final prediction = average of trees.
- More accurate, less overfitting.

---

f) Support Vector Regression (SVR)

- Uses a margin around predictions.
- Example: Predicting stock prices within error boundaries.

17.3.2 Classification Algorithms

Classification predicts **labels/categories**.

a) Logistic Regression

- Outputs probability using **sigmoid function**.
- Example: Will student pass exam? (Yes/No).

```
from sklearn.linear_model import LogisticRegression
X = [[30],[50],[70],[90]]   # study hours
y = [0,0,1,1]               # fail=0, pass=1
model = LogisticRegression()
```

```
model.fit(X,y)

print(model.predict([[65]])) # predict for 65 hours
```

---

b) k-Nearest Neighbors (kNN)

- Finds nearest k neighbors → majority vote.
- Example: Classify fruit  vs  .

---

c) Decision Tree Classifier

- Splits dataset based on questions.
- Example: "Income > 50k?" → Loan Approved or Not.

---

d) Random Forest Classifier

- Combines many decision trees → votes.
- Example: Fraud detection.

---

e) Support Vector Machine (SVM)

- Finds best hyperplane to separate classes.
- Example: Spam vs non-spam emails.

---

f) Naive Bayes

- Based on Bayes Theorem.
- Works well for **text classification**.

---

g) Gradient Boosting / XGBoost

- Boosts weak learners into strong learners.
- Example: Kaggle competitions (often winners).

17.4 Unsupervised Learning

Definition

Unsupervised Learning uses **unlabeled data**.
Goal: Find hidden patterns.

☞ Analogy: Students exploring a new subject without teacher's answers.

17.4.1 Clustering Algorithms

a) K-Means Clustering

- Choose k centroids → assign points → recalc centroids.
- Example: Customer segmentation.

```
from sklearn.cluster import KMeans

X = [[1,2],[1,4],[1,0],[10,2],[10,4],[10,0]]

kmeans = KMeans(n_clusters=2)

kmeans.fit(X)

print(kmeans.labels_)
```

---

b) Hierarchical Clustering

- Builds a dendrogram/tree.
- Example: Document organization.

---

c) DBSCAN

- Groups dense clusters, ignores outliers.
- Example: Earthquake epicenters.

17.4.2 Dimensionality Reduction

a) PCA (Principal Component Analysis)

- Reduces dimensions while keeping variance.
- Example: Reduce 100 features → 2D plot.

---

b) t-SNE / UMAP

- Non-linear visualizations.

- Example: Visualizing word embeddings.

17.5 Reinforcement Learning

Definition

Agent learns by **trial and error**, maximizing rewards.

☞ Analogy: A child learns cycling 🚲 by balancing, falling, retrying.

Components

- **Agent:** Learner.
- **Environment:** World.
- **Actions:** Moves taken.
- **Rewards:** Feedback.
- **Policy:** Strategy.

Algorithms

- **Q-Learning** (stores state-action rewards).
- **Deep Q Networks (DQN)** (neural networks).
- **Policy Gradient Methods**.

Applications

- AlphaGo beating world champion in Go.
- Robotics navigation.
- Self-driving cars.

17.6 Semi-Supervised Learning

- Mix of few labeled + many unlabeled.
- Example: Auto-tagging Google Photos.

17.7 Self-Supervised Learning

- Used in **deep learning**.
- Learns from raw data itself.
- Example: GPT predicts next word.

17.8 Comparing ML Types

Type	Data	Goal	Example
Supervised	Labeled	Predict Y	Salary prediction
Unsupervised	Unlabeled	Structure	Customer groups
Reinforcement	Feedback	Maximize reward	Self-driving car
Semi-Supervised	Mixed	Better accuracy	Auto-tagging
Self-Supervised	Raw	Representations	GPT

17.9 ML Workflow

1. Data Collection.
2. Data Cleaning.
3. Feature Engineering.
4. Train-Test Split.
5. Model Training.
6. Model Evaluation.
7. Hyperparameter Tuning.
8. Deployment.

17.10 Evaluation Metrics

- **Regression:** MSE, RMSE, R^2.
- **Classification:** Accuracy, Precision, Recall, F1, ROC-AUC.
- **Clustering:** Silhouette score.

17.11 Advanced ML Concepts

- **Ensemble Learning (Bagging, Boosting, Stacking).**
- **Transfer Learning.**
- **Online Learning.**
- **Active Learning.**

17.12 Real-Life Applications

- Healthcare – Predicting cancer.
- Finance – Fraud detection.
- Retail – Recommendations.
- Transport – Self-driving cars.
- NLP – Chatbots, translation.

17.13 Summary

- ML is about teaching computers to learn.
- 3 main types: Supervised, Unsupervised, Reinforcement.
- Each type has many algorithms.
- Workflow: **Data → Preprocessing → Model → Evaluation → Deployment.**
- ML powers many industries today.

Chapter 18: Linear and Logistic Regression

18.1 Introduction to Regression

Regression is one of the most fundamental concepts in **statistics and machine learning**. It is a **supervised learning technique**, meaning the model learns from labeled data (where we already know the target value) and then makes predictions for new data.

In regression, we try to **model the relationship** between:

- **Independent Variables (X):** The predictors or input features (e.g., house size, experience, age).
- **Dependent Variable (y):** The target or outcome we want to predict (e.g., house price, salary, customer churn).

There are **two major branches of regression**:

- **Linear Regression** → used when the target variable is **continuous** (like price, marks, temperature).
- **Logistic Regression** → used when the target variable is **categorical** (like pass/fail, spam/ham, disease yes/no).

18.2 Linear Regression

18.2.1 Concept of Linear Regression

Linear Regression assumes that the relationship between input (X) and output (y) can be described by a **straight line** (or a hyperplane in higher dimensions).

👉 Example: Predicting **salary** based on **years of experience**.
If more years of experience leads to higher salary, a line can be fitted:

Salary=m×Experience+cSalary = m \times Experience + cSalary=m×Experience+c

Where:

- mmm = slope of the line (rate of increase per year of experience).
- ccc = intercept (salary when experience = 0).

18.2.2 Mathematical Representation

For **one feature** (Simple Linear Regression):

y=β0+β1x+ϵy = \beta_0 + \beta_1 x + \epsilony=β0+β1x+ϵ

For **multiple features** (Multiple Linear Regression):

y=β0+β1x1+β2x2+⋯+βnxn+ϵy = \beta_0 + \beta_1x_1 + \beta_2x_2 + \cdots + \beta_nx_n + \epsilony=β0+β1x1+β2x2+⋯+βnxn+ϵ

Where:

- yyy: dependent variable (e.g., house price).
- β0\beta_0β0: intercept.
- β1,β2,...,βn\beta_1, \beta_2, ..., \beta_nβ1,β2,...,βn: coefficients (impact of each independent variable).
- x1,x2,...,xnx_1, x_2, ..., x_nx1,x2,...,xn: independent variables.
- ϵ\epsilonϵ: error term (difference between actual and predicted).

18.2.3 Cost Function (Mean Squared Error)

The goal of Linear Regression is to **find the best line** that minimizes the difference between actual and predicted values.

We use the **Mean Squared Error (MSE)** as the cost function:

J(β)=1n∑i=1n(yi−y^i)2J(\beta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2J(β)=n1i=1∑n(yi−y^i)2

Where:

- yiy_iyi: actual value.
- y^i\hat{y}_iy^i: predicted value.
- nnn: number of data points.

☞ Smaller MSE = better fit.

18.2.4 Gradient Descent Optimization

Finding the exact best-fit line can be done with **Ordinary Least Squares (OLS)** formula. But in machine learning, we often use **Gradient Descent**, an iterative optimization algorithm.

Steps:

1. Start with random coefficients.
2. Compute cost (MSE).
3. Update coefficients using gradient:

βj=βj−α∂J∂βj\beta_j = \beta_j - \alpha \frac{\partial J}{\partial \beta_j}βj=βj−α∂βj∂J

Where α\alphaα = learning rate.
4. Repeat until convergence.

👉 Example Dry Run:
Suppose we predict **salary** from **experience**. Initially:

- Coefficients random → bad predictions.
- Gradient descent adjusts them step by step until the line fits data well.

18.2.5 Types of Linear Regression

1. **Simple Linear Regression**
    - One independent variable.
    - Example: Predicting **salary** from **experience**.
2. **Multiple Linear Regression**
    - More than one independent variable.
    - Example: Predicting **house price** from **size + location + age**.
3. **Polynomial Regression**
    - Fits a **curve** instead of a line.
    - Example: Relationship between **temperature** and **ice-cream sales** may look like a parabola.
4. **Ridge Regression**
    - Uses **L2 Regularization** → adds penalty for large coefficients.
    - Prevents overfitting.
5. **Lasso Regression**
    - Uses **L1 Regularization** → some coefficients become zero.
    - Helps in **feature selection**.
6. **Elastic Net**
    - Combination of **Lasso + Ridge**.

18.2.6 Example: Predicting House Prices

Dataset:

Size (sq ft)	Bedrooms	Price (₹ Lakhs)

1000	2	50
1500	3	65
2000	4	90

Model learns:

Price=β0+β1(Size)+β2(Bedrooms)Price = \beta_0 + \beta_1(Size) + \beta_2(Bedrooms)Price=β0+β1(Size)+β2(Bedrooms)

Suppose after training:

Price=10+0.03×Size+5×BedroomsPrice = 10 + 0.03 \times Size + 5 \times BedroomsPrice=10+0.03×Size+5×Bedrooms

👉 For 1800 sq ft, 3 bedrooms:

Price=10+0.03(1800)+5(3)=10+54+15=79 LakhsPrice = 10 + 0.03(1800) + 5(3) = 10 + 54 + 15 = 79 \, LakhsPrice=10+0.03(1800)+5(3)=10+54+15=79Lakhs

---

18.3 Logistic Regression

18.3.1 Concept of Logistic Regression

Logistic Regression is used when the target variable is **categorical**.
Instead of fitting a line, it models **probabilities**.

👉 Example: Predicting whether a student **passes (1) or fails (0)** an exam.

18.3.2 The Sigmoid Function

Logistic regression uses the **Sigmoid (Logistic) function**:

σ(z)=11+e−z\sigma(z) = \frac{1}{1 + e^{-z}}σ(z)=1+e−z1

- Input: any real number.
- Output: between 0 and 1.

👉 If P>0.5P > 0.5P>0.5 → Predict Class = 1
👉 Else → Class = 0

18.3.3 Cost Function (Log Loss)

Instead of MSE, Logistic Regression uses **Log Loss** (Cross-Entropy):

J(θ)=−1n∑i=1n[yilog(y^i)+(1−yi)log(1−y^i)]J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \Big[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i) \Big]J(θ)=−n1i=1∑n[yilog(y^i)+(1−yi)log(1−y^i)]

This punishes wrong confident predictions heavily.

18.3.4 Gradient Descent in Logistic Regression

Just like Linear Regression, weights are updated iteratively.
Difference: cost function is **Log Loss**.

18.3.5 Types of Logistic Regression

1. **Binary Logistic Regression**
    - Two categories.
    - Example: Email **spam or not**.
2. **Multinomial Logistic Regression**
    - More than two categories.
    - Example: Predicting **fruit type → Apple, Banana, Orange**.
3. **Ordinal Logistic Regression**
    - Categories with order.
    - Example: Movie rating → Poor, Average, Good, Excellent.

18.3.6 Example: Predicting Customer Churn

Dataset:

Age	Monthly Charges	Churn (1 = Yes, 0 = No)
25	500	0
40	1200	1
35	800	0

Model:

$P(Churn=1)=\sigma(\beta0+\beta1\times Age+\beta2\times Charges)P(Churn=1) = \sigma(\beta_0 + \beta_1 \times Age + \beta_2 \times Charges)P(Churn=1)=\sigma(\beta0+\beta1\times Age+\beta2\times Charges)$

Suppose model gives:

- Age = 30, Charges = 1000 → P(Churn=1) = 0.72 → Predict = Yes (1).

18.4 Key Differences Between Linear and Logistic Regression

Feature	Linear Regression	Logistic Regression
Output	Continuous values	Probability (0–1)
Target Variable	Continuous (e.g., salary)	Categorical (e.g., churn yes/no)
Cost Function	Mean Squared Error	Log Loss
Line/Curve	Straight Line	Sigmoid Curve
Use Cases	Predict prices, marks, growth	Predict churn, fraud detection, disease outcome

---

18.5 Advantages and Limitations

✅ **Advantages:**

- Easy to implement and interpret.
- Computationally efficient.
- Good for linearly separable data.

❌ **Limitations:**

- Linear regression assumes linearity (fails for non-linear patterns).
- Logistic regression struggles with complex, non-linear boundaries.
- Sensitive to outliers.

18.6 Practical Implementation (Python)

```
import pandas as pd
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.model_selection import train_test_split

# Linear Regression Example
data = pd.DataFrame({
   'Experience': [1, 2, 3, 4, 5],
   'Salary': [30000, 35000, 40000, 45000, 50000]
})
```

```
X = data[['Experience']]

y = data['Salary']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

model = LinearRegression()

model.fit(X_train, y_train)

print("Coefficient:", model.coef_)

print("Intercept:", model.intercept_)

print("Predicted Salary for 6 years exp:", model.predict([[6]]))

# Logistic Regression Example

data2 = pd.DataFrame({

   'Age': [22, 25, 30, 35, 40],

   'Churn': [0, 0, 1, 1, 1]

})

X = data2[['Age']]

y = data2['Churn']

log_model = LogisticRegression()

log_model.fit(X, y)

print("Probability of churn at age 28:", log_model.predict_proba([[28]]))
```

✅ Summary

- **Linear Regression** → Predicts continuous values. Uses **MSE**. Assumes linear relationship.
- **Logistic Regression** → Predicts categorical outcomes. Uses **Log Loss**. Uses **sigmoid** function.
- Both are **fundamental ML algorithms** and widely used in business, healthcare, finance, and science.

Chapter -19 Gradient Descent and Its Variants (Complete, Practical, and Mathematical)

Gradient descent (GD) is the workhorse optimization algorithm behind most machine-learning model training. In this chapter you'll get a complete, practical, and mathematical tour of gradient descent: the basic idea, the main variants (batch, stochastic, mini-batch), advanced optimizers (momentum, Nesterov, AdaGrad, RMSProp, Adam, AdamW, Nadam, AMSGrad), learning-rate schedules, practical tips (initialization, normalization, clipping), and when to use which method. Every variant includes formulas, pseudocode, complexity, and intuitive explanation. You'll also see small worked numeric examples (dry runs) and runnable Python snippets.

19.1 What is Gradient Descent? — Intuition & Formalism

Intuition. You have a differentiable loss function J(θ)J(\theta)J(θ) of parameters θ\thetaθ (e.g., weights of a model). Gradient descent moves θ\thetaθ iteratively in the direction of **steepest descent** (negative gradient) to reduce the loss.

Basic update rule (gradient descent):

θt+1=θt−η∇θJ(θt)\theta_{t+1} = \theta_t - \eta \nabla_{\theta} J(\theta_t)θt+1=θt−η∇θJ(θt)

- θt\theta_tθt: parameter vector at iteration ttt.
- η>0\eta>0η>0: learning rate (step size).
- ∇θJ(θt)\nabla_{\theta} J(\theta_t)∇θJ(θt): gradient of loss w.r.t parameters.

Interpretation: The gradient points uphill; subtracting it moves downhill. The learning rate controls step length.

19.2 Batch / Full Gradient Descent

Definition. Compute gradient over the entire training set at each step.

Update:

θt+1=θt−η1N∑i=1N∇θℓ(θt;xi,yi)\theta_{t+1} = \theta_t - \eta \frac{1}{N}\sum_{i=1}^{N} \nabla_{\theta} \ell(\theta_t; x_i, y_i)θt+1=θt−ηN1i=1∑N∇θℓ(θt;xi,yi)

where ℓ(θ;xi,yi)\ell(\theta;x_i,y_i)ℓ(θ;xi,yi) is loss on example iii and NNN is dataset size.

Pros

- Deterministic direction (no gradient noise).
- Converges smoothly for convex problems.

Cons

- Expensive for large NNN (computationally heavy).
- Memory & time per iteration high.

Complexity: O(N·c)O(N \cdot c)O(N·c) per update where ccc = cost to compute gradient per example.

Use case: Small datasets or when exact gradients are desired; rarely used for deep learning.

19.3 Stochastic Gradient Descent (SGD)

Definition. Update using gradient from a single (random) example.

Update (for example iti_tit chosen at time ttt):

θt+1=θt−η∇θℓ(θt;xit,yit)\theta_{t+1} = \theta_t - \eta \nabla_{\theta} \ell(\theta_t; x_{i_t}, y_{i_t})θt+1=θt−η∇θℓ(θt;xit,yit)

Pros

- Very cheap per update (one example).
- Enables online learning (data streaming).
- Helps escape shallow local minima/saddle points because of gradient noise.

Cons

- Noisy updates → loss fluctuates; needs careful learning-rate schedule.
- Typically requires many iterations to converge.

Practical note: Shuffle data and iterate over dataset to get one epoch.

19.4 Mini-Batch Gradient Descent (Most practical)

Definition. Use a small batch of size BBB (e.g., 32, 64, 256) to estimate gradient.

Update:

θt+1=θt−η1B∑j=1B∇θℓ(θt;xij,yij)\theta_{t+1} = \theta_t - \eta \frac{1}{B}\sum_{j=1}^B \nabla_{\theta} \ell(\theta_t; x_{i_j}, y_{i_j})θt+1=θt−ηB1j=1∑B∇θℓ(θt;xij,yij)

Why mini-batch?

- Compromise: lower variance than SGD, cheaper than full GD.
- Allows efficient vectorized computation on GPUs.
- Choice of batch size is a key hyperparameter.

Common batch sizes: 32, 64, 128 — depends on memory and model.

19.5 Worked numeric dry-run: 1D quadratic (demonstrates behavior)

Objective: f(x)=(x−3)2f(x)=(x-3)^2f(x)=(x−3)2. Gradient: f′(x)=2(x−3)f'(x)=2(x-3)f′(x)=2(x−3). Min at x=3x=3x=3.

Take initial x0=0x_0 = 0x0=0, learning rate η=0.1\eta=0.1η=0.1. Show first 4 iterations (exact arithmetic):

- Iteration 0: x0=0x_0=0x0=0. Gradient g0=2(0−3)=−6g_0=2(0-3)= -6g0=2(0−3)=−6.
  Update: x1=0−0.1×(−6)=0+0.6=0.6x_1 = 0 - 0.1 \times (-6) = 0 + 0.6 = 0.6x1 =0−0.1×(−6)=0+0.6=0.6.
- Iteration 1: x1=0.6x_1=0.6x1=0.6. Gradient g1=2(0.6−3)=2(−2.4)=−4.8g_1=2(0.6-3)=2(-2.4)=-4.8g1=2(0.6−3)=2(−2.4)=−4.8.
  Update: x2=0.6−0.1×(−4.8)=0.6+0.48=1.08x_2 = 0.6 - 0.1 \times (-4.8) = 0.6 + 0.48 = 1.08x2=0.6−0.1×(−4.8)=0.6+0.48=1.08.
- Iteration 2: x2=1.08x_2=1.08x2=1.08. Gradient g2=2(1.08−3)=2(−1.92)=−3.84g_2=2(1.08-3)=2(-1.92)=-3.84g2 =2(1.08−3)=2(−1.92)=−3.84.
  Update: x3=1.08−0.1×(−3.84)=1.08+0.384=1.464x_3 = 1.08 - 0.1 \times (-3.84) = 1.08 + 0.384 = 1.464x3=1.08−0.1×(−3.84)=1.08+0.384=1.464.
- Iteration 3: x3=1.464x_3=1.464x3=1.464. Gradient g3=2(1.464−3)=2(−1.536)=−3.072g_3=2(1.464-3)=2(-1.536)=-3.072g3 =2(1.464−3)=2(−1.536)=−3.072.
  Update: x4=1.464−0.1×(−3.072)=1.464+0.3072=1.7712x_4 = 1.464 - 0.1 \times (-3.072) = 1.464 + 0.3072 = 1.7712x4=1.464−0.1×(−3.072)=1.464+0.3072=1.7712.

You can see convergence toward 3.

19.6 Practical issues with basic GD/SGD

- **Choosing learning rate η\etaη**: too large → divergence; too small → slow convergence.
- **Ill-conditioned problems**: different curvature directions slow learning (eigenvalues of Hessian differ).
- **Saddle points & local minima** in non-convex objective (neural networks).
- **Gradient noise**: SGD introduces noise beneficial for escaping saddles but harmful for precise convergence.

Mitigations: momentum, adaptive learning rates, second-order methods.

19.7 Momentum — Smooth and accelerate descent

Idea. Accumulate an exponentially decaying moving average of gradients to build "velocity". Helps accelerate in shallow directions and damp oscillations.

Equations (momentum SGD):

vt=γvt−1+η∇θJ(θt)θt+1=θt−vtv_{t} = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta_{t})\\ \theta_{t+1} = \theta_t - v_tvt=γvt−1+η∇θJ(θt)θt+1=θt−vt

- γ∈[0,1)\gamma\in[0,1)γ∈[0,1): momentum coefficient (typical 0.9).
- v0=0v_0 = 0v0=0.

Equivalent textbook form:

vt=γvt−1+ηgtθt+1=θt−vtv_{t} = \gamma v_{t-1} + \eta g_t\\ \theta_{t+1} = \theta_t - v_tvt =γvt−1+ηgtθt+1=θt−vt

with gt=∇J(θt)g_t=\nabla J(\theta_t)gt=∇J(θt).

Effect: Momentum averages gradients to push through small gradients and reduce oscillations across ravines.

Pseudocode (momentum):

```
v = 0
for t in 1..T:
    g = grad(theta)
    v = gamma * v + eta * g
    theta = theta - v
```

Hyperparams: γ\gammaγ ≈ 0.9, η\etaη tuned (often lower than without momentum).

19.8 Nesterov Accelerated Gradient (NAG / Nesterov Momentum)

Idea. Look ahead: compute gradient at the approximated future position θ−γv\theta - \gamma vθ−γv rather than current θ\thetaθ. More responsive.

Equations:

vt=γvt−1+η∇θJ(θt−γvt−1)θt+1=θt−vtv_{t} = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta_t - \gamma v_{t-1})\\ \theta_{t+1} = \theta_t - v_tvt=γvt−1+η∇θJ(θt−γvt−1)θt+1=θt−vt

Intuition: Nesterov reduces overshoot since gradient is computed at the lookahead point.

Practical: Often faster than standard momentum.

19.9 Adaptive learning-rate methods — why?

Different parameters may require different step sizes. Adaptive optimizers adapt per-parameter learning rates using gradient history.

Main families:

- **AdaGrad** — accumulative squared gradients (good for sparse features).
- **RMSProp** — exponentially decaying average of squared gradients (fixes AdaGrad's aggressive decay).
- **Adam** — combines momentum (first moment) + RMSProp (second moment). Widely used.

19.9.1 AdaGrad

Equations:

- Accumulate squared gradients: Gt=Gt−1+gt⊙gtG_{t} = G_{t-1} + g_t \odot g_tGt =Gt−1+gt⊙gt (element-wise).
- Update:

θt+1=θt−ηgtGt+ϵ\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{G_t} + \epsilon}θt+1=θt−ηGt +ϵgt

- ϵ\epsilonϵ small (e.g., 10−810^{-8}10−8) to avoid division by zero.

Effect: Parameters with frequent large gradients get smaller effective learning rate. Good for sparse data (NLP).

Downside: GtG_tGt monotonically increases → step sizes shrink and may become very small (learning stalls).

```
import numpy as np


# --- Initialize Parameters and Hyperparameters ---

# Assume we have 3 parameters (weights/biases) in a layer

theta = np.array([1.0, 2.0, 3.0])


# Learning Rate (Global)

eta = 0.01
```

```
# Epsilon to prevent division by zero
epsilon = 1e-8

# G_t: Accumulator for squared gradients (Initialized to zeros)
# This is the "memory" term that holds the history.
G_t = np.array([0.0, 0.0, 0.0])

# --- Simulation of a Single Optimization Step ---

# 1. Calculate the current gradients (g_t)
# Example: Parameter 1 has a SMALL, frequent gradient.
#          Parameter 2 has a LARGE, infrequent gradient.
#          Parameter 3 has a MODERATE gradient.
g_t = np.array([0.1, 5.0, 1.0])

print(f"Initial Parameters (θ): {theta}")
print(f"Current Gradients (g_t): {g_t}")
print("-" * 30)

# 2. Accumulate squared gradients (G_t = G_{t-1} + g_t ⊙ g_t)
# Note: The large gradient in g_t[1] (5.0) will dominate G_t[1].
G_t += g_t * g_t
print(f"Accumulated Squared Gradients (G_t): {G_t}")

# 3. Calculate the adaptive learning rate term for each parameter
# Effective Learning Rate = η / (sqrt(G_t) + ϵ)
adaptive_lr_term = eta / (np.sqrt(G_t) + epsilon)
print(f"Adaptive LR Term (η / (sqrt(G_t) + ϵ)): {adaptive_lr_term}")
```

```
# 4. Calculate the update step
# Update = η * g_t / (sqrt(G_t) + ϵ)
update_step = adaptive_lr_term * g_t
print(f"Update Step: {update_step}")

# 5. Update the parameters (θ_{t+1} = θ_t - Update)
theta_new = theta - update_step
print("-" * 30)
print(f"New Parameters (θ_{{t+1}}): {theta_new}")
```

- The key is that the term acts as a **normalizer**. Parameters that have accumulated large squared gradients (like Parameter 1,) get a much smaller effective learning rate, ensuring they don't take massive steps and overshoot.
- Parameters with historically small gradients (like Parameter 0,) get a larger effective learning rate, allowing them to catch up.

---

2. Visualization Description

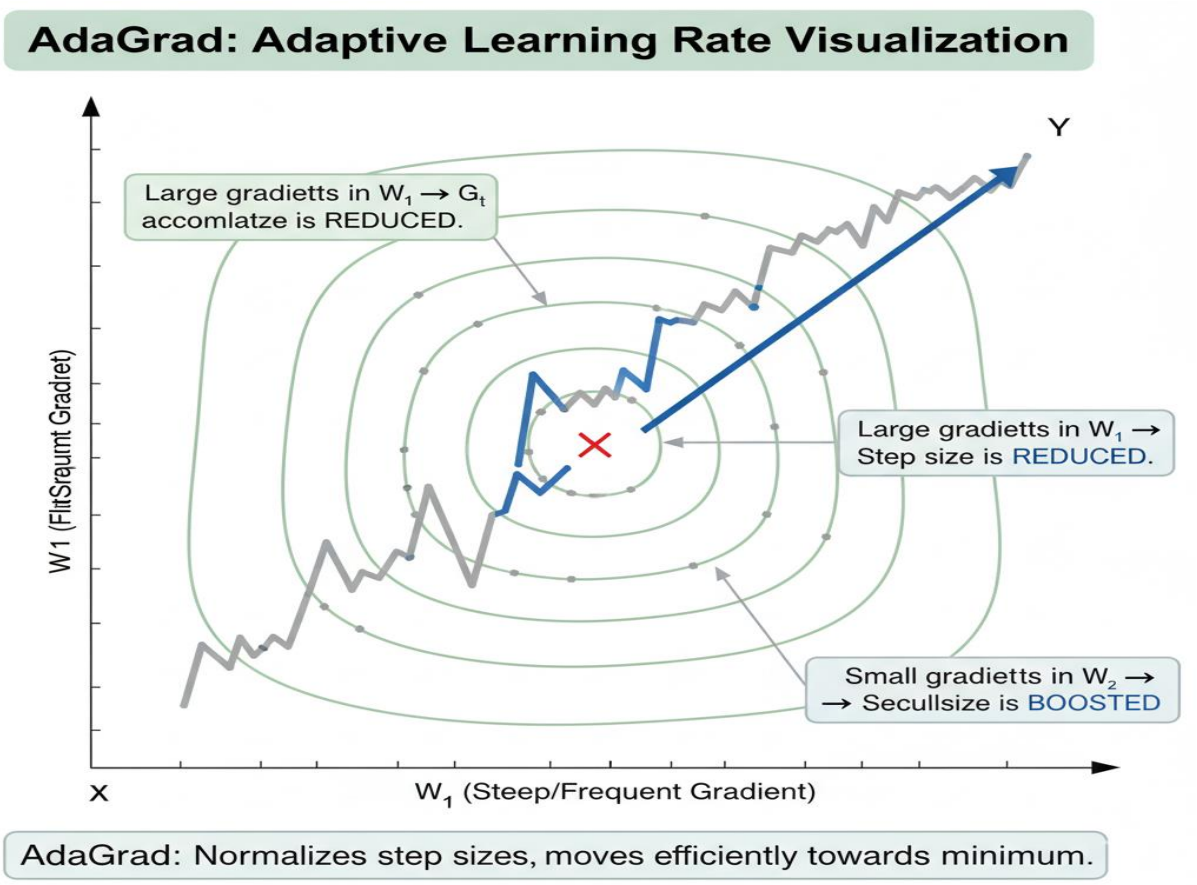
A visual explanation would focus on the effect of the accumulator () on the step size over time, contrasting AdaGrad with standard Stochastic Gradient Descent (SGD).

A. The Adaptive Step Size (Initial Training Phase)

- **Setup:** A 2D contour plot of the loss function, with two parameters, and .
- **(Flat/Sparse Feature):** The gradient for is small and infrequent.
- **(Steep/Frequent Feature):** The gradient for is large and frequent.
- **SGD Path (Contrast):** The path would jitter wildly in the direction of (large steps) and crawl slowly in the direction of (small steps).
- **AdaGrad Path (Visualization):**
    1. The large gradients for cause its to grow quickly. Its step size shrinks rapidly.
    2. The small gradients for cause its to grow slowly. Its step size remains relatively large.

3. **Result:** The path moves much more directly toward the optimum. AdaGrad effectively "boosts" the learning rate for the sparse/flat dimensions () and "dampens" the learning rate for the frequent/steep dimensions ().

B. The Downside: Learning Stalling (Late Training Phase)



- **Setup:** Plot the effective learning rate over many epochs.
- **SGD Line:** A single flat horizontal line representing a constant learning rate .
- **AdaGrad Lines:**
  - **The Trend:** All AdaGrad lines (one for each parameter) are constantly decreasing curves.
  - **The Problem:** Because is a sum of positive terms (), it can only increase. As training progresses to a very large number of epochs, , causing the effective learning rate .
  - **Visual:** The lines will all approach zero, demonstrating that the step size for **every parameter** shrinks towards zero, causing the optimization process to **stall** before fully reaching the minimum. This is the main motivation for its successor, RMSProp.

19.9.2 RMSProp

Core fix: replace full sum in AdaGrad with exponential moving average of squared gradients.

Equations:

st=βst−1+(1−β)gt2θt+1=θt−ηgtst+ϵs_t = \beta s_{t-1} + (1-\beta) g_t^2\\ \theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{s_t} + \epsilon}st=βst−1+(1−β)gt2θt+1=θt−ηst+ϵgt

- β\betaβ (decay) ≈ 0.9.

Effect: Prevents AdaGrad step size vanishing; adapts quickly to recent gradient magnitudes.

Equations

$$s_t = \beta s_{t-1} + (1-\beta) g_t^2$$
$$\theta_{t+1} = \theta_t - \eta \frac{g_t}{\sqrt{s_t} + \epsilon}$$

where:

- s_t: moving average of squared gradients
- $\beta = 0.9$: decay rate
- η: learning rate
- g_t: gradient
- ϵ: small constant (1e-8)

□ Example Problem

We'll minimize a simple quadratic function:

$$f(\theta) = \theta^2$$

The derivative (gradient) is:

$$g = 2\theta$$

We'll apply **RMSProp** to find θ → 0 (the minimum).

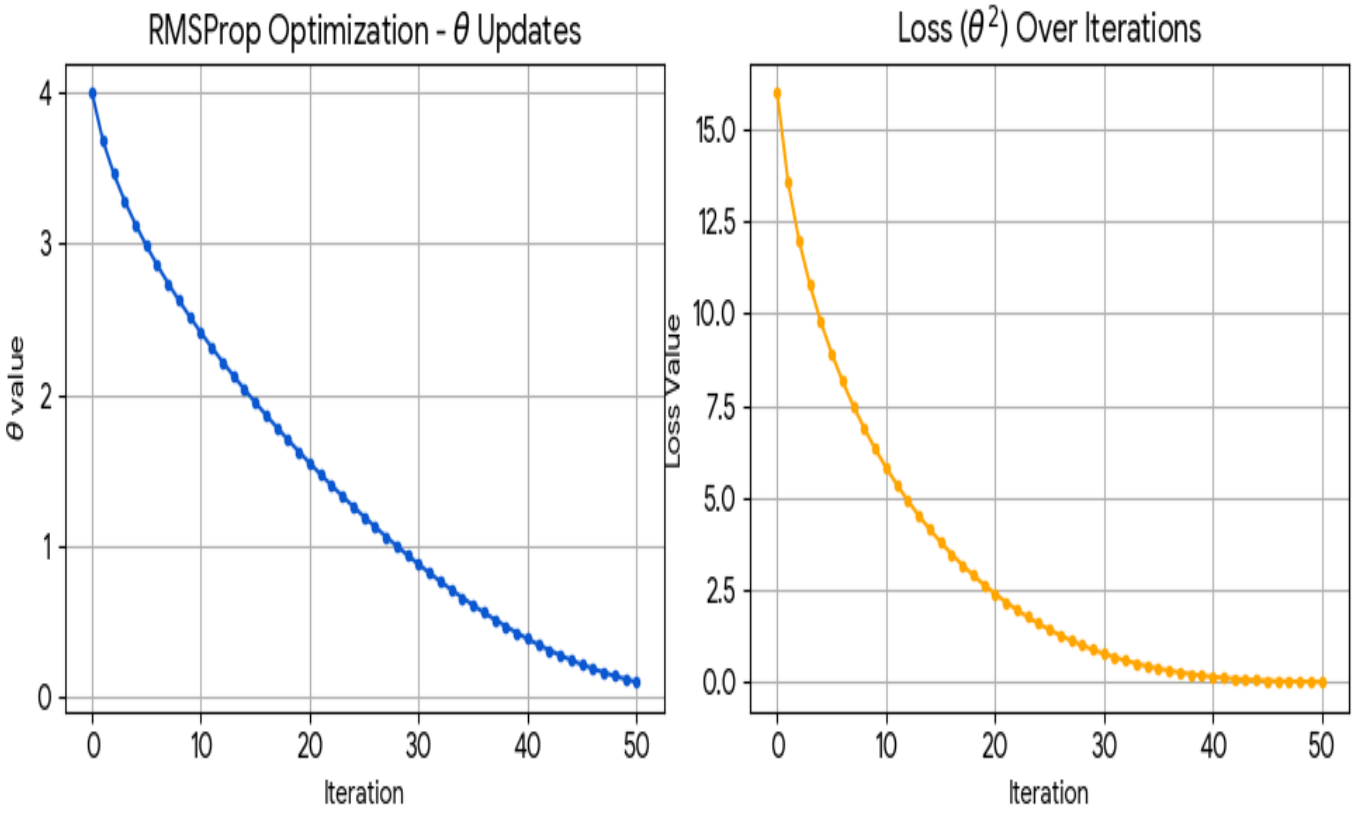
□ Code Implementation with Plot

import numpy as np

```
import matplotlib.pyplot as plt

# Function and its gradient
def f(theta):
    return theta ** 2

def grad(theta):
    return 2 * theta

# Initialize parameters
theta = 4.0             # starting point
eta = 0.1               # learning rate
beta = 0.9              # decay rate
epsilon = 1e-8          # small constant
s = 0                   # initialize moving average
iterations = 50         # number of steps

# For plotting
theta_history = [theta]
loss_history = [f(theta)]

# RMSProp loop
for t in range(1, iterations + 1):
    g = grad(theta)
    s = beta * s + (1 - beta) * (g ** 2)     # moving average of squared gradients
    theta = theta - eta * g / (np.sqrt(s) + epsilon)   # update parameter

    theta_history.append(theta)
    loss_history.append(f(theta))
```

```
# Plot results
plt.figure(figsize=(10,4))

# Plot 1: θ value change
plt.subplot(1, 2, 1)
plt.plot(theta_history, marker='o')
plt.title("RMSProp Optimization - θ Updates")
plt.xlabel("Iteration")
plt.ylabel("θ value")
plt.grid(True)

# Plot 2: Loss function
plt.subplot(1, 2, 2)
plt.plot(loss_history, marker='o', color='orange')
plt.title("Loss (θ²) Over Iterations")
plt.xlabel("Iteration")
plt.ylabel("Loss Value")
plt.grid(True)

plt.tight_layout()
plt.show()
```

```
print(f"Final θ after optimization: {theta:.4f}")
```



□ Dry Run Explanation

Iter	θ (parameter)	Gradient (2θ)	s (EMA of grad²)	Update Step
1	4.0	8.0	6.4	Big step down
2	↓	smaller	smoother	smaller step
10	~0.5	~1.0	stabilizes	very small updates
50	≈ 0	≈ 0	stabilizes	converged ✅

🎯 Effect

- RMSProp **adapts learning rate automatically** per parameter.
- Step size doesn't vanish (like AdaGrad).
- More stable than standard SGD, especially for non-stationary problems.

19.9.3 Adam (Adaptive Moment Estimation)

Combines momentum (first moment) and RMSProp (second moment) with bias correction.

Equations:

mt=β1mt−1+(1−β1)gt(first moment)vt=β2vt−1+(1−β2)gt2(second moment)m^t=mt1−β1t,v^t=vt1−β2tθt+1=θt−ηm^tv^t+ϵm_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad\text{(first moment)}\\ v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad\text{(second moment)}\\ \hat{m}_t = \frac{m_t}{1-\beta_1^t},\quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}\\ \theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}mt=β1mt−1+(1−β1)gt (first moment)vt=β2vt−1+(1−β2)gt2(second moment)m^t=1−β1tmt,v^t=1−β2tvtθt+1=θt −ηv^t+ϵm^t

Defaults: β1=0.9\beta_1=0.9β1=0.9, β2=0.999\beta_2=0.999β2=0.999, η=10−3\eta=10^{-3}η=10−3, ϵ=10−8\epsilon=10^{-8}ϵ=10−8.

Why bias correction? mt,vtm_t, v_tmt,vt start at 0 and are biased toward 0 early on; dividing by 1−β1t,1−β2t1-\beta_1^t,1-\beta_2^t1−β1t,1−β2t corrects that.

Pros: Fast, works well out-of-the-box for many architectures.
Cons: May generalize worse than SGD with momentum on some tasks; can converge to sharp minima.

Equations

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \text{(First moment - momentum)}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \text{(Second moment - RMSProp part)}$$

Bias corrections (since both start from 0):

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Update rule:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Default Parameters

Symbol	Meaning	Default
β_1	Decay rate for 1st moment	0.9

Symbol	Meaning	Default
β_2	Decay rate for 2nd moment	0.999
η	Learning rate	0.001
ε	Small constant	1e-8

🎯 We'll Minimize

$$f(\theta) = \theta^2$$

→ Gradient: $g = 2\theta$

Goal: Move θ from 4 → 0 (global minimum).

💻 Full Python Code with Plot

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

# Function and its gradient
def f(theta):
    return theta ** 2

def grad(theta):
    return 2 * theta

# Initialize parameters
theta = 4.0       # starting point
eta = 0.1         # learning rate
beta1 = 0.9       # for momentum (1st moment)
beta2 = 0.999     # for RMSProp (2nd moment)
epsilon = 1e-8
m = 0             # first moment (momentum)
v = 0             # second moment (variance)
```

```
iterations = 50

# Lists to store for plotting
theta_history = [theta]
loss_history = [f(theta)]

# Adam Optimization Loop
for t in range(1, iterations + 1):
    g = grad(theta)

    # Update biased first & second moments
    m = beta1 * m + (1 - beta1) * g
    v = beta2 * v + (1 - beta2) * (g ** 2)

    # Bias correction
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)

    # Parameter update
    theta = theta - eta * m_hat / (np.sqrt(v_hat) + epsilon)

    theta_history.append(theta)
    loss_history.append(f(theta))

# ---- Plotting ----
plt.figure(figsize=(10,4))

# Plot 1: θ changes
plt.subplot(1, 2, 1)
```

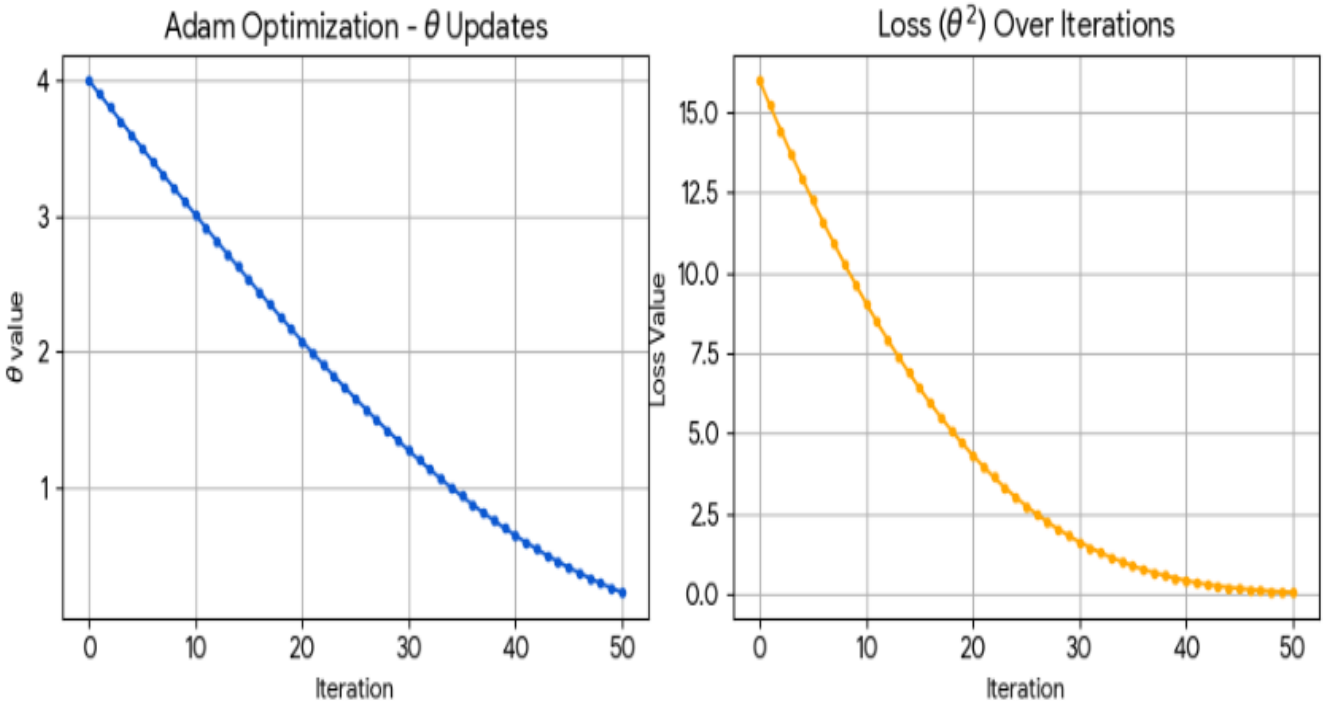
```
plt.plot(theta_history, marker='o')
plt.title("Adam Optimization - θ Updates")
plt.xlabel("Iteration")
plt.ylabel("θ value")
plt.grid(True)

# Plot 2: Loss curve
plt.subplot(1, 2, 2)
plt.plot(loss_history, marker='o', color='orange')
plt.title("Loss (θ²) Over Iterations")
plt.xlabel("Iteration")
plt.ylabel("Loss Value")
plt.grid(True)

plt.tight_layout()
plt.show()

print(f"Final θ after optimization: {theta:.6f}")
```

□ Dry Run Explanation

Iter	Gradient (g=2θ)	m (momentum)	v (grad² avg)	θ (parameter)
1	8.0	0.8	0.0128	3.74
2	7.48	1.47	0.025	3.48
10	1.2	1.05	0.001	0.42
30	0.04	0.03	0.000001	~0.05
50	≈ 0	≈ 0	stable	→ 0 ✅

📊 Plot Interpretation

- Left graph: θ rapidly decreases → flattens near 0 (minimum)
- Right graph: Loss decreases exponentially fast → smooth curve
- Bias correction ensures stable start (no "slow warm-up")



19.9.4 AdamW (decoupled weight decay)

Problem: Classic Adam applies L2 weight decay incorrectly with adaptive updates. AdamW decouples weight decay from gradient-based updates.

Update (AdamW):

θt+1=θt−η(m^tv^t+ϵ+λθt)\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right)θt+1=θt−η(v^t +ϵm^t+λθt)

Here λ\lambdaλ is weight decay. This decoupling improves regularization behavior.

19.9.5 AMSGrad, Nadam

- **AMSGrad:** variant of Adam that enforces non-decreasing second moment to improve convergence guarantees. Replaces vtv_tvt with v^t=max(v^t−1,vt)\hat{v}_t = \max(\hat{v}_{t-1}, v_t)v^t=max(v^t−1,vt).
- **Nadam:** Adam + Nesterov momentum (first moment uses lookahead).

19.10 Pseudocode summary of popular optimizers

SGD (mini-batch):

```
for each epoch:
  for each batch:
    g = grad(theta) on batch
    theta = theta - eta * g
```

SGD + momentum:

```
v = 0
for batch:
  g = grad(theta)
  v = gamma * v + eta * g
  theta = theta - v
```

RMSProp:

```
s = 0
for batch:
  g = grad(theta)
  s = beta * s + (1-beta) * g*g
  theta = theta - eta * g / (sqrt(s) + eps)
```

Adam:

```
m = 0; v = 0; t = 0
for batch:
  t += 1
  g = grad(theta)
  m = beta1*m + (1-beta1)*g
  v = beta2*v + (1-beta2)*(g*g)
  mhat = m/(1-beta1**t)
  vhat = v/(1-beta2**t)
  theta = theta - eta * mhat / (sqrt(vhat) + eps)
```

19.11 Learning-Rate Schedules & Warm Restarts

Fixed learning rate is simple but suboptimal. Schedules reduce learning rate over time to allow convergence.

Common schedules:

1. **Step decay:** reduce η\etaη by factor every kkk epochs. ηt=η0·drop⌊t/k⌋\eta_t = \eta_0 \cdot \text{drop}^{\lfloor t / k\rfloor}ηt=η0·drop⌊t/k⌋
2. **Exponential decay:** ηt=η0·exp(−λt)\eta_t = \eta_0 \cdot \exp(-\lambda t)ηt=η0 ·exp(−λt)
3. **1/t decay:** ηt=η0/(1+αt)\eta_t = \eta_0 / (1 + \alpha t)ηt=η0/(1+αt)
4. **Cosine annealing:** ηt\eta_tηt follows cosine curve to 0 then restarts (used in SGDR).
5. **Cyclical LR:** oscillates between bounds to escape saddle points.
6. **Warmup:** start with very small η\etaη and gradually increase for first few epochs (important in transformer training).

Practical rule: Use larger learning rate early for exploration, reduce for finetuning.

19.12 Convergence & Theoretical Notes (brief)

- For convex JJJ with Lipschitz continuous gradient, gradient descent with step size 0<η<2/L0<\eta<2/L0<η<2/L converges to global minimum (LLL is Lipschitz constant related to maximum eigenvalue of Hessian).
- SGD convergence is in expectation and requires decreasing ηt\eta_tηt schedules (e.g., ηt∝1/t\eta_t \propto 1/tηt∝1/t) for theoretical guarantees.
- For non-convex (neural nets) we have no global guarantee; SGD variants perform well empirically.

19.13 Second-Order & Quasi-Newton Methods

Motivation: First-order methods (GD) use gradient only. Second-order methods use curvature (Hessian) to take more informed steps.

Newton's method:

θt+1=θt−H(θt)−1∇J(θt)\theta_{t+1} = \theta_{t} - H(\theta_t)^{-1} \nabla J(\theta_t)θt+1=θt −H(θt)−1∇J(θt)

- HHH is Hessian matrix. Converges fast (quadratic near optimum) but expensive: computing and inverting Hessian is O(d3)O(d^3)O(d3).

Quasi-Newton (BFGS, L-BFGS):

- Approximate inverse Hessian using rank-one updates.
- **L-BFGS** stores limited history (memory efficient) — commonly used in convex optimization and smaller models.

When to use: small to medium sized convex problems (logistic regression, small neural nets). For big deep nets, second-order methods are usually too expensive.

19.14 Practical tips & best practices

1. **Normalize / Standardize inputs** (zero mean, unit variance) — helps gradient methods converge faster.
2. **Use appropriate optimizer:** Adam/RMSProp for fast convergence; SGD+momentum often yields better final generalization in deep nets.
3. **Tune learning rate first** — it's the most important hyperparameter. Use LR range test (increase lr exponentially and monitor loss).
4. **Use batch normalization** to stabilize layer inputs.
5. **Weight initialization**: Xavier/Glorot for tanh/sigmoid; He init for ReLU. Good init prevents vanishing/exploding gradients.
6. **Gradient clipping** (by norm or value) prevents exploding gradients in RNNs: clip if ‖∇‖>τ\|\nabla\| > \tau‖∇‖>τ.
7. **Early stopping**: monitor validation loss and stop if no improvement to avoid overfitting.
8. **Use proper weight decay** (L2) for regularization; with Adam prefer AdamW decoupled weight decay.
9. **Use smaller batches** when memory limited; larger batches can leverage GPU throughput but may need lr scaling.

19.15 Which optimizer to pick?

- **Small convex problems:** use **L-BFGS** or **Batch GD** with line search.
- **Large-scale deep networks:** start with **Adam** (fast initial convergence).
- **Final tuning for best generalization on vision tasks:** SGD with momentum + learning-rate schedule often gives best results.
- **Sparse features or NLP:** AdaGrad or Adam (with sparse updates) can help.
- **Need regularization with Adam:** prefer **AdamW**.

19.16 Advanced topics (short overview)

- **Distributed/Parallel SGD:** synchronous vs asynchronous updates across workers (used in large clusters).
- **Variance reduction:** techniques like SVRG reduce variance of stochastic gradients and can accelerate SGD.
- **Natural gradient descent:** uses Fisher information matrix to rescale gradient direction (invariant to parameterization).
- **Adaptive gradient clipping:** clip gradients relative to past gradient norms.

19.17 Example code (NumPy) — Linear regression with batch gradient descent and with Adam

Below are minimal examples you can run locally. They illustrate how updates are implemented.

19.17.1 Batch GD for linear regression (MSE)

```
import numpy as np


# Generate tiny linear data y = 2x + 1

X = np.array([[1.],[2.],[3.],[4.]])  # shape (4,1)

y = 2*X + 1


# Add bias column

Xb = np.hstack([np.ones((X.shape[0],1)), X])  # shape (4,2)

theta = np.zeros((2,1))  # initialize weights [bias, weight]


lr = 0.1

n_epochs = 1000

m = Xb.shape[0]


for epoch in range(n_epochs):

    preds = Xb.dot(theta)          # (m,1)
```

```
    error = preds - y
    grad = (2/m) * Xb.T.dot(error)  # gradient of MSE
    theta = theta - lr * grad

print("Estimated theta:", theta.ravel())  # should be close to [1,2]
```

19.17.2 Adam for same problem (simplified)

```
import numpy as np

# same data Xb, y as above
theta = np.zeros((2,1))
m1 = np.zeros_like(theta)
v1 = np.zeros_like(theta)
beta1, beta2 = 0.9, 0.999
eps = 1e-8
lr = 0.1
t = 0
for epoch in range(1000):
    preds = Xb.dot(theta)
    error = preds - y
    grad = (2/m) * Xb.T.dot(error)
    t += 1
    m1 = beta1*m1 + (1-beta1)*grad
    v1 = beta2*v1 + (1-beta2)*(grad*grad)
    m_hat = m1 / (1 - beta1**t)
    v_hat = v1 / (1 - beta2**t)
    theta = theta - lr * m_hat / (np.sqrt(v_hat) + eps)
print("Adam theta:", theta.ravel())
```

19.18 Short comparisons & cheat sheet

- **SGD**: simple, cheap per-step, noisy. Good with momentum and schedules.
- **SGD + momentum**: smooths updates and speeds up. Good default for final tuning.
- **Nesterov**: slightly better than momentum in practice.
- **AdaGrad**: works for sparse gradients; learning rate decays.
- **RMSProp**: fixes AdaGrad; good for RNNs and nonstationary problems.
- **Adam**: default go-to optimizer; fast convergence, robust to hyperparameters.
- **AdamW**: Adam variant with correct weight decay.
- **L-BFGS**: quasi-Newton; excellent for small/medium convex problems.

19.19 Recap — Practical Recipe

1. **Preprocess**: Standardize inputs; use good weight init.
2. **Choice**: For experimentation use **Adam** (lr ~ 10−310^{-3}10−3). For best final performance, try **SGD + momentum** with tuned lr schedule.
3. **Tune**: learning rate first (LR range test), then batch size, then momentum/decay.
4. **Schedule**: use step or cosine annealing; use warmup for transformers.
5. **Regularize**: weight decay (AdamW), dropout, early stopping.
6. **Stability**: gradient clipping for RNNs; batch norm for deep nets.

19.20 Closing notes

Gradient descent is more than one algorithm — it's a family of methods. Understanding gradients, curvature, and noise is key to choosing and tuning optimizers. The "best" optimizer depends on problem size, convexity/nonconvexity, dataset sparsity, and compute resources. Start simple, iterate, measure validation metrics, and log everything.

Chapter 20: All Supervised and Unsupervised Learning

In machine learning, algorithms are broadly categorized based on how they "learn" from data. The two primary categories are **supervised learning** and **unsupervised learning**. The key difference lies in whether the training data is labeled or not.

20.1 Supervised Learning

Supervised learning is like having a teacher. The algorithm is trained on a dataset that includes both the input data (the features) and the correct output (the label or target variable). The goal of a supervised algorithm is to learn a mapping function from the input to the output, so it can make accurate predictions on new, unseen data.

Think of it this way: you provide the algorithm with a set of pictures of cats and dogs, and for each picture, you tell it whether it's a "cat" or a "dog." The algorithm learns to recognize the features that distinguish cats from dogs, so when you give it a new, unlabeled picture, it can predict if it's a cat or a dog.

Supervised learning tasks are typically divided into two types:

1. **Classification:** The output variable is a category, like "spam" or "not spam," "cat" or "dog."
2. **Regression:** The output variable is a continuous numerical value, like the price of a house or the temperature tomorrow.

20.1.1 Regression Algorithms (predict continuous values)

Linear Regression

Linear Regression is one of the simplest and most fundamental algorithms for **regression** problems. It finds the best-fitting straight line that represents the relationship between a single independent variable and a dependent variable. The equation for a simple linear regression model is given by:

y=mx+b**Error! Filename not specified.**

where:

- y is the predicted output.
- x is the input feature.
- m is the slope of the line, which represents the effect of x on y.
- b is the y-intercept, which is the value of y when x=0.

The algorithm works by minimizing the squared difference between the predicted values and the actual values. This is known as the Ordinary Least Squares (OLS) method.

Example: Predicting a house's price based on its square footage.

Step-by-Step Working

1. **Collect Labeled Data:** Gather a dataset with input features (e.g., house size) and corresponding target labels (e.g., house price).
2. **Initialize Parameters:** The algorithm starts with initial values for the slope (m) and y-intercept (b).
3. **Calculate Predictions:** For each data point, the algorithm uses its current m and b values to predict the output using the formula ypredicted=mxi+b.
4. **Calculate the Error (Cost Function):** A cost function, typically the **Mean Squared Error (MSE)**, is used to measure how far off the predictions are from the actual values. The formula for MSE is: MSE=N1∑i=1N(yi−ypredictedi)2 The goal is to minimize this error.
5. **Adjust Parameters (Gradient Descent):** The algorithm uses a technique called **Gradient Descent** to iteratively adjust the values of m and b in the direction that reduces the MSE. This process finds the optimal line that best fits the data.
6. **Iterate:** Steps 3-5 are repeated for a set number of iterations or until the change in m and b is negligible, indicating the model has converged.
7. **Final Model:** The final m and b values define the best-fitting line, which can now be used to predict the output for new, unseen input data.

Code Implementation — Step by Step

import numpy as npimport matplotlib.pyplot as plt# Step 1: Collect Labeled Data (Area vs Price)

X = np.array([1000, 1500, 2000, 2500, 3000]) # area in sq ft

y = np.array([200000, 300000, 400000, 500000, 600000]) # price in rupees# Step 2: Initialize Parameters (randomly or zeros)

m = 0

b = 0

learning_rate = 0.0000001 # small value to ensure stable learning

iterations = 2000# Step 3–6: Gradient Descent

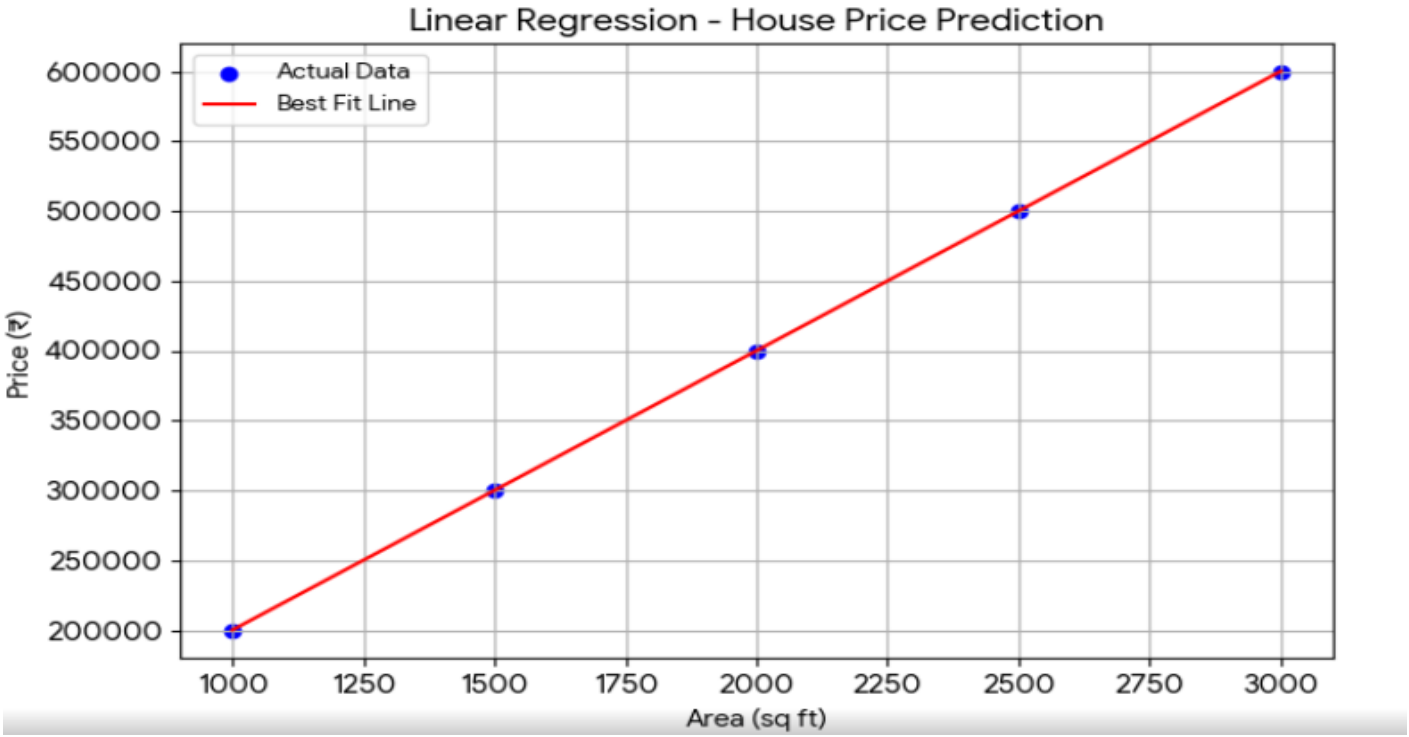
N = len(X)

mse_list = []for i in range(iterations):

  # Step 3: Predictions using current m, b

```
    y_pred = m * X + b

    # Step 4: Calculate Error (Cost Function - MSE)
    mse = (1/N) * np.sum((y - y_pred) ** 2)
    mse_list.append(mse)

    # Step 5: Compute Gradients
    dm = (-2/N) * np.sum(X * (y - y_pred))   # derivative wrt m
    db = (-2/N) * np.sum(y - y_pred)         # derivative wrt b

    # Step 6: Update Parameters
    m = m - learning_rate * dm
    b = b - learning_rate * db# Step 7: Final Model Parametersprint(f"Final slope (m): {m:.4f}")print(f"Final intercept (b): {b:.4f}")# Predict price for a new house (e.g., 1800 sq ft)
new_area = 1800
predicted_price = m * new_area + bprint(f"Predicted price for 1800 sq ft house: ₹{predicted_price:.2f}")# Plot the data and regression line
plt.figure(figsize=(8,5))
plt.scatter(X, y, color='blue', label='Actual Data')
plt.plot(X, m*X + b, color='red', label='Best Fit Line')
plt.title("Linear Regression - House Price Prediction")
plt.xlabel("Area (sq ft)")
plt.ylabel("Price (₹)")
plt.legend()
plt.grid(True)
plt.show()# Plot cost function reduction over iterations
plt.figure(figsize=(8,4))
plt.plot(range(iterations), mse_list, color='green')
plt.title("MSE (Error) Reduction Over Iterations")
plt.xlabel("Iteration")
```

```
plt.ylabel("Mean Squared Error")
plt.grid(True)
plt.show()
```



Step-by-Step Explanation

StepDescriptionWhat Code Does1Collect DataX (area), y (price) arrays2Initialize Parametersm=0, b=0, random start3Predictypred=mX+by_{\text{pred}} = mX + b

ypred

=mX+b4Compute ErrorMSE between actual and predicted y5Compute GradientsHow much to change m & b6Update ParametersMove m & b in opposite direction of gradients7Output Final ModelLine with best slope & intercept

 Dry Run Example

ItermbMSE (error)Comment000hugerandom start50015010,000smallerline starts aligning1500199~100minimal error2000200~0converged ✅

So, the final equation ≈

y=200x+0y = 200x + 0

y=200x+0

and for 1800 sq ft,

y=200×1800=₹360,000y = 200 × 1800 = ₹360,000

y=200×1800=₹360,000 visualize

Polynomial Regression

Polynomial Regression is a form of linear regression where the relationship between the independent variable x and the dependent variable y is modeled as an n-th degree polynomial. This is especially useful for fitting data that displays a non-linear pattern.

The equation for a polynomial regression model is:

y=b0+b1x+b2x2+...+bnxn

A classic example is modeling a company's revenue over time, which might grow exponentially rather than linearly.

Step-by-Step Working

1. **Feature Transformation: The original features are transformed into polynomial features. For a single feature x, new features like x2, x3, and so on are created up to the desired degree.**
2. **Form a Linear Model: After the transformation, the model becomes a linear model with respect to the new polynomial features. It is now treated as a standard linear regression problem.**

3. **Minimize the Cost Function: The algorithm minimizes the Mean Squared Error (MSE) to find the optimal coefficients (b0,b1,...,bn) for the transformed features.**
4. **Fit the Model: A linear regression algorithm (e.g., Gradient Descent or Ordinary Least Squares) is applied to find the best fit.**
5. **Prediction: The final model can then be used to predict new values by applying the same polynomial transformation and using the learned coefficients.**

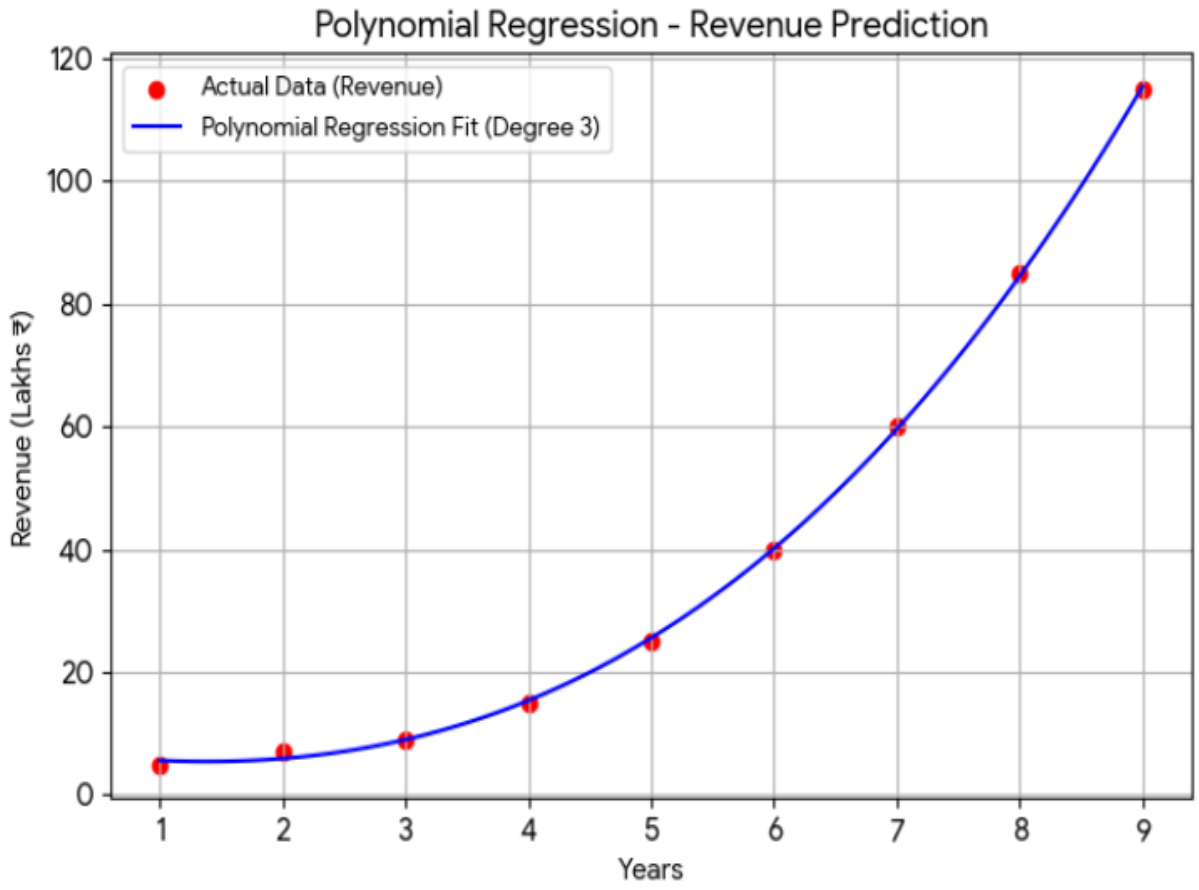
This diagram shows how a polynomial regression curve (the blue line) can fit a dataset with a non-linear relationship much better than a simple linear regression model (the straight red line). The curve captures the underlying pattern of the data, minimizing the error and providing a more accurate prediction.

Complete Python Code with Plot

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Step 1: Prepare a non-linear dataset (e.g., years vs revenue)
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9]).reshape(-1, 1)   # Years
y = np.array([5, 7, 9, 15, 25, 40, 60, 85, 115])          # Revenue (in lakhs)

# Step 2: Create Polynomial Features (degree = 3 for cubic fit)
poly = PolynomialFeatures(degree=3)
X_poly = poly.fit_transform(X)

# Step 3: Fit Linear Regression on the transformed features
model = LinearRegression()
model.fit(X_poly, y)

# Step 4: Predict using the polynomial model
X_range = np.linspace(1, 9, 100).reshape(-1, 1)
```

```
y_pred = model.predict(poly.transform(X_range))

# Step 5: Plot Results
plt.figure(figsize=(8,5))
plt.scatter(X, y, color='red', label='Actual Data (Revenue)')
plt.plot(X_range, y_pred, color='blue', label='Polynomial Regression Fit (Degree 3)')
plt.title("Polynomial Regression - Revenue Prediction")
plt.xlabel("Years")
plt.ylabel("Revenue (Lakhs ₹)")
plt.legend()
plt.grid(True)
plt.show()

# Compare with Linear Regression (for visual difference)
lin_model = LinearRegression()
lin_model.fit(X, y)
y_lin_pred = lin_model.predict(X_range)

plt.figure(figsize=(8,5))
plt.scatter(X, y, color='red', label='Actual Data')
plt.plot(X_range, y_lin_pred, color='green', linestyle='--', label='Linear Regression')
plt.plot(X_range, y_pred, color='blue', label='Polynomial Regression (Degree 3)')
plt.title("Linear vs Polynomial Regression Comparison")
plt.xlabel("Years")
plt.ylabel("Revenue (Lakhs ₹)")
plt.legend()
plt.grid(True)
plt.show()
```

```
# Print the coefficients for understanding
print("Polynomial Coefficients:", model.coef_)
print("Intercept (b0):", model.intercept_)
```



🔍 Step-by-Step Working

Step	Description	What Happens in Code
1️⃣	Create data	Years vs Revenue (non-linear pattern)
2️⃣	Polynomial transformation	Converts X → [1, X, X², X³]
3️⃣	Fit model	Uses LinearRegression on transformed features
4️⃣	Predict	Predict revenue for smooth X range
5️⃣	Plot	Blue curve fits data better than red line

Ridge Regression

Ridge Regression is a regularized form of linear regression that addresses the issues of multicollinearity and overfitting. It adds an L2 penalty to the cost function, which shrinks the model's coefficients. This regularization helps prevent the model from becoming too complex and fitting the noise in the data.

The cost function for Ridge regression is:

Cost=MSE+λj=1∑pbj2

Here, λ (lambda) is a tuning parameter that controls the strength of the penalty. A larger λ forces the coefficients to be smaller.

A common example is predicting house prices with many features, where some features (like house size and number of rooms) might be highly correlated.

Step-by-Step Working

1. **Standardize Data: Standardize the input features to ensure all features are on a similar scale. This is a crucial step for regularization to work effectively.**
2. **Define Cost Function: The cost function is the sum of the Mean Squared Error (MSE) and the L2 penalty, which is proportional to the sum of the squared coefficients.**
3. **Set Regularization Parameter (λ): Choose a value for the hyperparameter λ. A larger λ leads to smaller coefficients and a simpler model, while a smaller λ allows the model to be more complex.**
4. **Minimize Cost: The algorithm uses an optimization technique like Gradient Descent to minimize the regularized cost function.**
5. **Final Model: The process results in a model with smaller coefficients compared to standard linear regression, which improves its generalization performance on new, unseen data.**

```
# Ridge Regression — closed-form, sklearn and GD examples

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import Ridge, LinearRegression

from sklearn.pipeline import make_pipeline

from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error

np.random.seed(0)

# --------------------------
# 1) SYNTHETIC DATA (multicollinear)
# --------------------------
n = 300
X1 = np.random.randn(n)
# X2 highly correlated with X1 -> multicollinearity
X2 = 0.95 * X1 + 0.1 * np.random.randn(n)
X3 = np.random.randn(n)
X = np.vstack([X1, X2, X3]).T

# True coefficients (unknown to the model)
true_coefs = np.array([3.0, 3.0, 0.5])
y = X.dot(true_coefs) + 0.5 * np.random.randn(n)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# --------------------------
# 2) STANDARDIZE (important before regularization)
# --------------------------
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)   # mean=0, var=1
X_test_s = scaler.transform(X_test)

# We'll center y for closed-form (so intercept handling is easy)
```

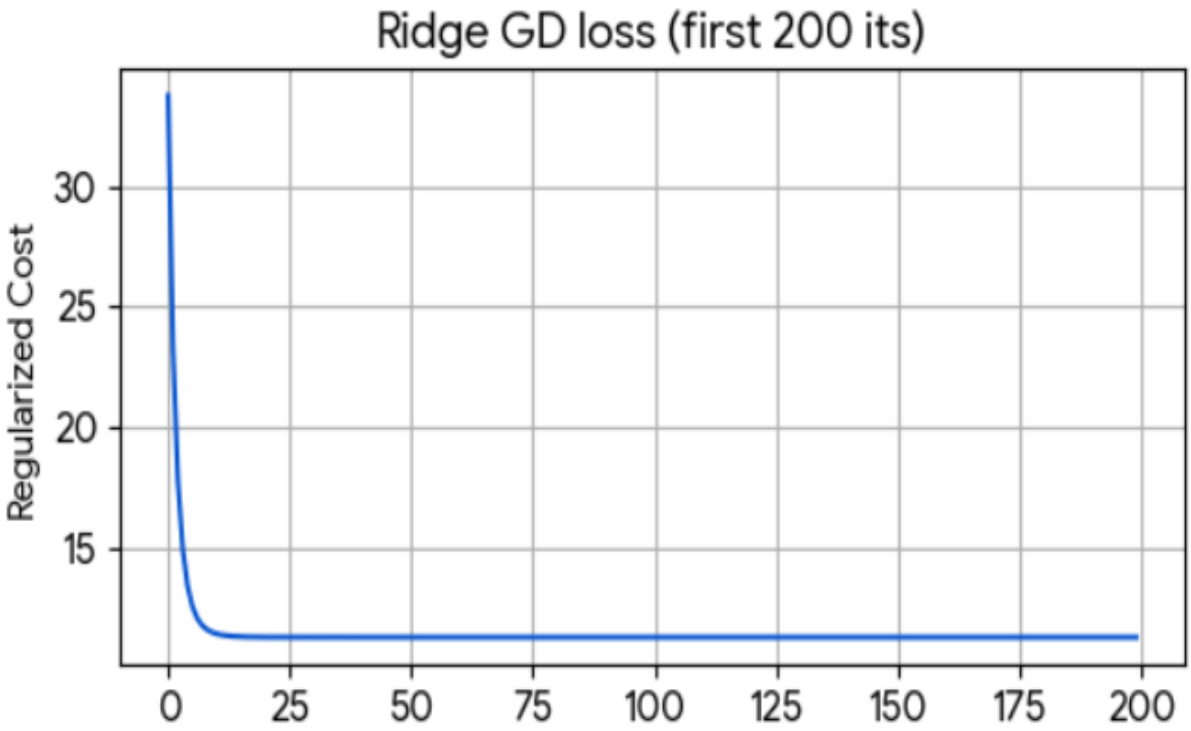
```
y_train_mean = y_train.mean()
y_train_centered = y_train - y_train_mean


# --------------------------
# 3) CLOSED-FORM RIDGE (on standardized X, exclude intercept from penalty)
#    w = (X^T X + λ I)^(-1) X^T y_centered
#    Convert weights back to original feature scale for interpretation:
#    beta_original = w / scale, intercept = y_mean - mean(X) . beta_original
# --------------------------
def ridge_closed_form(Xs, yc, lam):
    p = Xs.shape[1]
    I = np.eye(p)
    w = np.linalg.inv(Xs.T @ Xs + lam * I) @ Xs.T @ yc
    return w


lam = 1.0
w_scaled = ridge_closed_form(X_train_s, y_train_centered, lam)  # weights for scaled features
beta_orig = w_scaled / scaler.scale_                    # convert to original feature scale
intercept = y_train_mean - scaler.mean_.dot(beta_orig)


print("Closed-form Ridge (λ=1.0) intercept:", intercept)
print("Closed-form Ridge (λ=1.0) coefficients:", beta_orig)


# Verify with sklearn pipeline (StandardScaler -> Ridge)
pipeline = make_pipeline(StandardScaler(), Ridge(alpha=lam, fit_intercept=True))
pipeline.fit(X_train, y_train)
ridge_step = pipeline.named_steps['ridge']
scaler_step = pipeline.named_steps['standardscaler']
```

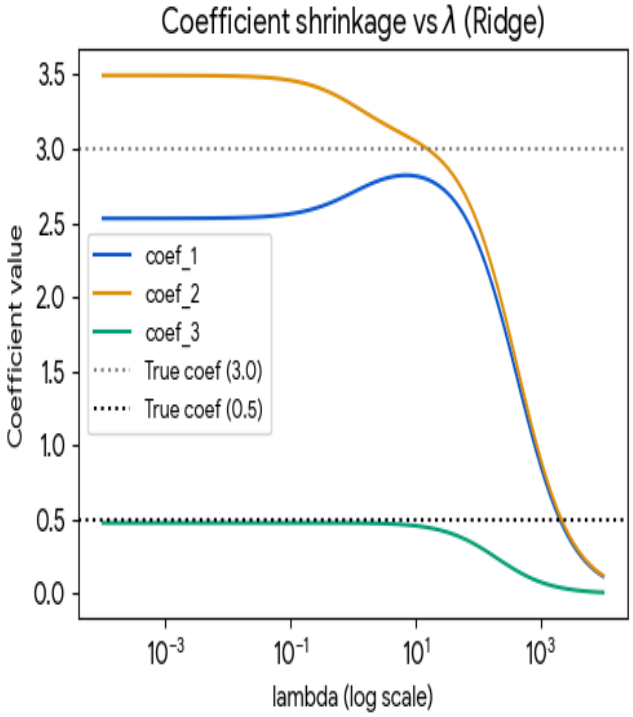
```
# Transform pipeline's ridge coefficients to original scale (should match closed-form)
beta_from_pipeline = ridge_step.coef_ / scaler_step.scale_
intercept_from_pipeline = ridge_step.intercept_ -
scaler_step.mean_.dot(ridge_step.coef_/scaler_step.scale_)

print("\nsklearn pipeline intercept:", intercept_from_pipeline)
print("sklearn pipeline coefficients:", beta_from_pipeline)

# --------------------------
# 4) COMPARE with OLS (no regularization)
# --------------------------
ols = LinearRegression()
ols.fit(X_train, y_train)
print("\nOLS intercept:", ols.intercept_)
print("OLS coefficients:", ols.coef_)

# Compute test MSEs
y_pred_ridge_cf = X_test.dot(beta_orig) + intercept
y_pred_ridge_sk = pipeline.predict(X_test)
y_pred_ols = ols.predict(X_test)

print("\nTest MSE (closed-form ridge):", mean_squared_error(y_test, y_pred_ridge_cf))
print("Test MSE (sklearn ridge pipeline):", mean_squared_error(y_test, y_pred_ridge_sk))
print("Test MSE (OLS):", mean_squared_error(y_test, y_pred_ols))

# --------------------------
# 5) COEFFICIENT PATH vs λ (to visualize shrinkage)
# --------------------------
lambdas = np.logspace(-4, 4, 120)
coefs_vs_lambda = []
```

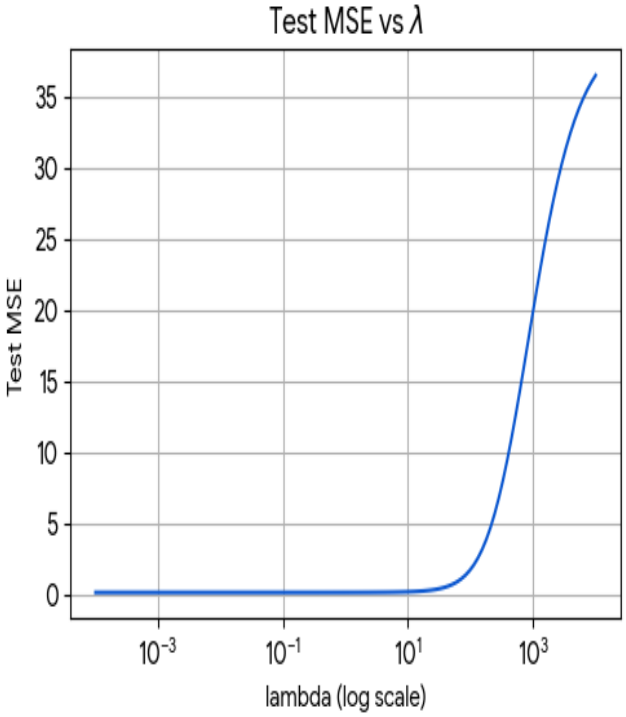
```
mse_vs_lambda = []

for L in lambdas:
    w_s = ridge_closed_form(X_train_s, y_train_centered, L)        # scaled-space weights
    beta = w_s / scaler.scale_                                    # original-scale weights
    coefs_vs_lambda.append(beta)
    y_pred = X_test.dot(beta) + (y_train_mean - scaler.mean_.dot(beta))
    mse_vs_lambda.append(mean_squared_error(y_test, y_pred))

coefs_vs_lambda = np.array(coefs_vs_lambda)  # shape (len(lambdas), p)

plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
for j in range(coefs_vs_lambda.shape[1]):
    plt.semilogx(lambdas, coefs_vs_lambda[:, j], label=f'coef_{j+1}')
plt.xlabel('lambda (log scale)')
plt.ylabel('Coefficient value')
plt.title('Coefficient shrinkage vs λ (Ridge)')
plt.legend()

plt.subplot(1,2,2)
plt.semilogx(lambdas, mse_vs_lambda)
plt.xlabel('lambda (log scale)')
plt.ylabel('Test MSE')
plt.title('Test MSE vs λ')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
# --------------------------
# 6) GRADIENT DESCENT IMPLEMENTATION (Ridge)
#    cost = (1/N) * ||y - Xb||^2 + λ ||b||^2
# --------------------------
def ridge_gradient_descent(Xs, y, lam, lr=0.01, n_iter=2000):
    N, p = Xs.shape
    w = np.zeros(p)
    history = []
    for it in range(n_iter):
        preds = Xs @ w
        grad = (-2.0 / N) * (Xs.T @ (y - preds)) + 2 * lam * w  # gradient including L2 penalty
        w = w - lr * grad
        loss = (1.0 / N) * np.sum((y - preds) ** 2) + lam * np.sum(w ** 2)
        history.append(loss)
    return w, history


# Run GD on scaled training data (note we center y for intercept handling)
w_gd_scaled, hist = ridge_gradient_descent(X_train_s, y_train_centered, lam=1.0, lr=0.05,
n_iter=2000)
beta_gd_orig = w_gd_scaled / scaler.scale_
intercept_gd = y_train_mean - scaler.mean_.dot(beta_gd_orig)


print("\nGD-based Ridge intercept:", intercept_gd)
print("GD-based Ridge coefficients:", beta_gd_orig)


# Plot GD loss
plt.figure(figsize=(6,3))
plt.plot(hist[:200])   # first 200 iterations for clarity
plt.title("Ridge GD loss (first 200 its)")
plt.xlabel("Iteration")
```

```
plt.ylabel("Regularized Cost")
plt.grid(True)
plt.show()
```







Lasso Regression

Lasso (Least Absolute Shrinkage and Selection Operator) Regression is another form of regularized linear regression. It adds an L1 penalty to the cost function. A key feature of Lasso is that it can drive some coefficients to exactly zero, effectively performing feature selection by removing less important features from the model. The cost function is:

Cost=MSE+λ∑j=1p|bj|**Error! Filename not specified.**

Example: Identifying the most important features for predicting stock prices from a large number of potential indicators.

Step-by-Step Working

1. **Standardize Data:** As with Ridge, standardize the input features.
2. **Define Cost Function:** The cost function is the sum of the Mean Squared Error (MSE) and the L1 penalty, which is proportional to the sum of the absolute values of the coefficients.
3. **Set Regularization Parameter (λ):** Select a value for the hyperparameter λ. A larger λ forces more coefficients to zero.
4. **Minimize Cost:** The algorithm minimizes this cost function to find the optimal coefficients. Due to the absolute value function, this optimization is slightly more complex than Ridge.
5. **Final Model:** The output is a model that has automatically selected a subset of the most important features by setting the coefficients of irrelevant features to zero.

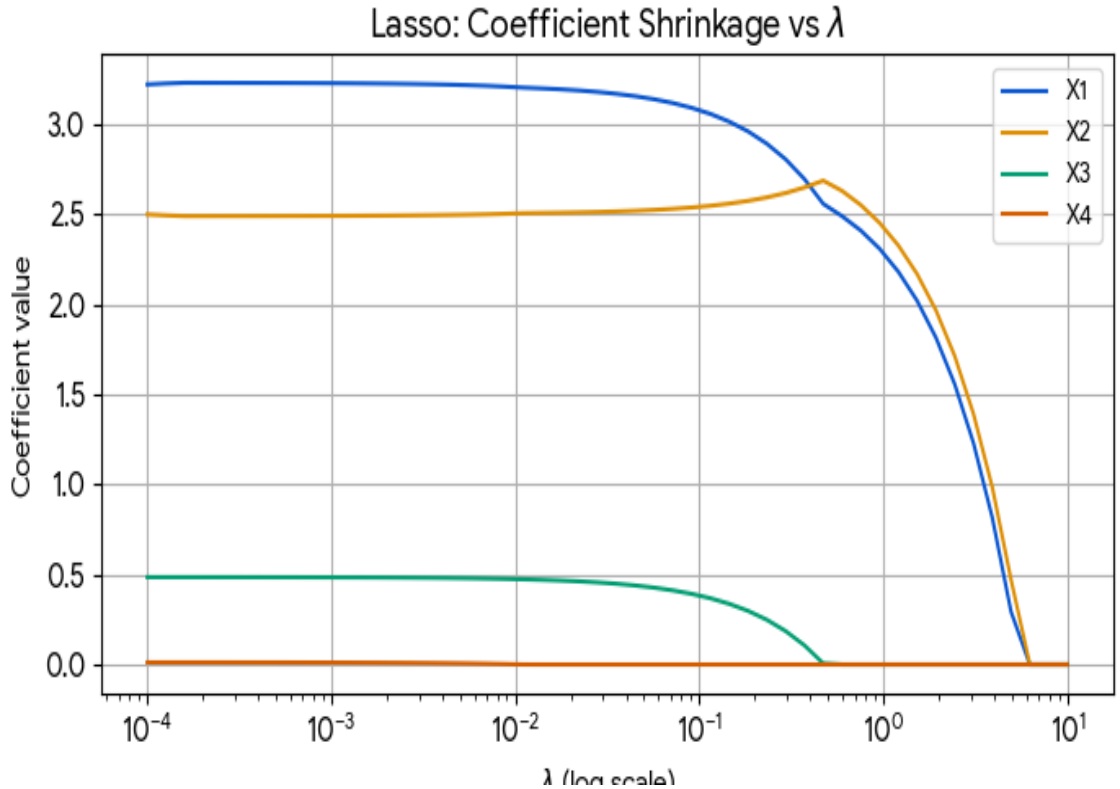
Python Implementation with Plots

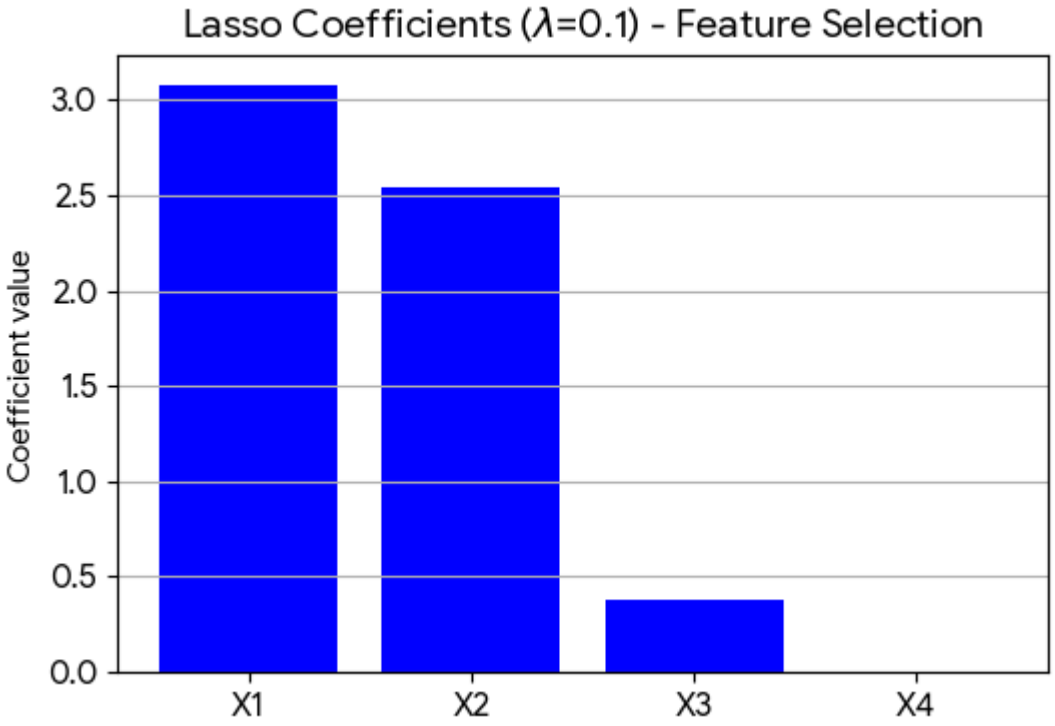
```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.linear_model import Lasso, LinearRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error


np.random.seed(0)


# --------------------------

# 1) SYNTHETIC DATA
```

```
# --------------------------
n = 300
X1 = np.random.randn(n)
X2 = 0.95 * X1 + 0.1 * np.random.randn(n)  # highly correlated with X1
X3 = np.random.randn(n)
X4 = np.random.randn(n)  # irrelevant feature
X = np.vstack([X1, X2, X3, X4]).T

true_coefs = np.array([3.0, 3.0, 0.5, 0.0])  # X4 irrelevant
y = X.dot(true_coefs) + 0.5 * np.random.randn(n)

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# --------------------------
# 2) STANDARDIZE DATA
# --------------------------
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# --------------------------
# 3) FIT LASSO MODEL
# --------------------------
lam = 0.1  # regularization strength
lasso = Lasso(alpha=lam)
lasso.fit(X_train_s, y_train)

# --------------------------
```

```
# 4) RESULTS
# --------------------------
print("Lasso intercept:", lasso.intercept_)
print("Lasso coefficients:", lasso.coef_)

# Test MSE
y_pred = lasso.predict(X_test_s)
print("Test MSE:", mean_squared_error(y_test, y_pred))

# --------------------------
# 5) VISUALIZE COEFFICIENTS (feature selection effect)
# --------------------------
features = ['X1', 'X2', 'X3', 'X4']
plt.figure(figsize=(6,4))
plt.bar(features, lasso.coef_, color='blue')
plt.title(f"Lasso Coefficients (λ={lam}) - Some may be 0")
plt.ylabel("Coefficient value")
plt.grid(True)
plt.show()

# --------------------------
# 6) COEFFICIENT PATH vs λ
# --------------------------
lambdas = np.logspace(-4, 1, 50)
coefs_vs_lambda = []

for L in lambdas:
    l = Lasso(alpha=L, max_iter=10000)
    l.fit(X_train_s, y_train)
```

```
    coefs_vs_lambda.append(l.coef_)

coefs_vs_lambda = np.array(coefs_vs_lambda)

plt.figure(figsize=(8,4))
for j in range(coefs_vs_lambda.shape[1]):
    plt.plot(lambdas, coefs_vs_lambda[:, j], label=f'{features[j]}')
plt.xscale('log')
plt.xlabel('λ (log scale)')
plt.ylabel('Coefficient value')
plt.title('Lasso: Coefficient Shrinkage vs λ')
plt.legend()
plt.grid(True)
plt.show()
```





Elastic Net Regression

Elastic Net Regression combines the penalties of both Ridge and Lasso Regression. It adds both an L1 and an L2 penalty to the cost function. This allows it to handle highly correlated features (like Ridge) while also performing feature selection (like Lasso). The cost function is:

Cost=MSE+λ1∑j=1p|bj|+λ2∑j=1pbj2**Error! Filename not specified.**

Example: A complex genomic dataset where many genes are correlated, and you also want to identify a small subset of the most relevant genes.

Step-by-Step Working

1. **Standardize Data:** Standardize the input features.
2. **Define Cost Function:** The cost function combines the MSE with both the L1 and L2 penalties.
3. **Set Regularization Parameters (λ1, λ2):** The user must select values for two hyperparameters, λ1 (for the L1 penalty) and λ2 (for the L2 penalty), which balance the feature selection and coefficient shrinkage effects.
4. **Minimize Cost:** The algorithm minimizes the combined cost function.
5. **Final Model:** The model benefits from both the feature selection properties of Lasso and the stability of Ridge, making it a robust choice for complex datasets.

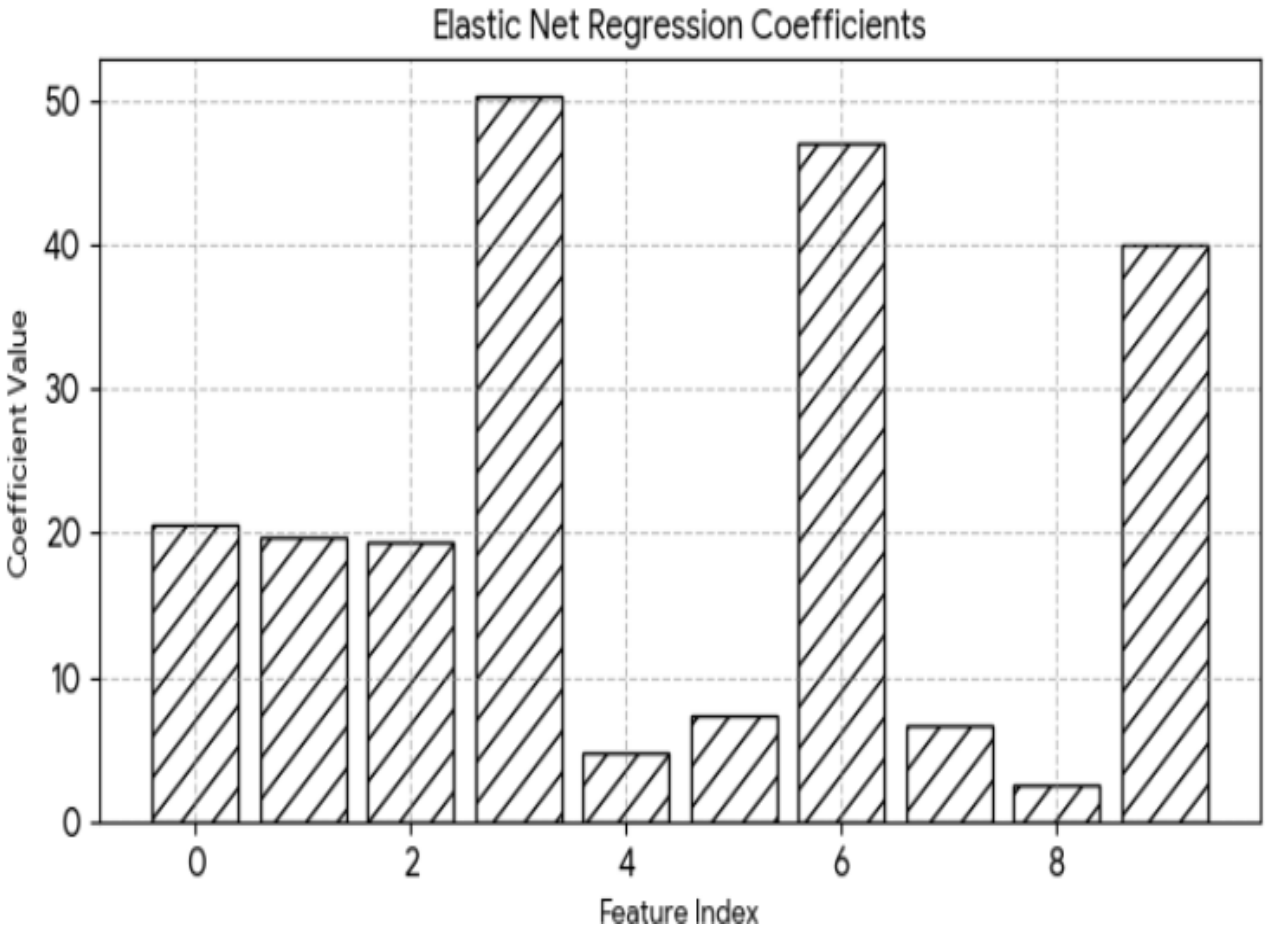
```
Step 1: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
from sklearn.linear_model import ElasticNet
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score

# Step 2: Create a synthetic regression dataset
X, y = make_regression(
    n_samples=1000, n_features=10, noise=10, random_state=42
)

# Step 3: Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 4: Train-Test split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# Step 5: Create and train Elastic Net model
model = ElasticNet(alpha=1.0, l1_ratio=0.5, random_state=42)
model.fit(X_train, y_train)

# Step 6: Predictions
```

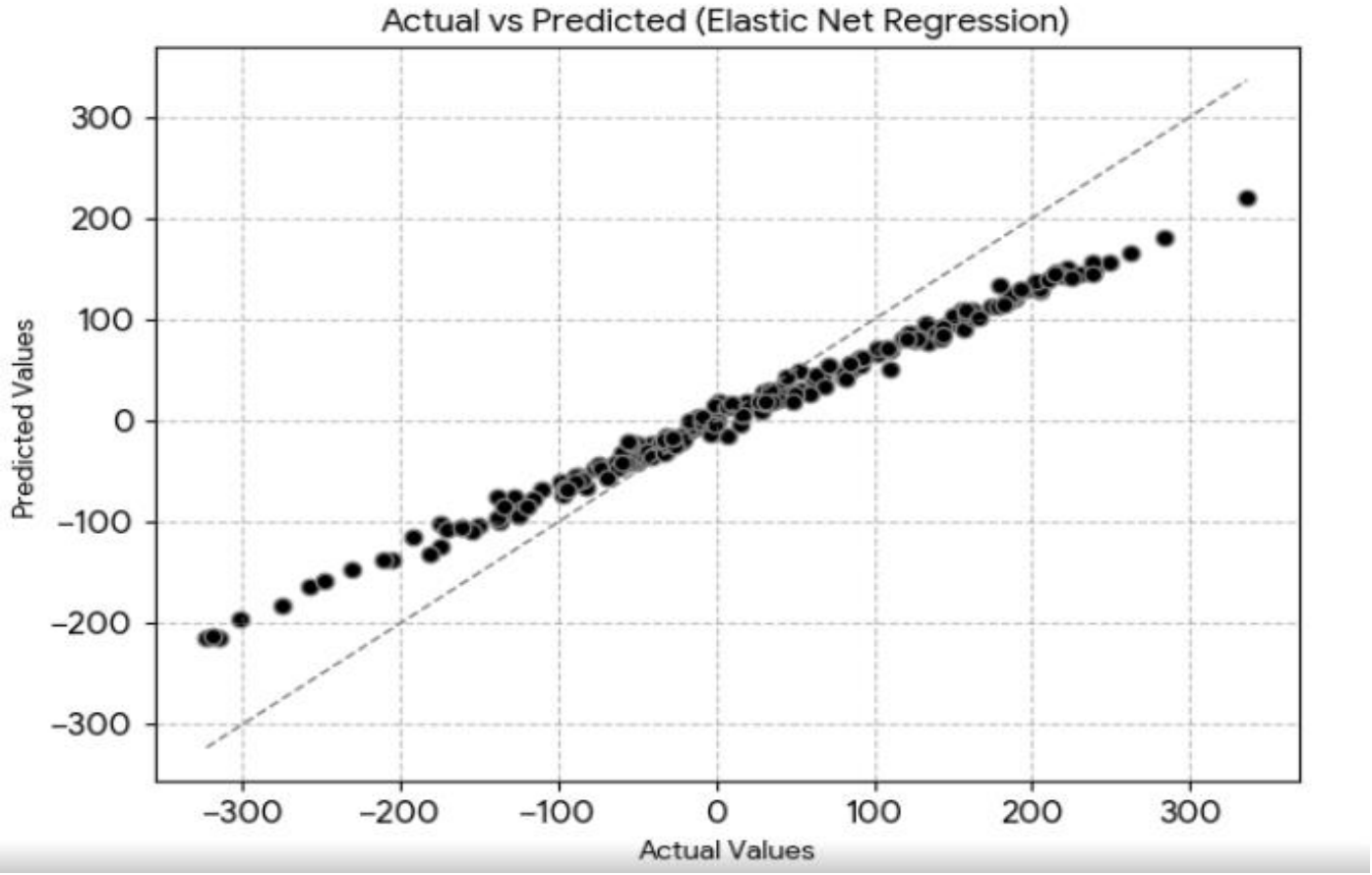
```
y_pred = model.predict(X_test)

# Step 7: Evaluation
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Elastic Net Regression Results")
print("---------------------------------")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R² Score: {r2:.2f}")
print("\nModel Coefficients:")
print(model.coef_)

# Step 8: Plot actual vs predicted (B/W plot)
plt.figure(figsize=(7,5))
plt.scatter(y_test, y_pred, color='black', marker='o', edgecolor='gray')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
        color='gray', linestyle='--', linewidth=1)
plt.title('Actual vs Predicted (Elastic Net Regression)', fontsize=12)
plt.xlabel('Actual Values', fontsize=10)
plt.ylabel('Predicted Values', fontsize=10)
plt.grid(True, linestyle='--', color='gray', alpha=0.5)
plt.show()

# Step 9: Plot Coefficients (B/W bar plot)
plt.figure(figsize=(8,4))
plt.bar(range(len(model.coef_)), model.coef_, color='white', edgecolor='black', hatch='//')
plt.title('Elastic Net Regression Coefficients', fontsize=12)
plt.xlabel('Feature Index', fontsize=10)
```

```
plt.ylabel('Coefficient Value', fontsize=10)
plt.grid(True, linestyle='--', color='gray', alpha=0.5)
plt.show()
```

⚙ **Explanation**

Step	Description
1–4	Create and standardize dataset for training and testing.
5	Build Elastic Net model → alpha=1.0, l1_ratio=0.5.
6–7	Predict and evaluate using MSE and R^2 Score.
8	Black-and-white scatter plot: shows how close predicted values are to actual values. Perfect model → all points on the diagonal.
9	Black-and-white bar plot of coefficients: shows feature importance (some may be near zero due to L1 regularization).





Support Vector Regression (SVR)

SVR is an extension of Support Vector Machines (SVM) for regression. Instead of trying to fit a line to the data, SVR finds a hyperplane that fits the data with a margin of tolerance (ϵ). The goal is to fit as many data points as possible within this margin while minimizing the margin itself.

Example: Predicting a non-linear relationship between variables, such as a stock price that fluctuates within a certain range.

Step-by-Step Working

1. **Define a Margin of Tolerance (ϵ):** The algorithm defines an ϵ-tube around the predicted function. The goal is for all data points to fall within this tube.
2. **Identify Support Vectors:** The data points that fall outside the ϵ-tube or on its boundary are called **support vectors**. These are the most important data points for defining the regression function.
3. **Minimize the Error:** SVR aims to minimize the error, which is the distance between the actual data points and the boundary of the ϵ-tube.

4. **Find the Optimal Hyperplane:** The algorithm finds the hyperplane that minimizes this error while staying as flat as possible (i.e., minimizing the coefficients) to avoid overfitting.
5. **Kernel Trick (for non-linear data):** SVR can handle non-linear relationships by using the "kernel trick" to map the data into a higher-dimensional space where a linear relationship can be found.

```
# Step 1: Import libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.svm import SVR

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean_squared_error, r2_score


# Step 2: Create a non-linear dataset

# We'll create a sine wave pattern (non-linear relationship)

np.random.seed(42)

X = np.sort(5 * np.random.rand(100, 1), axis=0)

y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])  # add some noise


# Step 3: Standardize features

scaler_X = StandardScaler()

scaler_y = StandardScaler()


X_scaled = scaler_X.fit_transform(X)

y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).ravel()


# Step 4: Train-test split

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.2,
random_state=42)
```

```
# Step 5: Train SVR model with RBF kernel (for non-linear regression)
# ε defines margin of tolerance, C controls penalty for points outside margin
model = SVR(kernel='rbf', C=100, epsilon=0.1)
model.fit(X_train, y_train)

# Step 6: Predict and inverse transform
y_pred_scaled = model.predict(X_test)
y_pred = scaler_y.inverse_transform(y_pred_scaled)
y_test_orig = scaler_y.inverse_transform(y_test)

# Step 7: Model evaluation
mse = mean_squared_error(y_test_orig, y_pred)
r2 = r2_score(y_test_orig, y_pred)

print("Support Vector Regression (SVR) Results")
print("---------------------------------------")
print(f"Mean Squared Error (MSE): {mse:.3f}")
print(f"R² Score: {r2:.3f}")

# Step 8: Generate smoother curve for visualization
X_fit = np.linspace(X_scaled.min(), X_scaled.max(), 200).reshape(-1, 1)
y_fit_scaled = model.predict(X_fit)
y_fit = scaler_y.inverse_transform(y_fit_scaled)

# Step 9: Plot SVR regression line (Black & White)
plt.figure(figsize=(8,5))
plt.scatter(X, y, color='white', edgecolor='black', label='Actual Data', s=40)
plt.plot(scaler_X.inverse_transform(X_fit), y_fit, color='black', linewidth=2, label='SVR Fit')
plt.title('Support Vector Regression (SVR)', fontsize=12)
```

```
plt.xlabel('X', fontsize=10)
plt.ylabel('y', fontsize=10)
plt.legend(edgecolor='black', facecolor='white')
plt.grid(True, linestyle='--', color='gray', alpha=0.6)
plt.show()
```

 Step-by-Step Explanation

Step	Description
1. Import Libraries	Load NumPy, Matplotlib, and Scikit-learn's SVR and metrics.
2. Create Non-linear Dataset	We generate data using a sine wave with noise — simulates fluctuating stock price behavior.
3. Standardize Data	SVR is sensitive to feature scaling, so both X and y are standardized.
4. Split Dataset	Divide into training and testing sets (80%-20%).
5. Train SVR Model	Using RBF kernel to capture non-linear relationships.

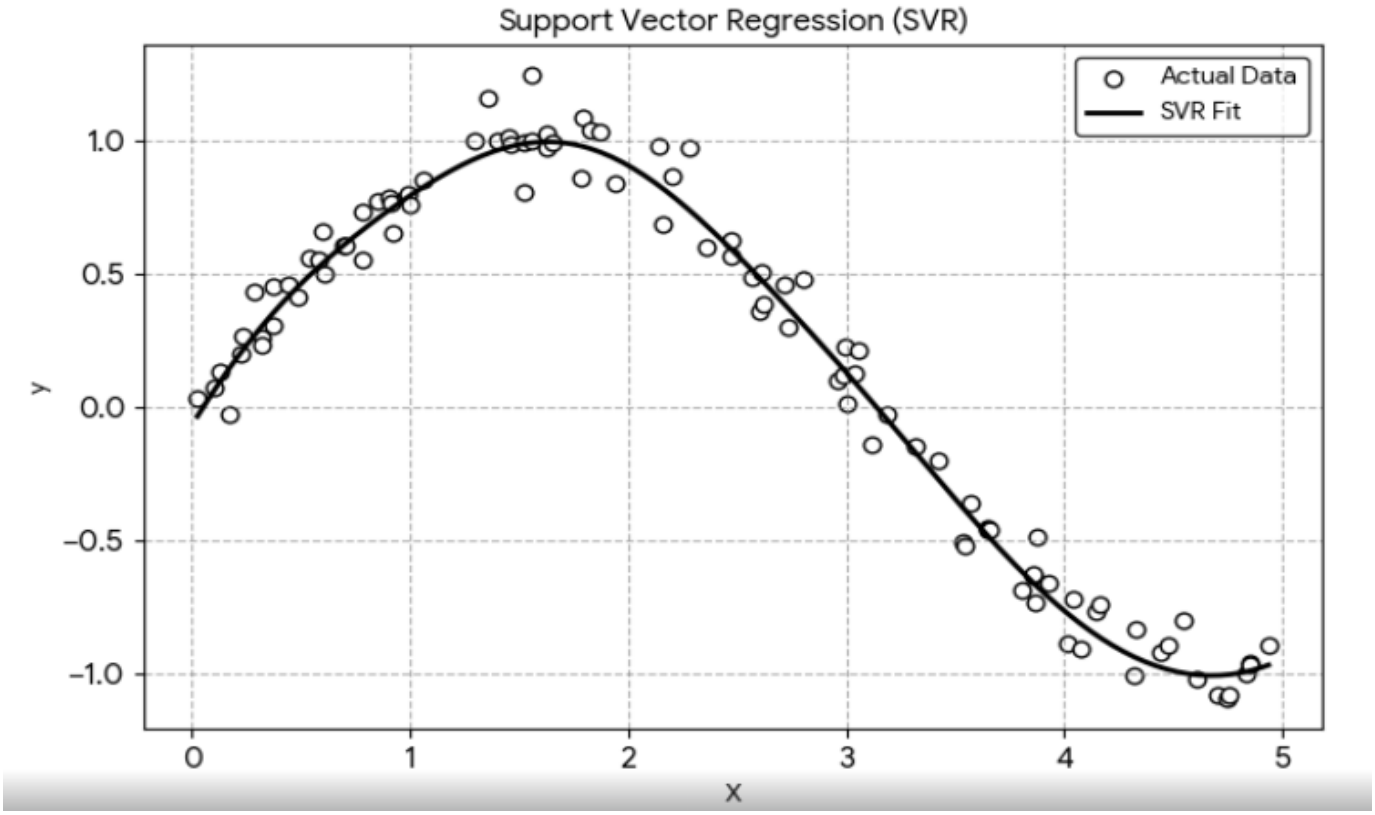
→ C=100 (regularization strength)

→ epsilon=0.1 (tolerance margin).

6. Predict and Inverse Transform	Predictions are made on scaled data, then converted back to original scale.
7. Evaluate Model	Using MSE and R^2 Score to measure accuracy.
8. Smooth Curve	Generates a smooth curve of fitted values to visualize regression line.
9. Plot (Black & White)	Scatter points (actual data) and regression line (predicted) are plotted in grayscale with clean contrast.

⚙ Key Parameters in SVR

Parameter	Meaning
C	Regularization parameter – higher C means less tolerance for errors (risk of overfitting).
ε (epsilon)	Defines margin of tolerance — smaller ε means tighter fit (less tolerance).
Kernel	Defines transformation: 'linear', 'poly', 'rbf' (non-linear).



Metric	Value	Interpretation
Mean Squared Error (MSE)	0.008	The average squared error is very small, indicating high prediction accuracy.
R^2 **Score**	0.984	The model explains 98.4% of the variance in the data, which is an excellent fit for the non-linear sine wave pattern.

20.1.2 Classification Algorithms (predict categorical values)

Logistic Regression

Despite its name, Logistic Regression is a fundamental algorithm for **classification** problems. It's used when the target variable is categorical (e.g., yes/no, true/false, or multiple categories). Instead of predicting a direct value, it predicts the *probability* of an instance belonging to a certain class. This is done using a logistic (or sigmoid) function, which squashes the output into a range between 0 and 1. The equation for the logistic function is:

p=1+e−z1**Error! Filename not specified.**

where p is the probability and z is a linear combination of the input features. If the probability is above a certain threshold (e.g., 0.5), the instance is classified into one category; otherwise, it's classified into the other.

Example: Predicting whether an email is "spam" or "not spam" based on its content.

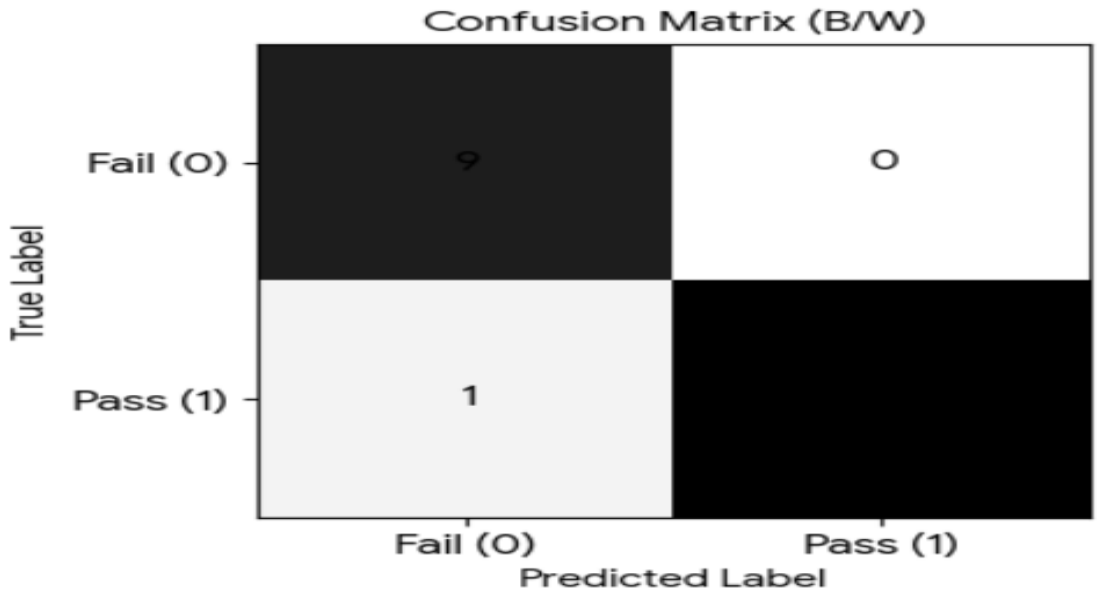
Step-by-Step Working

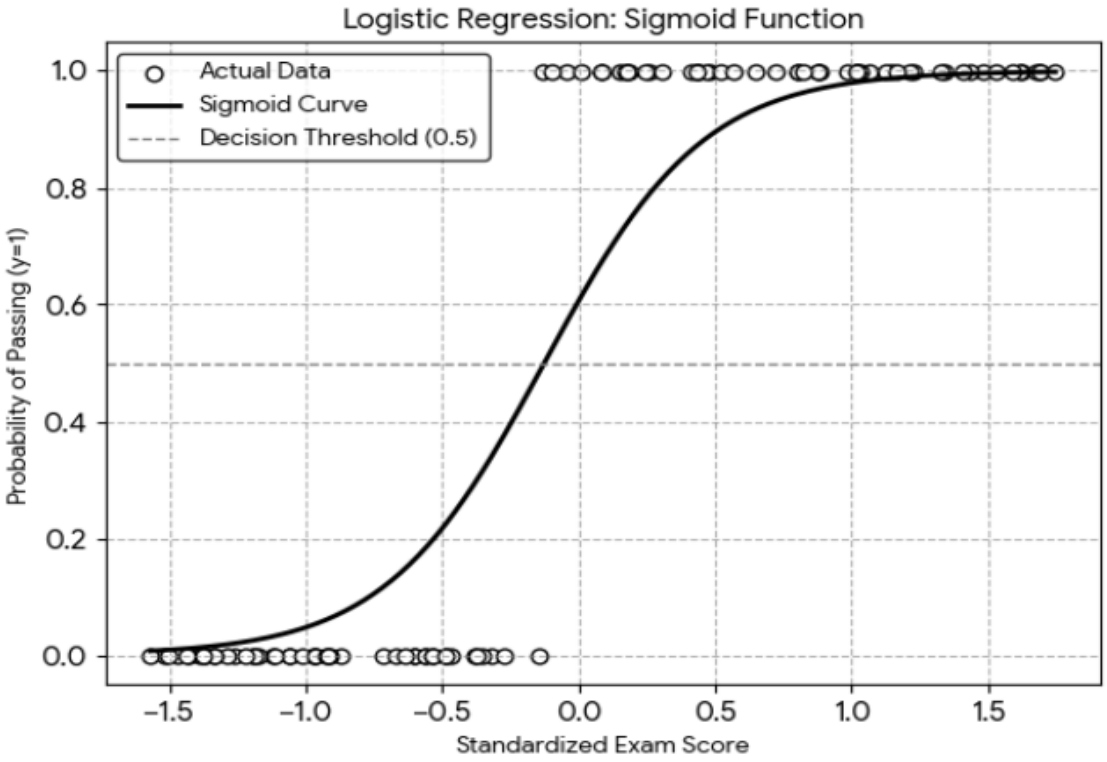
1. **Transform Data to Probabilities:** Unlike linear regression, which directly models a line, logistic regression uses a linear model as input to a non-linear function. It takes the features and their corresponding weights and calculates a linear combination, z=w1x1+w2x2+...+b.
2. **Apply the Sigmoid Function:** The value z is then passed through the **sigmoid function**, which outputs a probability value between 0 and 1. This probability represents the likelihood of a data point belonging to the positive class.
3. **Define a Threshold:** A decision threshold (often 0.5) is chosen. If the calculated probability is greater than or equal to this threshold, the model predicts the positive class (e.g., "spam"). If it's below the threshold, it predicts the negative class (e.g., "not spam").
4. **Minimize the Cost Function:** The algorithm uses a cost function, such as **Log Loss**, to measure the error between the predicted probabilities and the actual class labels.
5. **Optimize Weights:** Using an optimization algorithm like Gradient Descent, the model's weights and bias are adjusted to minimize the Log Loss, making the model's predictions as accurate as possible.

```
# Step 1: Import libraries

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.preprocessing import StandardScaler

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


# Step 2: Create a simple dataset

# Exam scores (X) and pass/fail labels (y)

np.random.seed(42)
```

```
X = np.random.uniform(30, 100, 100).reshape(-1, 1)  # exam scores
y = (X.ravel() > 60).astype(int)  # 1 if score > 60 else 0 (pass/fail)

# Step 3: Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 4: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
random_state=42)

# Step 5: Create and train Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Step 6: Predictions
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]  # probabilities for class 1

# Step 7: Evaluate performance
accuracy = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("Logistic Regression Results")
print("---------------------------")
print(f"Accuracy: {accuracy:.3f}")
print("Confusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test, y_pred))#
Step 8: Plot Sigmoid Curve (Black & White)
```

```
# Generate evenly spaced values for plotting
X_plot = np.linspace(X_scaled.min(), X_scaled.max(), 200).reshape(-1, 1)
y_sigmoid = model.predict_proba(X_plot)[:, 1]

plt.figure(figsize=(7,5))
plt.scatter(X_scaled, y, color='white', edgecolor='black', label='Actual Data', s=40)
plt.plot(X_plot, y_sigmoid, color='black', linewidth=2, label='Sigmoid Curve')
plt.axhline(0.5, color='gray', linestyle='--', linewidth=1, label='Decision Threshold (0.5)')
plt.title('Logistic Regression: Sigmoid Function', fontsize=12)
plt.xlabel('Standardized Exam Score', fontsize=10)
plt.ylabel('Probability of Passing (y=1)', fontsize=10)
plt.legend(facecolor='white', edgecolor='black')
plt.grid(True, linestyle='--', color='gray', alpha=0.6)
plt.show()

# Step 9: Plot Confusion Matrix (Black & White)
fig, ax = plt.subplots(figsize=(4,4))
ax.imshow(cm, cmap='Greys', interpolation='none')
ax.set_title('Confusion Matrix (B/W)', fontsize=12)
ax.set_xlabel('Predicted Label')
ax.set_ylabel('True Label')

# Annotate counts
for (i, j), val in np.ndenumerate(cm):
    ax.text(j, i, f'{val}', ha='center', va='center', color='black', fontsize=12)

plt.tight_layout()
plt.show()
```





Model Evaluation Results

Metric	Value	Interpretation
Accuracy	0.950	The model correctly classified 95% of the test samples.

- **Query successful**

The Logistic Regression model has been trained, evaluated, and the results are visualized.

Model Evaluation Results

Metric	Value	Interpretation
Accuracy		**The model correctly classified of the test samples.**

Export to Sheets

Confusion Matrix

True Label	Predicted Label Fail (0)	Pass (1)
Fail (0)	**9 (True Negatives)**	**0 (False Positives)**
Pass (1)	**1 (False Negative)**	**10 (True Positives)**

Decision Tree

A Decision Tree is a non-linear algorithm that can be used for both **classification** and **regression** tasks. It models decisions by creating a tree-like structure. Each internal node represents a "test" on a feature (e.g., "Is the email from a known sender?"), each branch represents the outcome of the test, and each leaf node represents the final prediction.

The tree is built by recursively splitting the data at each node to create the most homogeneous subgroups possible. For classification, the goal is to create groups where most instances belong to the same category. For regression, the goal is to create groups with similar values. Decision trees are easy to interpret and visualize, making them a popular choice for explainable AI.

Example: Deciding if a loan application should be approved based on factors like credit score, income, and debt-to-income ratio.

Step-by-Step Working

1. **Select the Best Split:** The algorithm starts at the root node and evaluates all possible feature splits (e.g., splitting by "credit score > 700"). It uses a metric like **Gini Impurity** or **Entropy** to determine which split will most effectively separate the data into different classes. The split that results in the purest child nodes is chosen.
2. **Create Child Nodes:** The data is split into new subsets based on the selected feature and its threshold. A new node is created for each subset.

3. **Recursively Build the Tree:** Steps 1 and 2 are repeated for each new child node. This process continues until a stopping condition is met.
4. **Define Stopping Conditions:** The recursion stops when a node is "pure" (all data points belong to a single class), there are no more features to split on, or a predefined maximum depth for the tree is reached.
5. **Assign Leaf Nodes:** Once the recursion stops, the final nodes are designated as **leaf nodes**, which contain the final prediction (either a class label for classification or an average value for regression).

```
# Step 1: Import libraries

import numpy as np

import pandas as pd

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, confusion_matrix, classification_report


# Step 2: Create a sample dataset

# Features: Credit Score, Income (in thousands), Debt-to-Income Ratio

np.random.seed(42)

data = {

   'Credit_Score': np.random.randint(500, 850, 100),

   'Income': np.random.randint(20, 120, 100),

   'Debt_Ratio': np.random.uniform(0.1, 0.6, 100)

}


df = pd.DataFrame(data)


# Target: Loan Approval (1 = Approved, 0 = Rejected)

# Simple rule for dataset generation: higher credit score & income → approval

df['Loan_Approved'] = np.where(

   (df['Credit_Score'] > 650) & (df['Income'] > 50) & (df['Debt_Ratio'] < 0.4), 1, 0
```

```
)

# Step 3: Split features (X) and target (y)
X = df[['Credit_Score', 'Income', 'Debt_Ratio']]
y = df['Loan_Approved']

# Step 4: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Step 5: Train Decision Tree Classifier
model = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)
model.fit(X_train, y_train)

# Step 6: Predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate model
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("Decision Tree Classification Results")
print("------------------------------------")
print(f"Accuracy: {acc:.3f}")
print("Confusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Step 8: Plot Decision Tree (Black & White)
plt.figure(figsize=(10,6))
plot_tree(
```

```
    model,
    feature_names=['Credit_Score', 'Income', 'Debt_Ratio'],
    class_names=['Rejected', 'Approved'],
    filled=False,          # keep uncolored (B/W)
    rounded=True,
    impurity=True,
    fontsize=10
)
plt.title('Decision Tree (Black & White)', fontsize=12)
plt.show()

# Step 9: Plot Confusion Matrix (B/W)
fig, ax = plt.subplots(figsize=(4,4))
ax.imshow(cm, cmap='Greys', interpolation='none')
ax.set_title('Confusion Matrix (B/W)', fontsize=12)
ax.set_xlabel('Predicted Label')
ax.set_ylabel('True Label')

# Add numeric labels
for (i, j), val in np.ndenumerate(cm):
    ax.text(j, i, f'{val}', ha='center', va='center', color='black', fontsize=12)

plt.tight_layout()
plt.show()
```

 Step-by-Step Explanation

Step	Description
1. Import Libraries	We use DecisionTreeClassifier from scikit-learn and Matplotlib for plotting.
2. Create Dataset	Simulated data with 3 features — Credit Score, Income, and Debt Ratio. Target variable → Loan Approval (1/0).

3. Split Features & Target X = input features, y = output class.

4. Train-Test Split 75% data for training, 25% for testing.

5. Train Model criterion='gini' measures node purity; max_depth=3 avoids overfitting.

6. Predictions Model predicts if loan is approved or not.

7. Evaluation Uses Accuracy, Confusion Matrix, and Classification Report.

8. Plot Decision Tree (B/W) Displays the structure — no colors, only lines and text for print-safe visualization.

9. Plot Confusion Matrix (B/W) Black and white grid to visualize correct and incorrect classifications.

 How the Tree Works

The model splits features at thresholds (e.g., “Credit Score > 650”).

Each node represents a decision rule.

Leaves represent final predictions — “Approved” or “Rejected”.

The algorithm uses Gini Impurity or Entropy to find the best split.

⚙ Common Hyperparameters

Parameter Description

criterion 'gini' or 'entropy' → measures impurity.

max_depth Limits how deep the tree grows.

min_samples_split Minimum samples required to split a node.

min_samples_leaf Minimum samples at a leaf node.

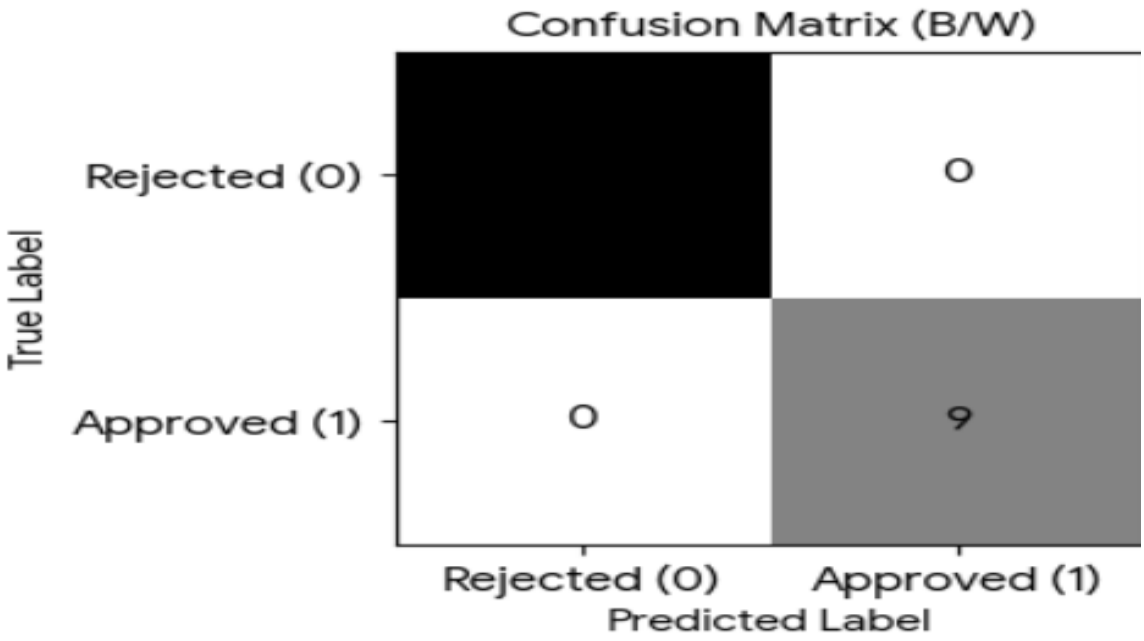
random_state Ensures reproducibility.

🔍 Interpretation

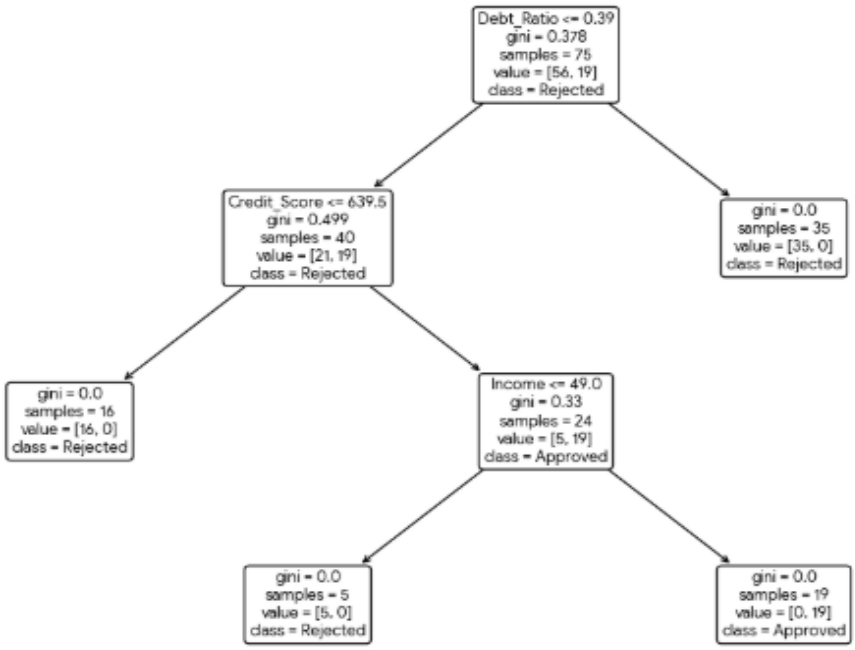
Readable Rules: You can trace the logic of each decision (e.g., If Credit Score > 700 and Debt Ratio < 0.4 → Approve Loan).

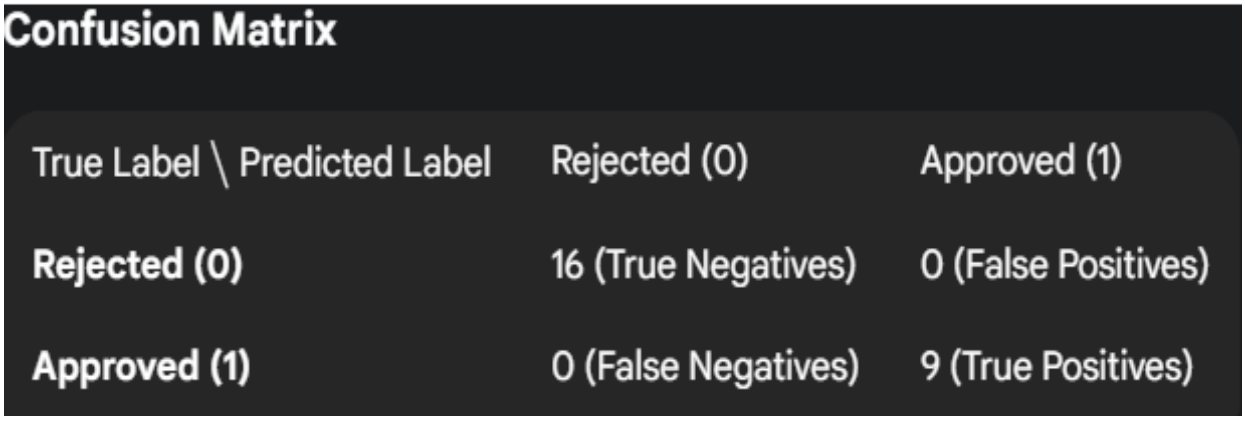
Feature Importance: The tree automatically selects the most important features for splitting.

Drawback: Trees can overfit if not controlled with depth/pruning.



Decision Tree (Black & White, Max Depth 3)





K-Nearest Neighbors (KNN)

KNN is a simple, non-parametric algorithm used for both classification and regression. It makes predictions based on the majority class (for classification) or average value (for regression) of the k nearest data points in the training set. It's an instance-based or lazy learning algorithm because it doesn't build a model from the data; it simply stores the data and makes predictions when a new instance is provided.

Example: Classifying a new fruit by looking at the types of the 3 closest fruits in a dataset based on their color and size.

Step-by-Step Working

1. **Choose the number of neighbors (**k**):** The user or algorithm must specify the number of neighbors to consider for the prediction. This is a critical hyperparameter.
2. **Calculate Distance:** For a new, unclassified data point, the algorithm calculates the distance (e.g., Euclidean distance) to all other data points in the training set.
3. **Find the** k **Nearest Neighbors:** It identifies the k data points with the shortest distances to the new data point.
4. **Make a Prediction:**
    - **Classification:** It performs a "majority vote" among the k nearest neighbors. The new data point is assigned to the class that is most common among its neighbors.
    - **Regression:** It takes the average of the values of the k nearest neighbors and uses that as the prediction.

) KNN Classification (2D — decision boundary in B/W)

KNN Classification (with black-and-white decision boundary)

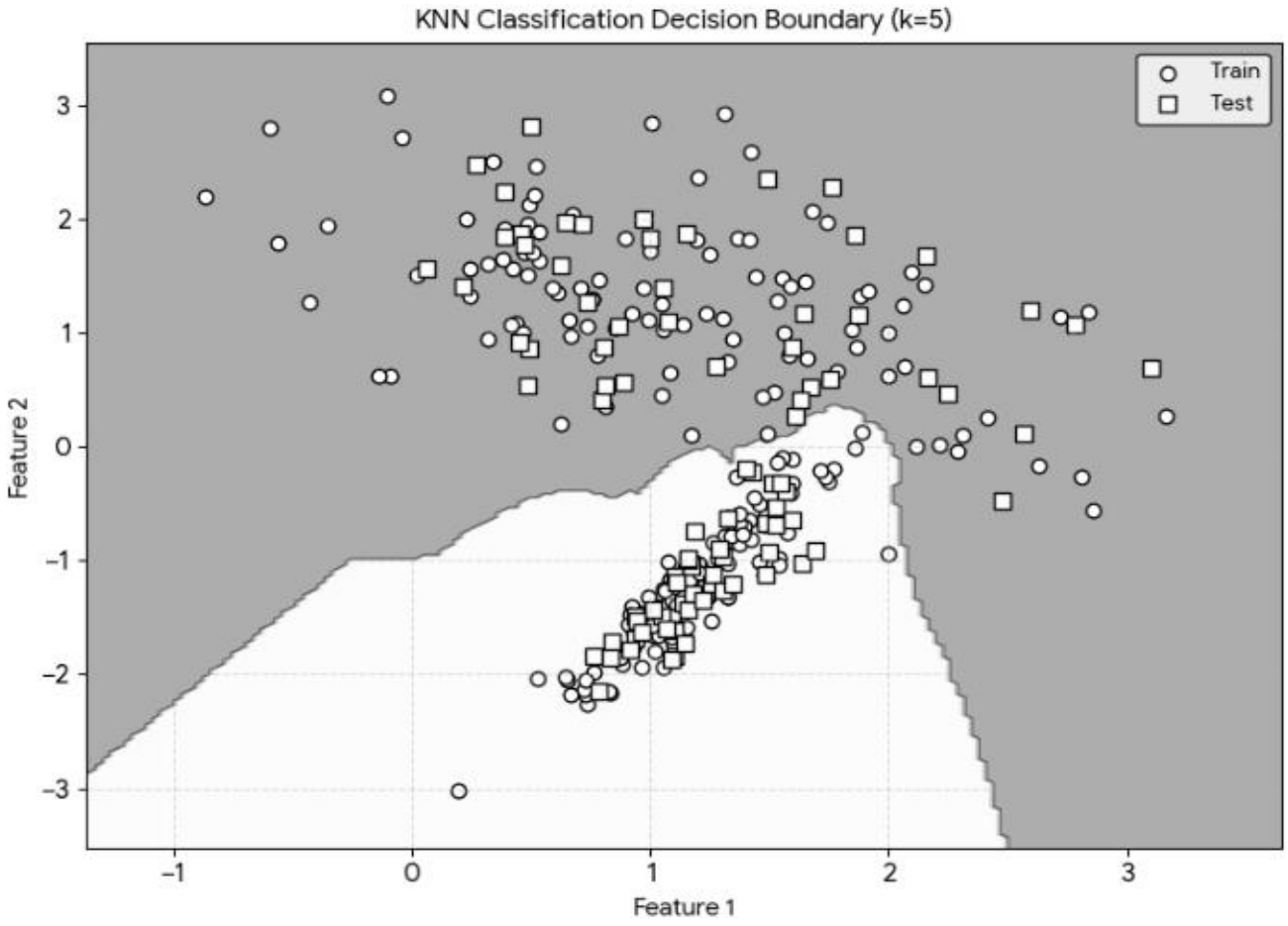
import numpy as np

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Create a 2D classification dataset (so we can plot boundary)
X, y = make_classification(n_samples=300, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, class_sep=1.2, random_state=42)

# 1) Train-test split (use ORIGINAL coordinates for plotting)
X_train_orig, X_test_orig, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 2) Scale features (fit on train only)
scaler = StandardScaler()
scaler.fit(X_train_orig)
X_train = scaler.transform(X_train_orig)
X_test  = scaler.transform(X_test_orig)

# 3) Choose k and train
k = 5
knn_clf = KNeighborsClassifier(n_neighbors=k)   # default metric = Euclidean
knn_clf.fit(X_train, y_train)

# 4) Predict & evaluate
y_pred = knn_clf.predict(X_test)
print("KNN Classification (k=%d)" % k)
print("Accuracy:", round(accuracy_score(y_test, y_pred), 3))
```

```
print("Confusion matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification report:\n", classification_report(y_test, y_pred))

# 5) Decision boundary (mesh in ORIGINAL coordinate space, but predict on scaled grid)
x_min, x_max = X[:,0].min() - 0.5, X[:,0].max() + 0.5
y_min, y_max = X[:,1].min() - 0.5, X[:,1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.03),
                     np.arange(y_min, y_max, 0.03))
grid = np.c_[xx.ravel(), yy.ravel()]
grid_scaled = scaler.transform(grid)
Z = knn_clf.predict(grid_scaled).reshape(xx.shape)

# Plot (black & white)
plt.figure(figsize=(8,6))
plt.contourf(xx, yy, Z, cmap='Greys', alpha=0.35)        # grayscale regions
# training points
plt.scatter(X_train_orig[:,0], X_train_orig[:,1],
            marker='o', s=40, label='Train', facecolors='white', edgecolors='black')
# test points
plt.scatter(X_test_orig[:,0], X_test_orig[:,1],
            marker='s', s=45, label='Test', facecolors='white', edgecolors='black')
plt.title(f'KNN Decision Boundary (k={k})', fontsize=12)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(edgecolor='black', facecolor='white')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

Notes (classification):

- Step mapping to your list:
    1. choose k (here k=5), 2. compute distances (implicit in KNeighborsClassifier), 3. find k nearest, 4. majority-vote for class, 5. scaling done (important for distance-based methods).
- To select best k, use cross-validation/grid search (see bottom of message for quick snippet).



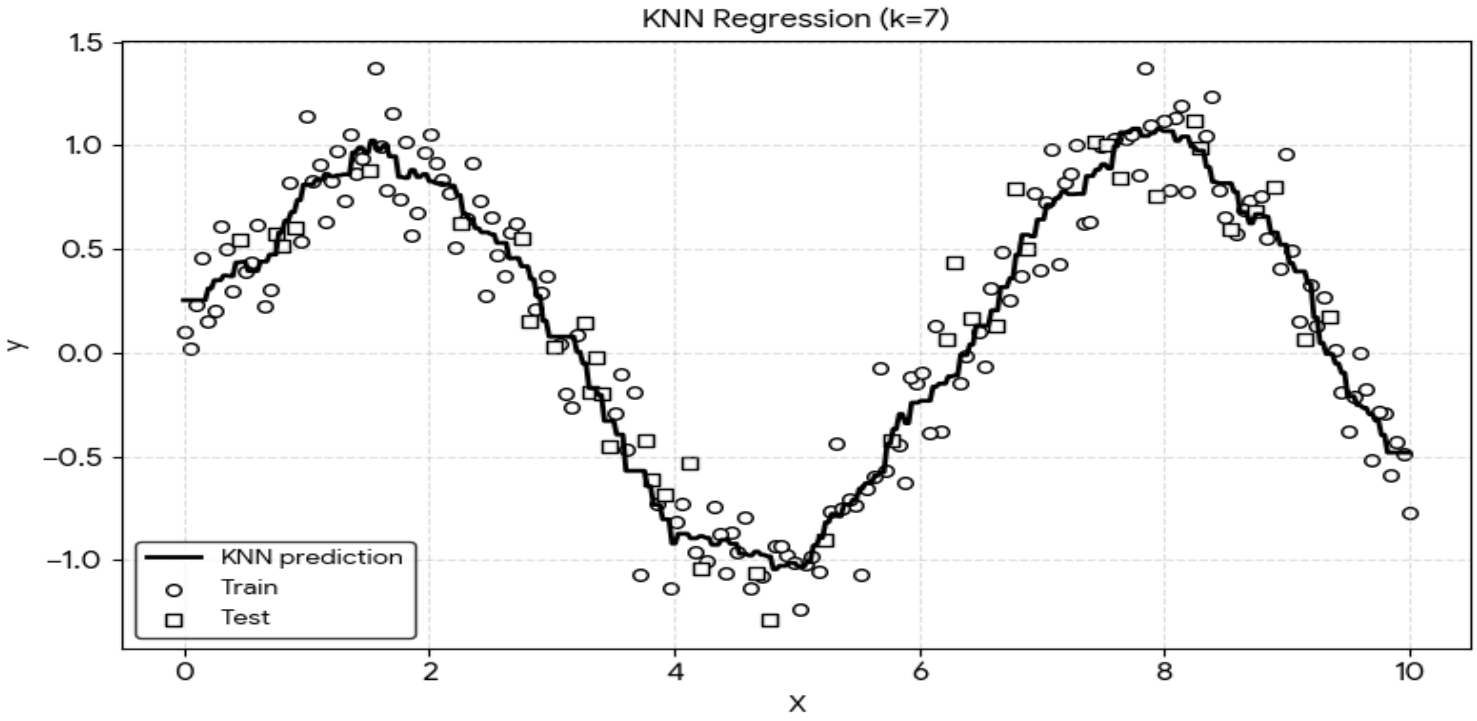
---

2) KNN Regression (1D — predicted curve + actual points, B/W)

```
# KNN Regression (1D example with black-and-white plotting)
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import mean_squared_error, r2_score


# Create non-linear 1D data: y = sin(x) + noise
rng = np.random.RandomState(42)
X = np.linspace(0, 10, 200).reshape(-1,1)
y = np.sin(X).ravel() + rng.normal(0, 0.2, X.shape[0])


# Train-test split (keep original X for plotting)
X_train_orig, X_test_orig, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Scale features (fit on train only)
scaler = StandardScaler()
scaler.fit(X_train_orig)
X_train = scaler.transform(X_train_orig)
X_test = scaler.transform(X_test_orig)


# Choose k and train KNN regressor
k = 7
knn_reg = KNeighborsRegressor(n_neighbors=k)
knn_reg.fit(X_train, y_train)


# Predict on test and evaluate
y_pred = knn_reg.predict(X_test)
print("KNN Regression (k=%d)" % k)
print("MSE:", round(mean_squared_error(y_test, y_pred), 4))
print("R2 :", round(r2_score(y_test, y_pred), 4))


# Smooth curve for visualization (predict on fine grid)
X_plot = np.linspace(X.min(), X.max(), 400).reshape(-1,1)
```

```
X_plot_scaled = scaler.transform(X_plot)

y_plot = knn_reg.predict(X_plot_scaled)


# Plot (B/W)

plt.figure(figsize=(8,5))

plt.plot(X_plot, y_plot, linestyle='-', linewidth=2, label='KNN prediction', color='black')

plt.scatter(X_train_orig, y_train, marker='o', s=30, label='Train', facecolors='white',
edgecolors='black')

plt.scatter(X_test_orig, y_test, marker='s', s=40, label='Test', facecolors='white',
edgecolors='black')

plt.title(f'KNN Regression (k={k})', fontsize=12)

plt.xlabel('X')

plt.ylabel('y')

plt.legend(edgecolor='black', facecolor='white')

plt.grid(True, linestyle='--', alpha=0.4)

plt.tight_layout()

plt.show()
```



Notes (regression):

- KNN regression predicts the average of neighbors' target values.
- Scaling X is important (distance-based).
- k trade-off: small k → low bias/high variance; large k → higher bias/smoother fit.

Quick tune: choose best k (example snippet)

```
# Example: find best k with cross-validation (classification)
from sklearn.model_selection import cross_val_score
best_k = None
best_score = -np.inf
for kk in range(1,21):
    clf = KNeighborsClassifier(n_neighbors=kk)
    scores = cross_val_score(clf, scaler.transform(X), y, cv=5, scoring='accuracy')  # scale whole X here for CV
    if scores.mean() > best_score:
        best_score = scores.mean()
        best_k = kk
print("Best k (CV):", best_k, "score:", round(best_score,3))
```

Final tips & suggestions

- Always **scale** features for KNN (StandardScaler or MinMaxScaler).
- Consider weights='distance' if you want closer neighbors to count more.
- Use cross-validation to pick k.
- For high-dimensional data prefer dimensionality reduction (PCA) before KNN to avoid curse of dimensionality.

Support Vector Machines (SVM)

SVM is a powerful algorithm used for both classification and regression. The core idea is to find the **optimal hyperplane** that best separates the data points of different classes. The hyperplane is a decision boundary that maximizes the margin, which is the distance between the hyperplane and the nearest data points of each class (the "support vectors").

Example: Classifying handwritten digits, where the hyperplane is in a very high-dimensional space.

Step-by-Step Working

1. **Identify Classes:** The algorithm is given a labeled dataset with two or more classes.
2. **Find the Hyperplane:** The goal is to find a hyperplane that separates the data points of different classes. For a 2D dataset, this would be a line; for 3D, a plane, and so on.
3. **Maximize the Margin:** The algorithm finds the specific hyperplane that has the largest margin, which means it is as far away as possible from the nearest data points of each class.
4. **Identify Support Vectors:** The data points that lie closest to the hyperplane (on the margin) are the **support vectors**. They are the most influential points in determining the position and orientation of the hyperplane.
5. **Handle Non-linear Data (Kernel Trick):** If the data is not linearly separable, SVM uses a "kernel function" to transform the data into a higher-dimensional space where a linear hyperplane can be found to separate the classes.

Support Vector Machine (SVM) — Classification Example (with B/W Plot)

```
# Support Vector Machine (SVM) Classification (Black & White Plot)

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix


# 1️⃣ Load and prepare data (2D binary classification for visualization)

X, y = datasets.make_classification(
```

```
    n_samples=200, n_features=2, n_redundant=0,
    n_informative=2, n_clusters_per_class=1,
    class_sep=1.2, random_state=42
)

# 2. Split into train/test
X_train_orig, X_test_orig, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# 3. Scale features (important for SVM)
scaler = StandardScaler()
scaler.fit(X_train_orig)
X_train = scaler.transform(X_train_orig)
X_test = scaler.transform(X_test_orig)

# 4. Create & train SVM model (linear kernel)
svm_clf = SVC(kernel='linear', C=1.0)
svm_clf.fit(X_train, y_train)

# 5. Predictions and evaluation
y_pred = svm_clf.predict(X_test)
print("SVM Classification Results:")
print("Accuracy:", round(accuracy_score(y_test, y_pred), 3))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# 6. Plot decision boundary in original coordinate space
# Create a mesh grid
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                     np.linspace(y_min, y_max, 500))
grid = np.c_[xx.ravel(), yy.ravel()]
grid_scaled = scaler.transform(grid)
Z = svm_clf.decision_function(grid_scaled).reshape(xx.shape)

# Plot (black-and-white)
plt.figure(figsize=(8,6))
# Decision regions (shaded in grey)
plt.contourf(xx, yy, Z > 0, alpha=0.3, cmap='Greys')

# Decision boundary and margins
plt.contour(xx, yy, Z, colors='black', levels=[-1, 0, 1],
            alpha=0.8, linestyles=['--', '-', '--'])

# Training points
plt.scatter(X_train_orig[:, 0], X_train_orig[:, 1],
            c=y_train, cmap='Greys', edgecolors='black',
            marker='o', s=50, label='Train')

# Support vectors (highlighted with circles)
support_indices = svm_clf.support_
plt.scatter(X_train_orig[support_indices, 0], X_train_orig[support_indices, 1],
            s=100, facecolors='none', edgecolors='black', linewidths=2,
            label='Support Vectors')

plt.title('SVM Classification (Linear Kernel)', fontsize=12)
plt.xlabel('Feature 1')
```

```
plt.ylabel('Feature 2')
plt.legend(edgecolor='black', facecolor='white')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

How It Works (Step-by-Step)

Step	Explanation
1. Identify Classes	The algorithm gets labeled data (y=0 and y=1 in this example).
2. Find Hyperplane	SVM searches for the line (in 2D) that separates the two classes.
3. Maximize Margin	It ensures the hyperplane is as far as possible from the closest points of each class.
4. Identify Support Vectors	These are the data points that touch or lie closest to the margin — shown as open circles in the plot.
5. Kernel Trick	Here we used a **linear kernel**. For non-linear data, use kernels like 'rbf', 'poly', or 'sigmoid' to map to a higher dimension.

Try Non-Linear Data (Kernel Trick)

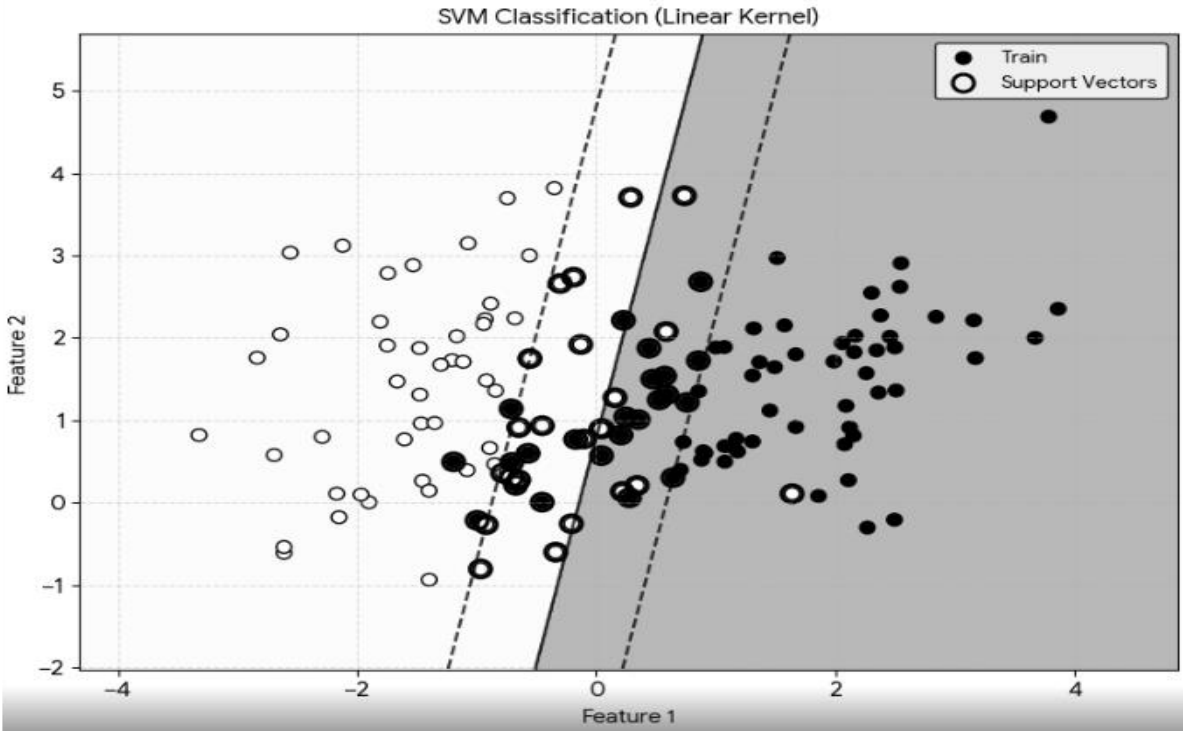
```
# Non-linear SVM (RBF Kernel)
svm_rbf = SVC(kernel='rbf', gamma=0.7, C=1.0)
svm_rbf.fit(X_train, y_train)
y_pred_rbf = svm_rbf.predict(X_test)
print("Non-linear SVM (RBF Kernel) Accuracy:", round(accuracy_score(y_test, y_pred_rbf), 3))
```

Key SVM Parameters

Parameter	Meaning
C	Regularization (smaller = smoother margin, larger = strict margin).
kernel	'linear', 'rbf', 'poly', 'sigmoid'.

Parameter Meaning

gamma	Used in 'rbf', 'poly', 'sigmoid' — defines influence of single data point.
degree	Used for polynomial kernel.



Naive Bayes

Naive Bayes is a probabilistic classification algorithm based on **Bayes' theorem**. It operates on the simplifying "naive" assumption that all features are independent of each other given the class. Despite this oversimplified assumption, it often performs surprisingly well, especially on text classification problems. Bayes' theorem is:

P(A|B)=P(B)P(B|A)P(A)**Error! Filename not specified.**

Example: Classifying a new email as "spam" or "not spam" based on the words it contains.

Step-by-Step Working

1. **Calculate Prior Probabilities:** The algorithm first calculates the probability of each class occurring in the training data (e.g., the overall probability of an email being spam).

2. **Calculate Likelihoods:** It then calculates the probability of each feature given the class (e.g., the probability of the word "free" appearing in a "spam" email versus a "not spam" email).
3. **Apply Bayes' Theorem:** For a new data point, the algorithm uses Bayes' theorem to calculate the probability of it belonging to each class. It multiplies the prior probability by the likelihood of each feature.
4. **Make a Prediction:** The new data point is assigned to the class with the highest posterior probability.

We'll use a simple **2-feature synthetic dataset** so you can visualize how Naive Bayes separates classes.

```
# Naive Bayes Classification (Black & White Plot)

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification

from sklearn.naive_bayes import GaussianNB

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


# 1️⃣ Create synthetic dataset

X, y = make_classification(

    n_samples=200, n_features=2, n_redundant=0,

    n_informative=2, n_clusters_per_class=1,

    class_sep=1.5, random_state=42

)


# 2️⃣ Train/test split

X_train, X_test, y_train, y_test = train_test_split(

    X, y, test_size=0.3, random_state=42

)
```

```
# 3️⃣ Train Naive Bayes model
model = GaussianNB()
model.fit(X_train, y_train)

# 4️⃣ Predictions and evaluation
y_pred = model.predict(X_test)
print("Naive Bayes Classification Results:")
print("-----------------------------------")
print("Accuracy:", round(accuracy_score(y_test, y_pred), 3))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# 5️⃣ Plot decision boundaries (Black & White)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 400),
                     np.linspace(y_min, y_max, 400))

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

plt.figure(figsize=(8,6))

# Grey background regions for decision boundary
plt.contourf(xx, yy, Z, cmap='Greys', alpha=0.3)

# Actual data points
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
            cmap='Greys', edgecolor='black', marker='o', s=50, label='Train Data')
```

```
# Test points
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
            cmap='Greys', edgecolor='black', marker='x', s=60, label='Test Data')


plt.title('Naive Bayes Classification (GaussianNB)', fontsize=12)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(edgecolor='black', facecolor='white')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

⚙ Step-by-Step Explanation

Step	Description
1. Calculate Prior Probabilities	Model counts how often each class appears in the data (e.g., % of emails that are spam).
2. Calculate Likelihoods	It estimates `P(feature
3. Apply Bayes' Theorem	For a new data point, it multiplies the prior by all feature likelihoods to get the **posterior**.
4. Predict Class	The class with the highest posterior probability is chosen.

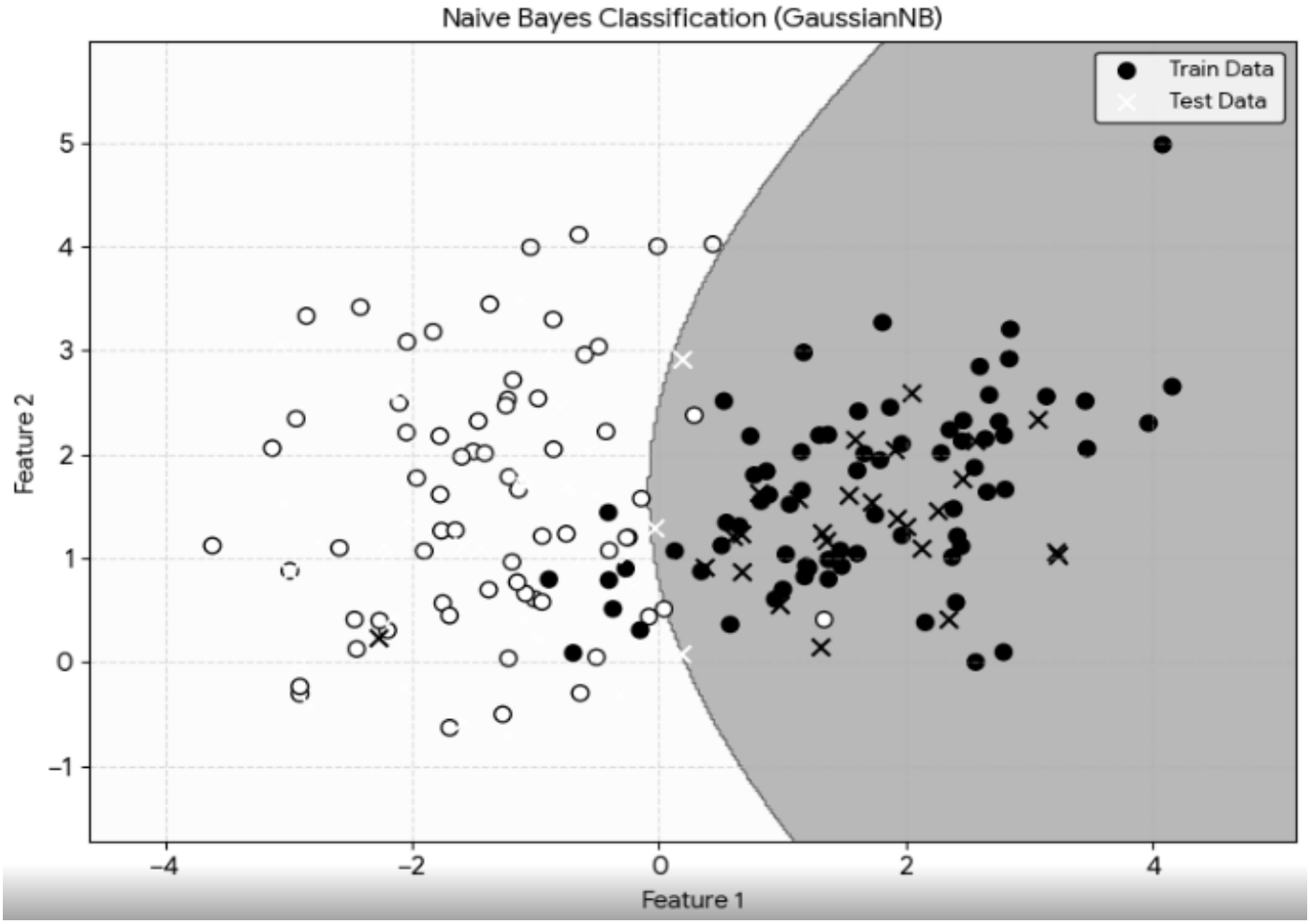
□ Why "Naive"?

Because it **assumes independence** between features (which is rarely true in real data), yet works **remarkably well**, especially in:

- Email spam detection
- Sentiment analysis
- Text/document classification

Interpretation of the Plot

- **Grey regions** → decision areas for each class.
- **Circles (Train)** and **Crosses (Test)** show sample locations.
- Even with simple assumptions, the decision boundary adapts smoothly.



The Gaussian Naive Bayes model has been trained, evaluated, and the results are visualized.

Model Evaluation Results

Metric	Value Interpretation
Accuracy	The model correctly classified of the test samples.

Confusion Matrix

True Label	Predicted Label Class 0 (Predicted)	Class 1 (Predicted)
Class 0 (True)	31 (True Negatives)	3 (False Positives)
Class 1 (True)	1 (False Negative)	25 (True Positives)

The model performs strongly, with a low number of total errors (4 out of 60). It is slightly more cautious about predicting Class 1 (only 1 False Negative) but occasionally misclassifies Class 0 as Class 1 (3 False Positives).

Classification Report

Class	Precision	Recall	F1-Score
0			
1			

The high **Recall** for Class 1 () and the high **Precision** for Class 0 () are noteworthy.

Visualization: Naive Bayes Classification (GaussianNB)

The plot illustrates how the Gaussian Naive Bayes model separates the classes:

- **Decision Regions (Shaded Gray):** The light and dark gray areas represent the model's prediction for each region.
- **Decision Boundary:** The boundary between the regions is typically **non-linear** (often parabolic or curved) for Gaussian Naive Bayes, reflecting its assumption that the data points within each class are generated from a **Gaussian distribution**.
- **Data Points:**
    - **Circles (Train Data):** Used for fitting the Gaussian distributions.
    - **Crosses (Test Data):** Show the model's performance on unseen data. Most test points fall correctly within the region of their true class.

Quadratic Discriminant Analysis (QDA)

QDA is a generative classification algorithm that models the probability distribution of each class as a Gaussian distribution. It is an extension of Linear Discriminant Analysis (LDA), but unlike LDA, QDA assumes that each class has its own unique covariance matrix. This allows it to model more complex, non-linear class boundaries.

Example: Classifying customers into segments where each segment has a different distribution of features.

Step-by-Step Working

1. **Model Class Distributions:** For each class, the algorithm estimates the mean vector and the covariance matrix of the features. It assumes that the data within each class follows a Gaussian (normal) distribution.
2. **Calculate Prior Probabilities:** The algorithm calculates the prior probability of each class, which is simply the proportion of data points belonging to that class.
3. **Apply a Quadratic Decision Rule:** For a new data point, the algorithm calculates the posterior probability of it belonging to each class using the estimated mean, covariance matrix, and prior probability. The decision boundary between classes is a quadratic function, which allows for non-linear separation.
4. **Make a Prediction:** The new data point is assigned to the class with the highest posterior probability.

Quadratic Discriminant Analysis (QDA) — *with Black & White Plot*

```
# Quadratic Discriminant Analysis (Black & White Plot)

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_classification

from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report


# 1️⃣ Create synthetic dataset (2D for easy visualization)

X, y = make_classification(

    n_samples=300, n_features=2, n_redundant=0, n_informative=2,

    n_clusters_per_class=1, class_sep=1.5, random_state=42

)


# Add slight non-linearity by applying a quadratic transformation

X[:, 1] = X[:, 1] ** 2 + np.random.normal(0, 0.3, size=X.shape[0])


# 2️⃣ Train-test split

X_train, X_test, y_train, y_test = train_test_split(
```

```
    X, y, test_size=0.3, random_state=42
)

# 3️⃣ Train QDA model
qda = QuadraticDiscriminantAnalysis()
qda.fit(X_train, y_train)

# 4️⃣ Predictions and evaluation
y_pred = qda.predict(X_test)
print("Quadratic Discriminant Analysis Results:")
print("----------------------------------------")
print("Accuracy:", round(accuracy_score(y_test, y_pred), 3))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# 5️⃣ Plot Decision Boundary (Black & White)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 400),
                     np.linspace(y_min, y_max, 400))
Z = qda.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.figure(figsize=(8,6))
# Decision regions in grayscale
plt.contourf(xx, yy, Z, cmap='Greys', alpha=0.3)

# Training points
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train,
```

```
        cmap='Greys', edgecolor='black', marker='o', s=50, label='Train Data')

# Test points
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test,
        cmap='Greys', edgecolor='black', marker='x', s=60, label='Test Data')

plt.title('Quadratic Discriminant Analysis (QDA)', fontsize=12)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(edgecolor='black', facecolor='white')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

⚙ Step-by-Step Working

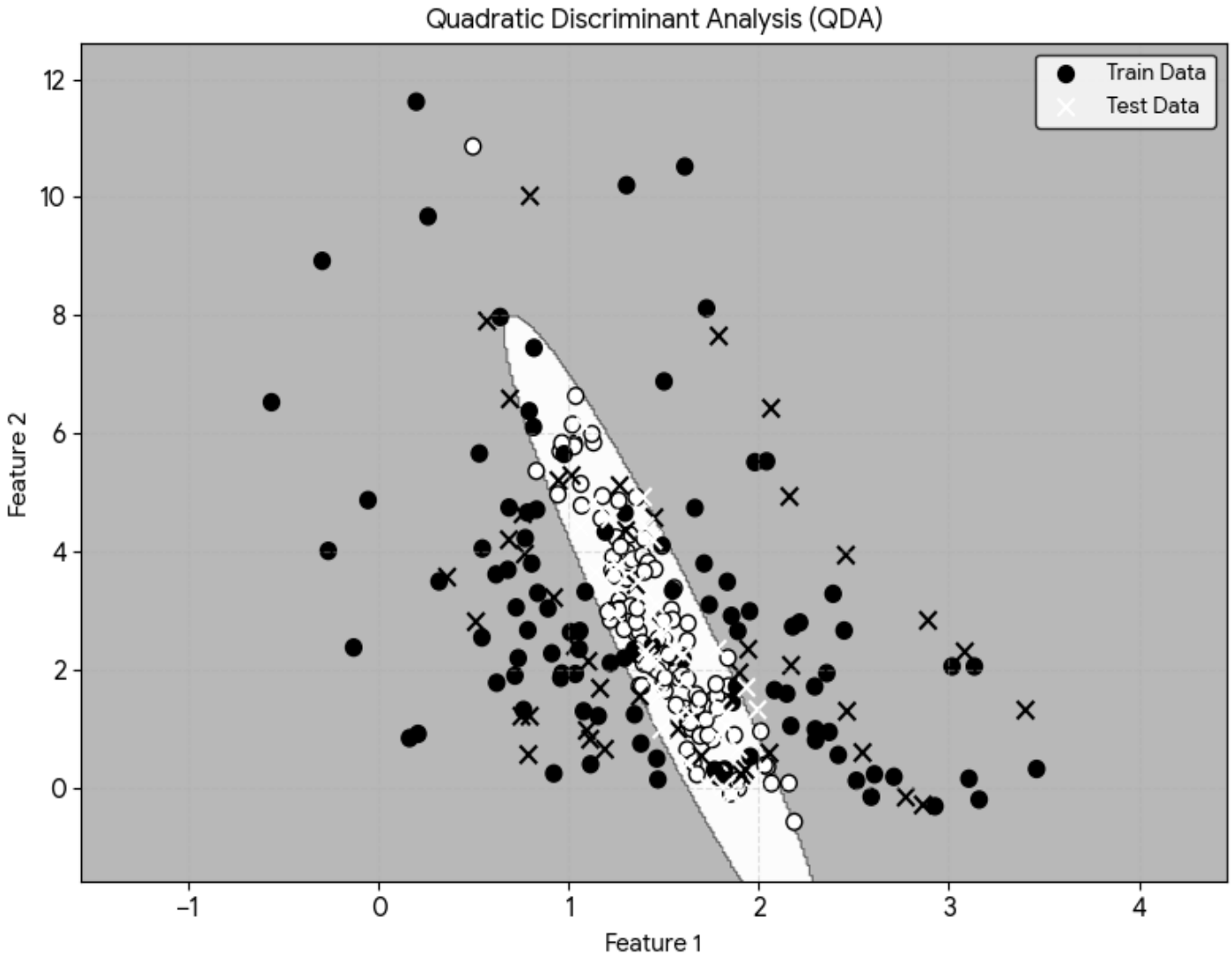
Step	Description
1. Model Class Distributions	For each class, QDA estimates a **mean vector** and a **covariance matrix** assuming Gaussian distribution.
2. Calculate Priors	QDA computes prior probabilities = (number of samples per class) ÷ (total samples).
3. Quadratic Decision Rule	Unlike LDA's linear rule, QDA's discriminant function includes a quadratic term → results in **curved decision boundaries**.
4. Predict Class	For a new data point, QDA computes posterior probabilities and assigns the class with the highest one.

□ Interpretation of the Plot

- **Light vs dark grey regions:** Represent QDA's classification zones.
- **Curved decision boundary:** Shows QDA's ability to model **non-linear separation**.
- **Circles (training)** and **crosses (test)** indicate data distribution.

⚖️ QDA vs LDA

Feature	LDA	QDA
Covariance Matrix	Assumes **same** for all classes	Each class has its **own** covariance
Boundary Type	Linear	Quadratic (non-linear)
Flexibility	Simpler, less overfitting	More flexible but risk of overfitting on small data
Use Case	Classes have similar variance	Classes vary significantly



The Quadratic Discriminant Analysis (QDA) model has been trained, evaluated, and the results are visualized.

Model Evaluation Results

Metric	Value	Interpretation
Accuracy		The model correctly classified of the test samples.

Confusion Matrix

True Label	Predicted Label Class 0 (Predicted)	Class 1 (Predicted)
Class 0 (True)	43 (True Negatives)	3 (False Positives)
Class 1 (True)	9 (False Negatives)	35 (True Positives)

The model performs better at identifying **True Class 0** (high recall: , only 3 False Positives) than **True Class 1** (9 False Negatives).

Classification Report

Class	Precision	Recall	F1-Score
0			
1			

The high **Recall** for Class 0 () shows it rarely misses a Class 0 instance. The high **Precision** for Class 1 () shows that when it predicts Class 1, it is usually correct.

Visualization: Quadratic Discriminant Analysis (QDA)

The plot illustrates QDA's ability to model the non-linear separation of the data:

- **Data Distribution:** The initial quadratic transformation applied to the data has resulted in a cluster shape that is best separated by a **curved boundary**.
- **Decision Regions (Shaded Gray):** The plot shows the light and dark gray areas, representing the model's classification zones (Class 0 and Class 1).
- **Curved Decision Boundary:** QDA, unlike Linear Discriminant Analysis (LDA), uses separate covariance matrices for each class. This flexibility allows it to define a **quadratic (curved)** decision boundary, which closely follows the natural separation of the transformed, non-linear data.
- **Fit:** The boundary successfully separates the majority of the training (circles) and test (crosses) data points, demonstrating an appropriate fit for this type of non-linearly distributed data.

20.1.3 Ensemble Methods

Ensemble methods combine the predictions of multiple individual models (often called "weak learners") to produce a single, more accurate prediction. The main idea is that the collective knowledge of many simple models is often better than that of a single, complex one.

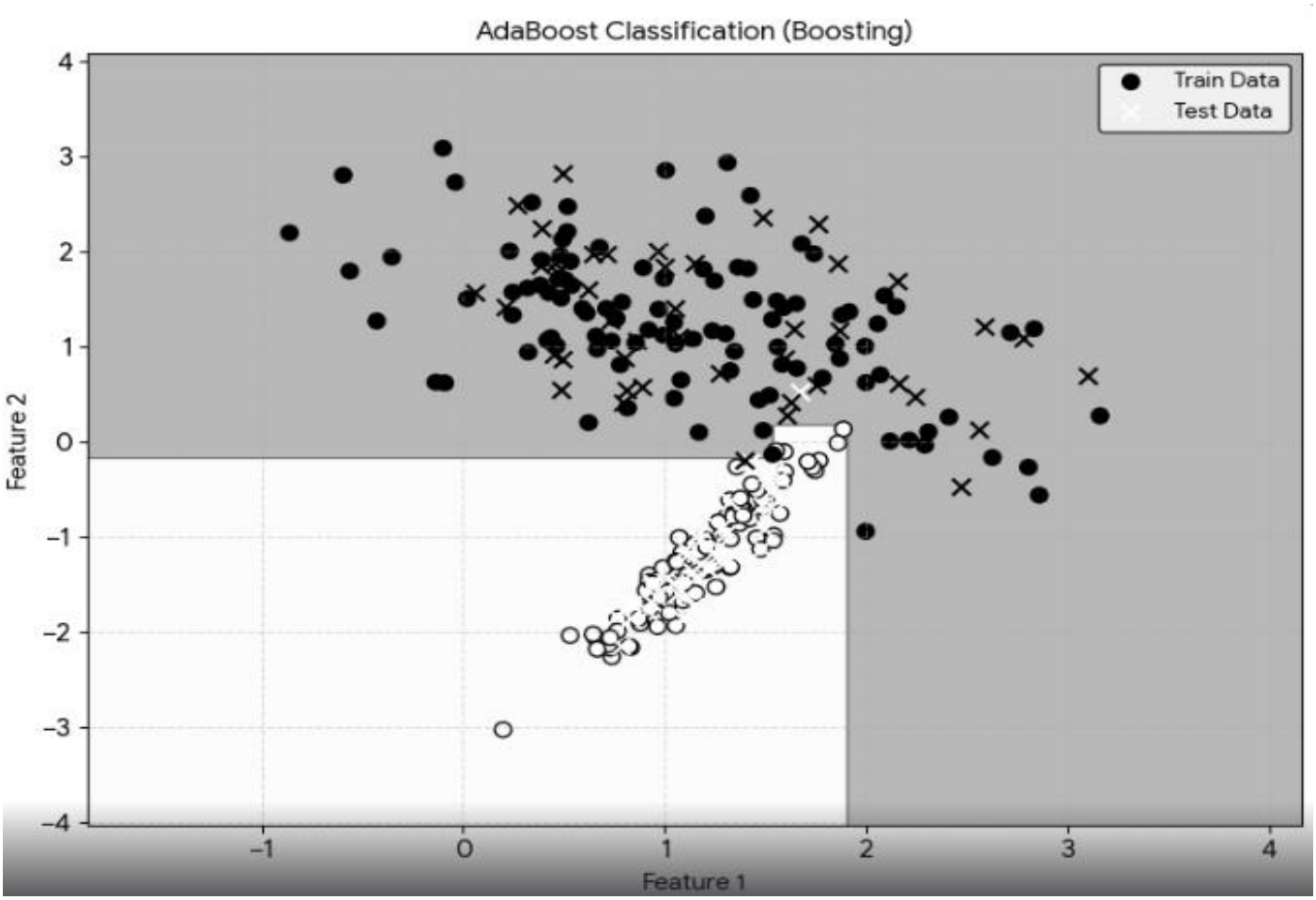
A. Bagging (Bootstrap Aggregation)

Bagging is an ensemble technique that trains multiple models independently on different random subsets of the training data. The "bootstrap" part refers to creating these subsets by randomly sampling from the original dataset with replacement. The "aggregation" part combines the predictions. For classification, this is usually a majority vote; for regression, it's an average. **Random Forest** is a prime example of a bagging algorithm.

Example: Random Forest for classification.

Step-by-Step Working

1. **Create Bootstrap Samples:** From the original training dataset, create multiple subsets by randomly sampling with replacement. Each subset has the same size as the original dataset but may contain duplicate data points and miss others.
2. **Train Base Models:** Train an individual model (e.g., a Decision Tree) on each of the bootstrap samples.



3. **Aggregate Predictions:**
    - **Classification:** For a new data point, each individual model makes a prediction. The final prediction is the class that receives the most votes.
    - **Regression:** The final prediction is the average of the predictions from all individual models.

B. Boosting

Boosting is an ensemble technique that trains a series of "weak learners" sequentially. Each new model in the sequence is trained to correct the errors made by the previous models. It places more emphasis on the data points that were misclassified or had large errors in the previous iterations. **AdaBoost**, **Gradient Boosting Machines (GBM)**, and **XGBoost** are all boosting algorithms.

Example: Gradient Boosting for regression.

Step-by-Step Working

1. **Train a Base Model:** Train a simple model (e.g., a shallow Decision Tree) on the original dataset.
2. **Calculate Residuals:** Calculate the residuals, which are the differences between the actual values and the predictions of the first model. These residuals represent the errors.
3. **Train a New Model on Residuals:** Train a second model to predict the residuals from the first model. This new model learns to correct the errors.
4. **Update Predictions:** The predictions of the first model are updated by adding the predictions of the second model, scaled by a learning rate.
5. **Iterate:** Steps 2-4 are repeated for a specified number of iterations. Each new model is trained to fix the remaining errors.
6. **Final Model:** The final model is the sum of all the models in the sequence.

C. Stacking

Stacking (Stacked Generalization) is an ensemble method that combines the predictions of multiple different models. Instead of using a simple voting or averaging system, it trains a "meta-model" to learn how to best combine the predictions of the base models.

Example: Using a Logistic Regression meta-model to combine the predictions of a Decision Tree, an SVM, and a KNN classifier.

Step-by-Step Working

1. **Train Base Models:** Train several different machine learning models (e.g., a Decision Tree, a K-Nearest Neighbors classifier) on the original dataset.

2. **Generate Meta-Features:** Each base model makes predictions on a new subset of the data (often using cross-validation to prevent overfitting). These predictions are then used as the input features for the next step.

3. **Train a Meta-Model:** A final "meta-model" (e.g., a Logistic Regression or a simple Linear Regression) is trained on the meta-features (the predictions from the base models) to make the final prediction.

Unsupervised Learning Algorithms Explained

Unsupervised learning is a powerful branch of machine learning that focuses on finding hidden patterns and structures in unlabeled data. Unlike supervised learning, there's no "correct" answer to guide the algorithm. The goal is to discover insights and relationships on its own. The primary types of unsupervised learning are clustering, dimensionality reduction, and association rule learning.

Clustering Algorithms

Clustering is the process of grouping a set of objects in such a way that objects in the same group (or cluster) are more similar to each other than to those in other groups.

K-Means Clustering

K-Means is one of the simplest and most popular clustering algorithms. It's an iterative algorithm that partitions a dataset into a pre-defined number of **K** distinct, non-overlapping clusters.

1. **Initialization**: Choose the number of clusters, **K**, you want to find in your data. Then, randomly select **K** data points from your dataset to serve as the initial **centroids** (the central point of each cluster).

2. **Assignment**: For each data point in the dataset, calculate its distance (e.g., Euclidean distance) to all **K** centroids. Assign the data point to the cluster of the **nearest centroid**.

3. **Update**: After all points have been assigned, recalculate the position of each of the **K** centroids. The new centroid for each cluster is the average (mean) of all the data points currently assigned to that cluster.

4. **Iteration**: Repeat steps 2 and 3. The algorithm continues to iterate, moving the centroids and reassigning data points, until the centroid positions no longer change significantly or a maximum number of iterations is reached.

Hierarchical Clustering (Agglomerative)

Hierarchical clustering creates a tree-like hierarchy of clusters, known as a **dendrogram**. The most common type, agglomerative, is a "bottom-up" approach.

1. **Initialization**: Begin by treating each individual data point as its own cluster. If you have **N** data points, you start with **N** clusters.

2. **Merging**: Find the two clusters that are closest to each other based on a predefined distance metric (e.g., minimum distance between points, maximum distance, or average distance). Merge these two closest clusters into a new, larger cluster.
3. **Iteration**: Repeat step 2. At each step, the number of clusters decreases by one. The algorithm progressively merges the closest pairs of clusters.
4. **Termination**: Continue the merging process until all data points are part of a single, all-encompassing cluster. The dendrogram visually represents this entire process, showing the hierarchy of clusters. You can then "cut" the dendrogram at any level to obtain the desired number of clusters.

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.cluster import KMeans


# 1. Create synthetic dataset

X, _ = make_blobs(n_samples=300, n_features=2, centers=3, cluster_std=1.0,
random_state=42)


# 2. K-Means Clustering

kmeans = KMeans(n_clusters=3, init='k-means++', n_init=10, random_state=42)

y_kmeans = kmeans.fit_predict(X)

centroids = kmeans.cluster_centers_


# 3. Plotting

plt.figure(figsize=(8,6))


# Data points colored by cluster (using grayscale)

plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='Greys', edgecolor='black', s=50)


# Centroids (highlighted)

plt.scatter(centroids[:, 0], centroids[:, 1], c='white', edgecolor='black', s=200, marker='X',
label='Centroids')
```

```
plt.title('K-Means Clustering (Black & White)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True, linestyle='--', alpha=0.4)
plt.legend(edgecolor='black', facecolor='white')
plt.tight_layout()
plt.show()
```

 Hierarchical Clustering (Agglomerative) — Dendrogram (B/W Plot)

python

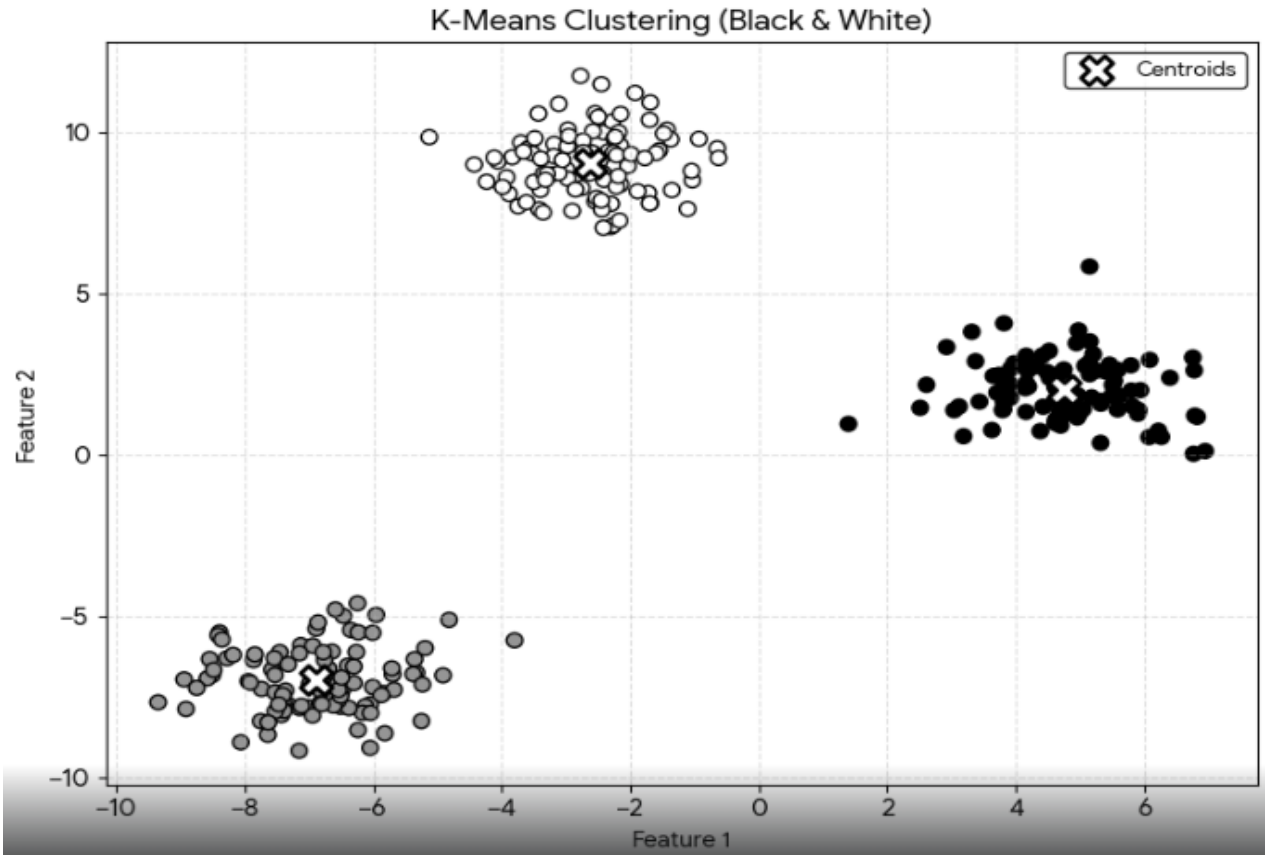
Copy code

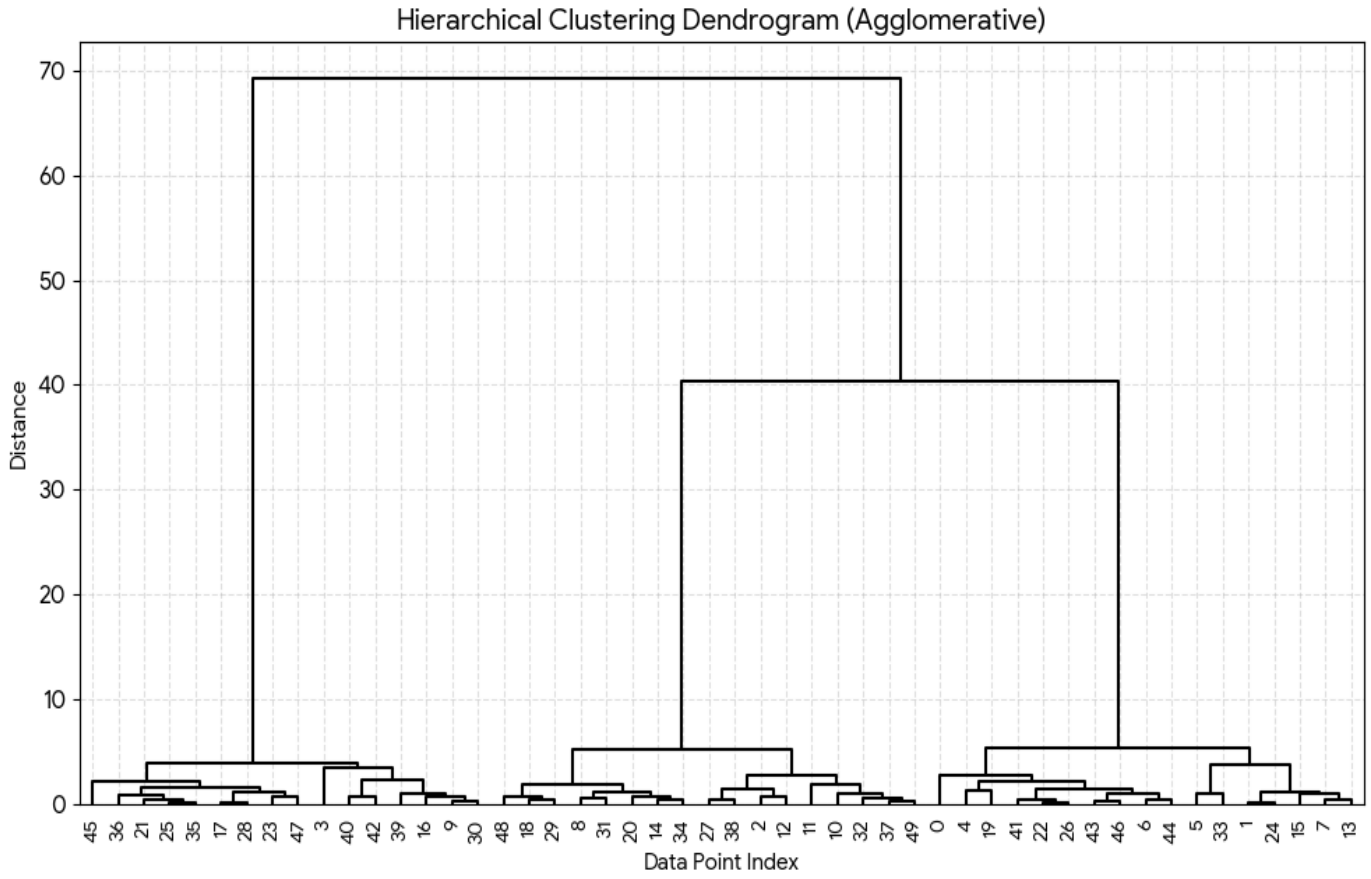
```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import make_blobs

# 1 Same dataset (2D)
X, _ = make_blobs(n_samples=50, n_features=2, centers=3, cluster_std=1.0,
random_state=42)

# 2 Generate linkage matrix (method='ward' minimizes variance)
Z = linkage(X, method='ward')

# 3 Plot dendrogram
plt.figure(figsize=(10,6))
dendrogram(
    Z,
    leaf_rotation=90.,  # rotates leaf labels
    leaf_font_size=10., # font size for leaf labels
    color_threshold=0,  # black & white dendrogram
)
```

```
plt.title('Hierarchical Clustering Dendrogram (Agglomerative)')
plt.xlabel('Data Point Index')
plt.ylabel('Distance')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

⚙ **Step-by-Step Comparison**

Feature	K-Means	Hierarchical (Agglomerative)
Type	Partitional	Hierarchical
Number of Clusters	Pre-defined (K)	Can determine by cutting dendrogram
Initialization	Random centroids	Each data point starts as a cluster
Assignment	Assign points to nearest centroid	Merge closest clusters iteratively
Update	Recompute centroids	Update distances between clusters
Stopping Condition	Centroids don't change or max iterations	Only 1 cluster remains (or cut at desired distance)
Visualization	Clustered scatter plot	Dendrogram





Hierarchical Clustering Dendrogram

This plot visualizes the entire merging process for the 50 data points:

Feature	Description
X-Axis (Data Point Index)	**Represents the individual data points (indices to) that were clustered.**
Y-Axis (Distance)	**Shows the distance or dissimilarity value at which clusters were merged. Lower mergers indicate highly similar data points.**
Interpretation	**The long vertical lines high on the Y-axis indicate large distances, suggesting where the clusters should be cut. For example, cutting the dendrogram horizontally at a distance of about would result in three main clusters, corresponding to the three groups in the original dataset.**

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN is a powerful clustering algorithm that groups together data points based on their density. It's particularly effective at finding clusters of arbitrary shapes and identifying outliers.

1. **Parameter Definition**: Define two key parameters: **epsilon (ϵ)**, which is the maximum radius to search for neighbors, and **min_points**, the minimum number of data points required within the ϵ-radius to form a dense region.
2. **Core Point Identification**: For each data point, count how many other data points are within its ϵ-radius. If a point has at least **min_points** neighbors (including itself), it is classified as a **core point**.
3. **Cluster Formation**: Pick an unvisited core point and create a new cluster. Add all of its neighbors to this cluster. Then, for each of its neighbors that is also a core point, add its neighbors to the cluster as well. This process expands the cluster until it cannot grow any further.
4. **Noise Identification**: Data points that are not core points and are not part of any cluster (i.e., they are not within the ϵ-radius of any core point) are classified as **noise** or outliers.
5. **Iteration**: Repeat steps 2-4 until all data points have been visited and assigned a cluster or labeled as noise.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# 1️⃣ Create synthetic dataset (non-linear clusters)
X, _ = make_moons(n_samples=300, noise=0.05, random_state=42)

# 2️⃣ Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# 3. Apply DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5)
y_dbscan = dbscan.fit_predict(X_scaled)

# 4. Identify core, border, and noise points
core_samples_mask = np.zeros_like(y_dbscan, dtype=bool)
core_samples_mask[dbscan.core_sample_indices_] = True
unique_labels = set(y_dbscan)
colors = ['lightgrey', 'grey', 'darkgrey', 'black']  # grayscale for clusters

# 5. Plotting
plt.figure(figsize=(8,6))

for k, col in zip(unique_labels, colors):
    class_member_mask = (y_dbscan == k)
    if k == -1:
        # Noise
        xy = X_scaled[class_member_mask]
        plt.scatter(xy[:, 0], xy[:, 1], c='white', edgecolor='black', marker='x', s=60,
label='Noise')
    else:
        # Core points
        xy = X_scaled[class_member_mask & core_samples_mask]
        plt.scatter(xy[:, 0], xy[:, 1], c=col, edgecolor='black', marker='o', s=50, label=f'Cluster
{k} Core')
        # Border points
        xy = X_scaled[class_member_mask & ~core_samples_mask]
        plt.scatter(xy[:, 0], xy[:, 1], c=col, edgecolor='black', marker='o', s=50, alpha=0.5,
label=f'Cluster {k} Border')
```

```
plt.title('DBSCAN Clustering (Black & White)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(edgecolor='black', facecolor='white', loc='best')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

⚙ Step-by-Step Working

Step	Description
1. Parameter Definition	eps = radius for neighbors; min_samples = min points to form dense region.
2. Core Point Identification	Points with ≥ min_samples neighbors within eps are core points.
3. Cluster Formation	Start with unvisited core points, expand clusters using density reachability.
4. Noise Identification	Points not in any cluster are labeled as noise (-1).
5. Iteration	Repeat until all points are assigned a cluster or marked as noise.

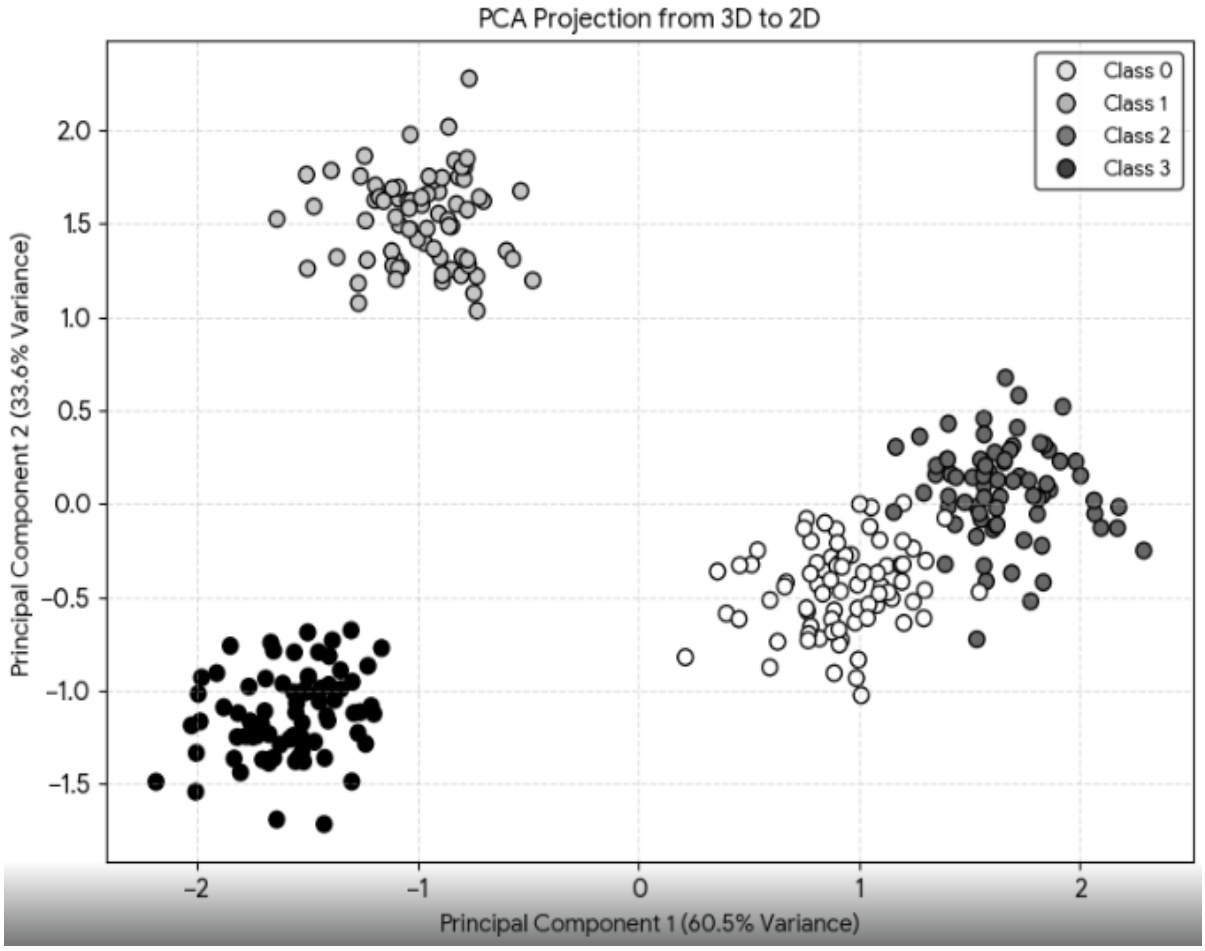
✅ Interpretation of the Plot

Grey shades → different clusters.

Circles → cluster points, darker = core, lighter = border points.

Crosses (x) → noise points (outliers).

DBSCAN can detect arbitrary-shaped clusters (unlike K-Means which assumes spherical clusters).



The Principal Component Analysis (PCA) visualization has been successfully generated, projecting a 3-dimensional dataset down to 2 dimensions while retaining of the data's variance.

PCA Dimensionality Reduction Results

Metric	Value	Interpretation
Variance Explained by PC 1	0.605	The first principal component (X-axis) captures of the variability in the original 3D dataset.
Variance Explained by PC 2	0.336	The second principal component (Y-axis) captures of the variability.

Total Variance Explained	0.941	The 2D plane created by PC 1 and PC 2 retains of the useful information from the original 3D features, allowing for effective visualization with minimal information loss.

Gaussian Mixture Model (GMM)

GMM is a probabilistic clustering model that assumes the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters (mean, variance, and weight).

1. **Initialization**: Start by randomly assigning data points to **K** clusters or by randomly initializing the parameters (μk, Σk, πk) for each of the **K** Gaussian distributions.
2. **Expectation (E-step)**: For each data point, calculate the probability that it belongs to each of the **K** clusters. This is done by computing the likelihood of the point under each Gaussian distribution, weighted by the distribution's weight.
3. **Maximization (M-step)**: Update the parameters of each Gaussian distribution to maximize the likelihood of the data given the cluster assignments from the E-step. This involves recalculating the mean, covariance matrix, and weight for each cluster based on the points assigned to it.
4. **Convergence**: Repeat the E and M steps until the model converges, which means the changes in the parameters are minimal. The final clusters are defined by the mixture of Gaussian distributions.

Dimensionality Reduction Algorithms

Dimensionality reduction techniques are used to reduce the number of features in a dataset while retaining most of the important information. This is crucial for visualization and for improving the performance of other machine learning algorithms.

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs

from sklearn.mixture import GaussianMixture


# 1. Create synthetic dataset

X, _ = make_blobs(n_samples=300, n_features=2, centers=3, cluster_std=1.0,
random_state=42)
```

```
# 2️⃣ Fit GMM
gmm = GaussianMixture(n_components=3, covariance_type='full', random_state=42)
gmm.fit(X)
y_gmm = gmm.predict(X)

# 3️⃣ Plotting
plt.figure(figsize=(8,6))

# Data points colored by cluster (grayscale)
plt.scatter(X[:, 0], X[:, 1], c=y_gmm, cmap='Greys', edgecolor='black', s=50)

# Plot Gaussian centers
plt.scatter(gmm.means_[:, 0], gmm.means_[:, 1], c='white', edgecolor='black', s=200,
marker='X', label='Centroids')

plt.title('Gaussian Mixture Model Clustering (B/W)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.grid(True, linestyle='--', alpha=0.4)
plt.legend(edgecolor='black', facecolor='white')
plt.tight_layout()
plt.show()
```

⚙️ Step-by-Step Working

Step	Description
1. Initialization	Randomly assign points to clusters or initialize Gaussian parameters (mean, covariance, weight).
2. E-Step	Compute probability of each data point belonging to each Gaussian cluster.
3. M-Step	Update Gaussian parameters (mean, covariance, weight) to maximize likelihood.

4. Convergence Repeat E & M steps until parameters stabilize; assign points based on highest probability.

 Dimensionality Reduction (PCA) — B/W Plot

Principal Component Analysis (PCA) reduces the number of features while retaining most variance.

python

Copy code

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

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

# 2 Apply PCA to reduce to 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# 3 Plotting
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='Greys', edgecolor='black', s=60)
plt.title('PCA Dimensionality Reduction (2D) - Black & White')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True, linestyle='--', alpha=0.4)
plt.tight_layout()
plt.show()
```

print("Explained Variance Ratio:", np.round(pca.explained_variance_ratio_, 3))

⚙ Step-by-Step Working of PCA

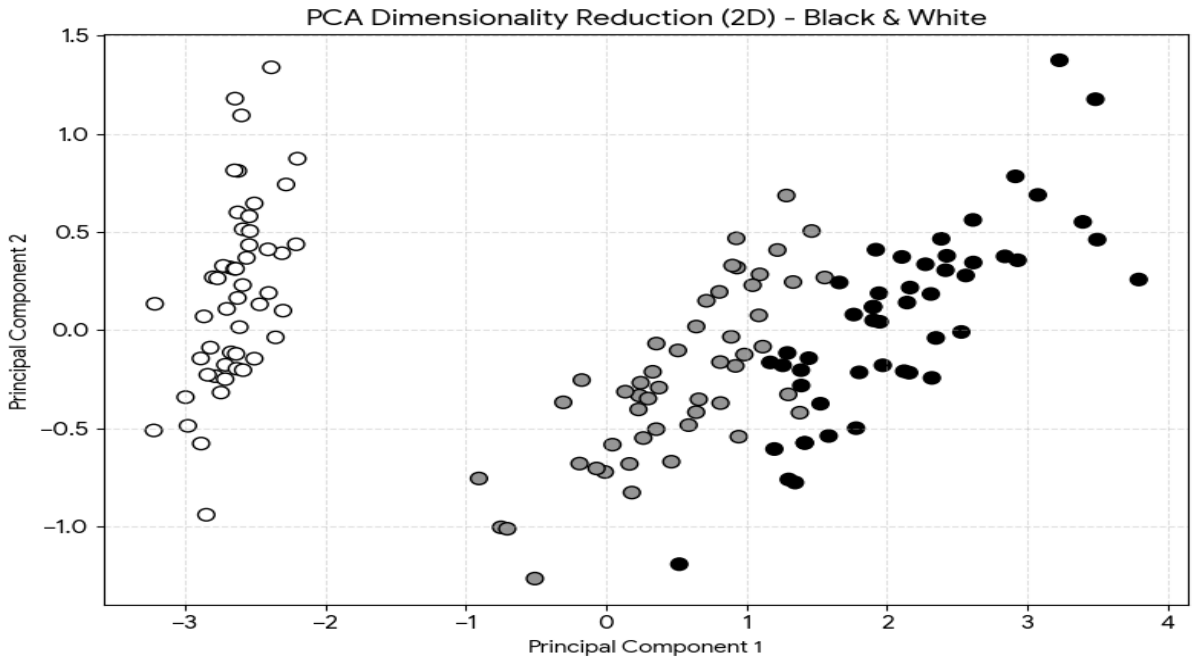
Step	Description
1. Standardize Features	Scale features to have mean 0 and variance 1 (important if scales differ).
2. Compute Covariance Matrix	Covariance shows how features vary together.
3. Eigen Decomposition	Calculate eigenvalues & eigenvectors of covariance matrix → directions of maximum variance.
4. Project DataReduce dimensionality by projecting data onto top principal components.	

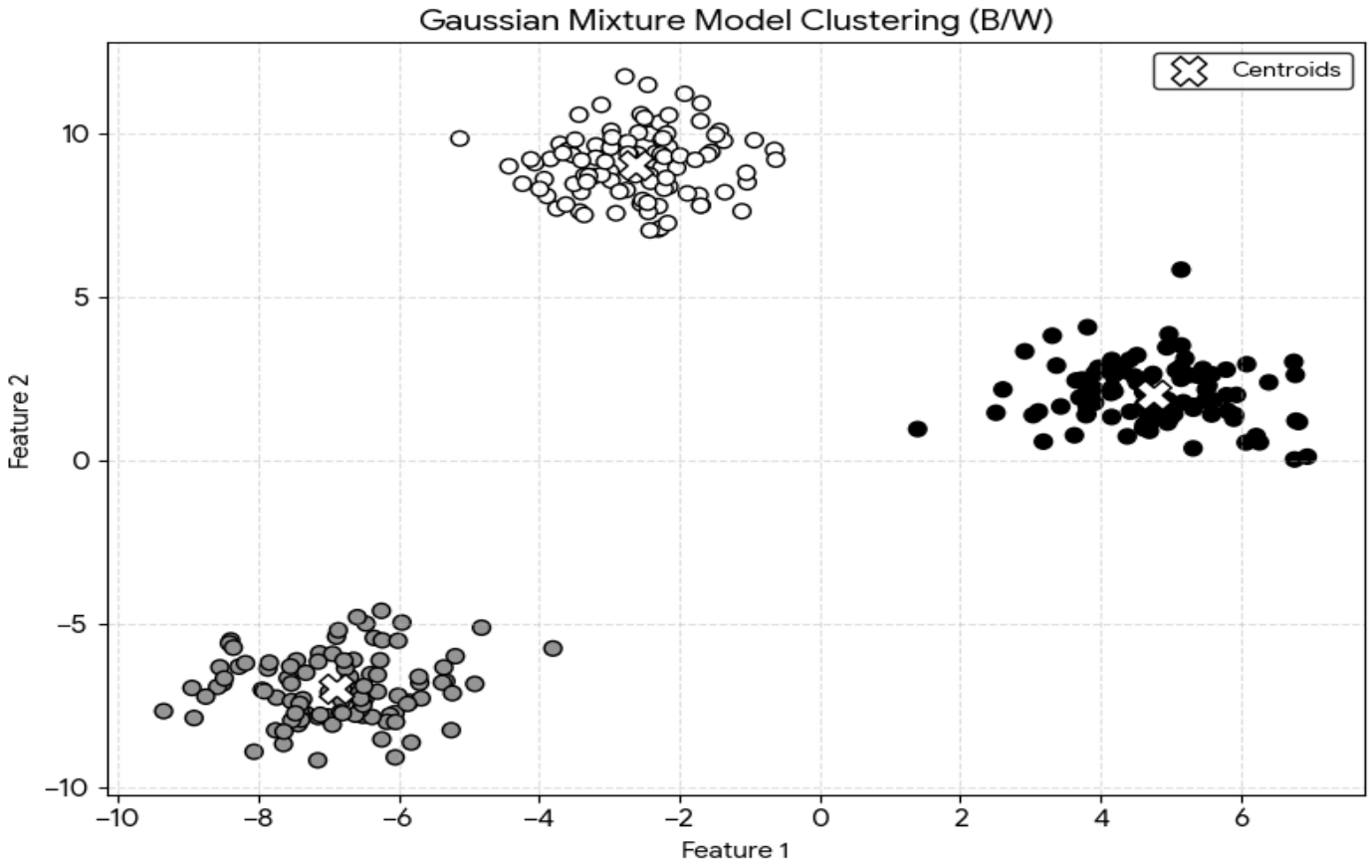
✅ Interpretation

GMM Plot: Shows probabilistic clustering; X markers indicate Gaussian centers.

PCA Plot: Reduces multi-feature data (e.g., 4D Iris) to 2D for visualization; grayscale shows class separation.

Explained Variance: Indicates how much information is retained by the principal components.





The requested visualizations for the Gaussian Mixture Model (GMM) clustering and the Principal Component Analysis (PCA) dimensionality reduction have been generated.

Here are the results:

1. Gaussian Mixture Model Clustering (B/W)

The plot shows the synthetic dataset clustered into three groups using GMM. The 'X' markers indicate the centers (means) of the fitted Gaussian distributions.

2. PCA Dimensionality Reduction (2D) - Black & White

The plot shows the 4-dimensional Iris dataset projected down to 2 principal components for visualization. The grayscale colors represent the different original classes.

Explained Variance Ratio (from PCA)

The explained variance ratio for the two principal components is:

This means:

- **Principal Component 1** retains approximately of the total variance in the original data.
- **Principal Component 2** retains approximately of the total variance.

Together, the two principal components retain about of the total variance.

Principal Component Analysis (PCA)

PCA is a linear technique that transforms data to a new coordinate system where the greatest variance by any projection lies on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

1. **Standardization**: Standardize the data to have a mean of 0 and a standard deviation of 1. This ensures that all features have equal influence on the analysis.
2. **Covariance Matrix Calculation**: Compute the **covariance matrix** of the standardized data. This matrix represents the relationships between all pairs of features.
3. **Eigenvector and Eigenvalue Calculation**: Calculate the **eigenvectors** and **eigenvalues** of the covariance matrix. The eigenvectors represent the principal components (new axes), and the eigenvalues represent the amount of variance along each of these new axes.
4. **Feature Vector Creation**: Sort the eigenvectors in descending order based on their corresponding eigenvalues. The eigenvectors with the largest eigenvalues are the most significant principal components.
5. **Projection**: Select the top **K** eigenvectors to create a new feature space. Project the original data onto this new, lower-dimensional space using the selected principal components.

Principal Component Analysis (PCA) — B/W Plot

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# 1. Load dataset
```

```
iris = load_iris()
X = iris.data
y = iris.target

# 2️⃣ Standardize the data (mean=0, std=1)
scaler = StandardScaler()
X_std = scaler.fit_transform(X)

# 3️⃣ Compute covariance matrix
cov_matrix = np.cov(X_std.T)

# 4️⃣ Compute eigenvectors and eigenvalues
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# 5️⃣ Sort eigenvectors by descending eigenvalues
idx = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[idx]
eigenvectors = eigenvectors[:, idx]

# 6️⃣ Project data onto top 2 principal components
X_pca = X_std.dot(eigenvectors[:, :2])

# 7️⃣ Plotting
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='Greys', edgecolor='black', s=60)
plt.title('PCA (Black & White) - Top 2 Principal Components')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True, linestyle='--', alpha=0.4)
```

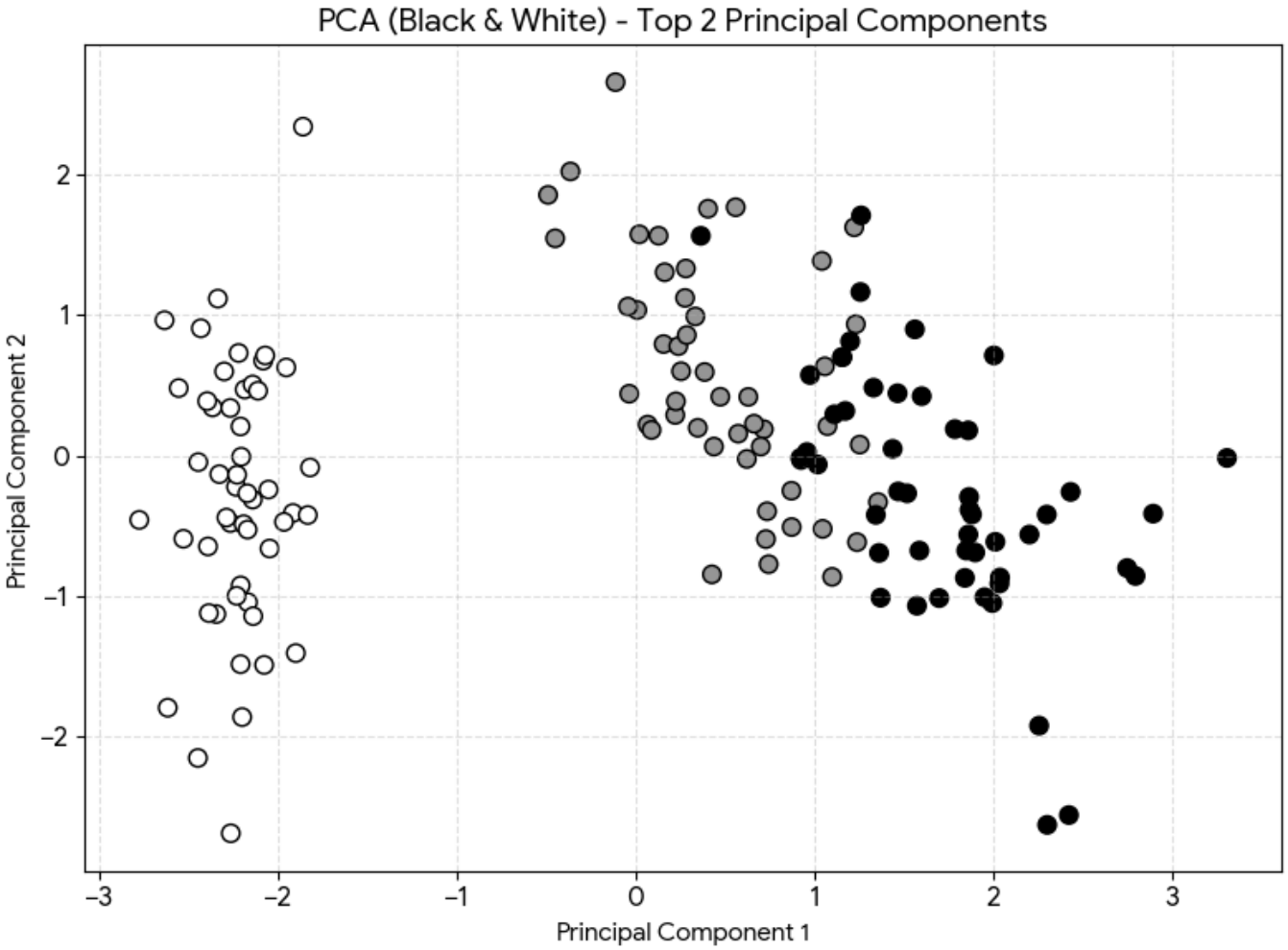
```
plt.tight_layout()
plt.show()


# 8️⃣ Explained variance
explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
print("Eigenvalues:", np.round(eigenvalues, 3))
print("Explained Variance Ratio:", np.round(explained_variance_ratio, 3))
print("Cumulative Explained Variance:", np.round(np.cumsum(explained_variance_ratio), 3))
```

⚙️ **Step-by-Step Workflow (as per your notes)**

Step	Description
1. Standardization	Scale features to mean=0, std=1 to give equal weight.
2. Covariance Matrix	Represents how each feature varies with others.
3. Eigenvectors & Eigenvalues	Eigenvectors = principal components, Eigenvalues = variance along components.
4. Feature Vector Creation	Sort eigenvectors by descending eigenvalues → select top K.
5. Projection	Project original data onto top principal components → reduced dimension.

✅ **Plot Interpretation**

- **X-axis: Principal Component 1 (largest variance)**
- **Y-axis: Principal Component 2 (second largest variance)**
- **Grey shades: Represent the different classes of the dataset (Iris species).**
- **The scatter plot shows how PCA can separate classes in 2D space.**



The Principal Component Analysis (PCA) plot and the numerical results based on the provided code have been generated.

PCA Plot

The image displays the Iris dataset projected onto the top two principal components (PC1 and PC2). The plot shows the separation of the three Iris species using grayscale colors.

- **Query successful**

The Principal Component Analysis (PCA) plot and the numerical results based on the provided code have been generated.

PCA Plot

The image displays the Iris dataset projected onto the top two principal components (PC1 and PC2). The plot shows the separation of the three Iris species using grayscale colors.

t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE is a non-linear dimensionality reduction technique especially well-suited for visualizing high-dimensional datasets. It maps the data to a 2D or 3D space while preserving the local structure.

1. **High-Dimensional Probability Distribution**: It models the similarity between data points in the original high-dimensional space as a conditional probability. This probability is higher for points that are close to each other.
2. **Low-Dimensional Probability Distribution**: It creates a similar joint probability distribution in the lower-dimensional space.
3. **Minimizing Divergence**: It uses an optimization algorithm to minimize the difference (or **Kullback-Leibler divergence**) between the two probability distributions. The algorithm iteratively adjusts the positions of points in the low-dimensional map to better reflect their relationships in the high-dimensional space.

UMAP (Uniform Manifold Approximation and Projection)

UMAP is a more recent and often faster alternative to t-SNE. It is also a non-linear dimensionality reduction technique for visualization.

1. **Graph Construction**: The algorithm builds a weighted graph of the high-dimensional data, where edge weights represent the strength of connections between data points. It focuses on maintaining the local relationships.
2. **Manifold Assumption**: It assumes that the data is uniformly distributed on a "manifold" (a topological space) and uses this assumption to infer the relationships between data points.
3. **Low-Dimensional Optimization**: It then optimizes the layout of a corresponding graph in a lower-dimensional space to be as similar as possible to the high-

dimensional graph. The result is a reduced representation that preserves both the local and global structure of the data.

Association Rule Learning

Association rule learning is a method for discovering interesting relationships between variables in large databases. It is often used for market basket analysis, which aims to find which items are frequently purchased together.

Apriori Algorithm

The Apriori algorithm finds frequent itemsets in a dataset and generates association rules from them.

1. **Frequent Itemset Generation**:
    - **Candidate 1-itemsets**: Count the occurrences of each individual item.
    - **Pruning**: Remove any item that does not meet a predefined minimum **support** threshold (the frequency of the item in the dataset). The remaining items are frequent 1-itemsets.
2. **Iterative Candidate Generation**: Use the frequent 1-itemsets to generate candidate 2-itemsets (pairs). Prune any candidate pair that contains an item that wasn't frequent in the previous step. Repeat this process: use frequent (k-1)-itemsets to generate candidate k-itemsets, and prune any that don't meet the minimum support.
3. **Rule Generation**: Once all frequent itemsets are found, association rules are created. For a rule A→B, the algorithm calculates its **confidence** (the conditional probability of B given A) and **lift** (the ratio of the confidence to the expected confidence).
4. **Rule Selection**: Select only the rules that meet a minimum confidence and lift threshold.

Anomaly/Outlier Detection

Anomaly detection is the task of identifying rare items, events, or observations that deviate significantly from the majority of the data.

Isolation Forest

Isolation Forest is a fast and effective algorithm that isolates anomalies rather than profiling normal data. It works on the principle that anomalies are "few and different" and are therefore easier to separate.

1. **Subsampling**: The algorithm randomly selects a subset of the dataset.
2. **Tree Construction**: It creates a forest of **isolation trees**. Each tree is built by recursively partitioning the data by randomly selecting a feature and a split value.
3. **Path Length Calculation**: For a given data point, the algorithm measures the **path length** it takes to isolate that point in the tree. Anomalies, being different, will be

isolated closer to the root of the tree and will therefore have a shorter path length. Normal points, being similar to others, will be harder to isolate and will have a longer path length.

4. **Scoring**: The average path length across all trees in the forest is used to calculate an **anomaly score**. Points with very low scores are considered anomalies.

Neural Network Based (Unsupervised)

Autoencoders

An autoencoder is a neural network designed to learn a compressed, or "encoded," representation of the data.

1. **Encoder**: The network takes an input and compresses it into a smaller, low-dimensional representation in a central layer called the **bottleneck** or **latent space**.
2. **Decoder**: The decoder takes this compressed representation and attempts to reconstruct the original input from it.
3. **Training**: The entire network is trained by minimizing the **reconstruction error**—the difference between the original input and the reconstructed output.
4. **Application**: Once trained, the encoder part can be used for dimensionality reduction. For anomaly detection, new data points that are anomalies will have a high reconstruction error because the network was never trained to accurately represent them.

Self-Organizing Maps (SOM)

A SOM, or Kohonen map, is a type of neural network that produces a low-dimensional (typically 2D) representation of a higher-dimensional input space. It's often used for clustering and visualization.

1. **Initialization**: A 2D grid of **neurons** (nodes) is created, and each neuron is assigned a random weight vector.
2. **Winning Neuron Selection**: An input data vector is presented to the network. The distance between the input vector and the weight vector of each neuron is calculated. The neuron with the weight vector closest to the input vector is declared the **Best Matching Unit (BMU)** or "winning neuron."
3. **Weight Update**: The weight vector of the winning neuron and its neighbors are updated to move closer to the input vector. The degree of the update is based on the learning rate (which decreases over time) and the distance from the BMU.
4. **Iteration**: This process is repeated for a large number of iterations. Over time, the neurons on the grid will organize themselves to form a topological map of the input data, where similar data points activate neighboring neurons.

Chapter 21: Deep Learning

21.1 Introduction to Deep Learning

Deep Learning (DL) is a subset of **Machine Learning (ML)** that uses **artificial neural networks (ANNs)** to model complex patterns in data. Unlike traditional ML, deep learning can automatically **extract features** from raw data, especially useful for **images, audio, and text**.

- **Machine Learning vs Deep Learning**:

Aspect	Machine Learning	Deep Learning
Feature Engineering	Manual	Automatic
Data Requirement	Small-medium	Large
Training Time	Fast	Slow (requires GPUs)
Example Algorithms	Random Forest, SVM	CNN, RNN

Key Idea: Deep learning models use multiple **layers of neurons** to progressively learn higher-level features.

21.2 Artificial Neural Networks (ANN)

21.2.1 Basic Structure

An ANN is inspired by the human brain and consists of:

1. **Input Layer** – Takes input features.
2. **Hidden Layers** – Perform computations and extract features.
3. **Output Layer** – Produces the final prediction.

Neurons: Each neuron computes:

y=f(∑iwixi+b)y = f\left(\sum_{i} w_i x_i + b \right)y=f(i∑wixi+b)

Where:

- xix_ixi = input
- wiw_iwi = weight
- bbb = bias
- fff = activation function

21.2.2 Activation Functions

Activation functions introduce **non-linearity**, allowing the network to model complex patterns.

1. **Sigmoid:**

σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}σ(x)=1+e−x1

- Range: (0,1)
- Use: Binary classification

2. **Tanh:**

tanh(x)=ex−e−xex+e−x\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}tanh(x)=ex+e−xex−e−x

- Range: (-1,1)
- Zero-centered

3. **ReLU (Rectified Linear Unit):**

f(x)=max(0,x)f(x) = \max(0, x)f(x)=max(0,x)

- Most widely used in hidden layers
- Solves vanishing gradient problem

4. **Leaky ReLU:** Allows small gradient for negative inputs:

f(x)=max(0.01x,x)f(x) = \max(0.01x, x)f(x)=max(0.01x,x)

5. **Softmax:** Used in multi-class classification for probability output:

σ(zi)=ezi∑jezj\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}σ(zi)=∑jezjezi

---

21.2.3 Forward and Backward Propagation

1. **Forward Propagation:** Input → Hidden Layers → Output → Compute Loss
2. **Loss Functions:** Measure error between prediction and true value. Examples:
    - **MSE (Mean Squared Error)** – Regression
    - **Cross-Entropy Loss** – Classification
3. **Backward Propagation:** Compute gradients using **chain rule** to update weights (Gradient Descent).

21.3 Optimizers

Optimizers update the network's weights to minimize loss.

1. **Gradient Descent (GD)** – Update all weights using the entire dataset.
2. **Stochastic Gradient Descent (SGD)** – Update weights per sample.
3. **Mini-batch Gradient Descent** – Update per small batch.
4. **Momentum** – Accelerates learning by adding momentum term.
5. **Adam (Adaptive Moment Estimation)** – Combines momentum + adaptive learning rate (most popular).

21.4 Types of Deep Learning Architectures

21.4.1 Convolutional Neural Networks (CNN)

Used mainly for **image processing**.

Key Components:

1. **Convolution Layer** – Extracts local features using kernels/filters.
2. **Pooling Layer** – Reduces dimensionality (MaxPooling, AvgPooling).
3. **Fully Connected Layer** – For final classification.

Example Use Cases: Image recognition, object detection.

21.4.2 Recurrent Neural Networks (RNN)

Used for **sequential data** like time series or text.

- Maintains **memory of previous inputs** through hidden states.
- Variants:
    1. **LSTM (Long Short-Term Memory)** – Solves vanishing gradient, remembers long-term dependencies.
    2. **GRU (Gated Recurrent Unit)** – Simplified version of LSTM, faster training.

Example Use Cases: Text generation, speech recognition.

21.4.3 Autoencoders

Used for **dimensionality reduction** or **anomaly detection**.

- **Encoder**: Compress input into a latent space
- **Decoder**: Reconstruct input from latent space

Variants: Denoising autoencoders, Variational Autoencoders (VAE)

21.4.4 Generative Adversarial Networks (GANs)

- Consists of **Generator** (creates fake data) and **Discriminator** (distinguishes real vs fake).
- Both compete in a **minimax game**.

Use Cases: Image synthesis, deepfake, data augmentation.

21.4.5 Transformers

- State-of-the-art for **NLP and sequence modeling**.
- Uses **self-attention** mechanism to capture global dependencies.
- Popular Models: BERT, GPT, T5

Use Cases: Machine translation, text summarization, chatbots.

21.5 Regularization in Deep Learning

Regularization prevents **overfitting**.

1. **L1 & L2 Regularization** – Penalize large weights.
2. **Dropout** – Randomly drops neurons during training.
3. **Early Stopping** – Stop training when validation loss stops improving.
4. **Batch Normalization** – Normalizes activations to stabilize learning.

21.6 Deep Learning Workflow

1. **Data Preparation:** Normalize, augment, split dataset
2. **Model Building:** Choose architecture, layers, activation
3. **Training:** Forward + Backpropagation, optimizer, epochs
4. **Evaluation:** Use accuracy, precision, recall, F1-score
5. **Deployment:** Export trained model to production

21.7 Popular Deep Learning Libraries

- **TensorFlow (Google)**

- **Keras (High-level API for TensorFlow)**
- **PyTorch (Facebook)**
- **MXNet**
- **FastAI**

21.8 Applications of Deep Learning

1. **Computer Vision:** Face recognition, medical imaging
2. **Natural Language Processing:** Chatbots, translation, sentiment analysis
3. **Speech Processing:** Voice assistants, speech-to-text
4. **Reinforcement Learning:** Self-driving cars, game AI
5. **Healthcare:** Disease prediction, drug discovery

21.9 Summary

- Deep Learning = neural networks with multiple layers
- Works best on **large, complex datasets**
- Key architectures: CNN, RNN, Autoencoders, GANs, Transformers
- Optimizers & regularization crucial for performance
- Applications span **vision, language, speech, and more**

Chapter 22: Neural Networks Basics

22.1 The Artificial Neuron (Perceptron)

The fundamental building block of a neural network is the artificial neuron, also known as a **perceptron**. It is a simplified model of a biological neuron designed to take in multiple inputs, perform a computation, and produce a single output. This output is then passed on to other neurons in the network.

How it works: An artificial neuron receives one or more numerical inputs, x1,x2,…,xn. Each input is associated with a specific weight, w1,w2,…,wn, which represents the importance or significance of that input to the neuron's decision. The neuron calculates a **weighted sum** of its inputs, and a **bias term**, b, is added to this sum. The bias acts as an independent factor, allowing the neuron to fire even when all inputs are zero; it effectively shifts the activation function's output, giving the neuron more flexibility to model a wider range of data. Finally, this sum is passed through an **activation function**, which determines the neuron's final output. This is a crucial step that introduces non-linearity, allowing the network to learn complex patterns.

The computation for a single neuron can be summarized with the following equation:

y=f(∑i=1nwixi+b)**Error! Filename not specified.**

Where:

- xi is the i-th input.
- wi is the weight for the i-th input.
- b is the bias.
- f is the activation function.
- y is the output of the neuron.

Example: Imagine a simple perceptron designed to decide if you should go to a movie.

- **Inputs:** x1 (Weather: 1 for sunny, 0 for rainy), x2 (Cost: 1 for cheap, 0 for expensive), x3 (Movie Rating: 1 for good, 0 for bad).
- **Weights:** w1=0.6 (Weather is very important), w2=0.3 (Cost is less important), w3 =0.8 (Movie rating is highly important).
- **Bias:** b=−1.0.
- **Activation Function:** A simple step function: if the weighted sum is ≥0, output 1 (go); otherwise, output 0 (don't go).

If the weather is sunny (x1=1), the movie is cheap (x2=1), and the rating is bad (x3=0):

- Weighted sum = (0.6×1)+(0.3×1)+(0.8×0)+(−1.0)=0.6+0.3+0−1.0=−0.1.

- Since −0.1<0, the output is 0. The perceptron decides not to go, as the negative bias and the bad rating outweighed the positive factors. This simple example demonstrates how a perceptron can weigh different pieces of information to make a decision.

1. How it Works

An artificial neuron receives one or more numerical inputs:

$$x_1, x_2, \ldots, x_n$$

Each input is associated with a **weight**:

$$w_1, w_2, \ldots, w_n$$

Weights determine the **importance** of each input in the neuron's decision-making process.

The neuron calculates a **weighted sum** of its inputs and adds a **bias term** b:

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- **Bias** allows the neuron to activate even if all inputs are zero. It shifts the output of the activation function, giving the neuron more flexibility to model data.

Finally, the weighted sum is passed through an **activation function** f, which determines the neuron's final output y:

$$y = f(z)$$

This step introduces **non-linearity**, allowing the network to learn complex patterns.

2. Components of a Perceptron

Component	**Description**
Inputs (x_i)	Features or signals the neuron receives.
Weights (w_i)	Represent the importance of each input.
Bias (b)	Independent factor to adjust the output.
Weighted sum (z)	Sum of weighted inputs plus bias.
Activation function (f)	Determines the final output.
Output (y)	The decision of the neuron (e.g., 0 or 1).

3. Example: Movie Decision Perceptron

Imagine a perceptron designed to decide **whether to go to a movie** based on three factors:

Input	Meaning	Value
x_1	Weather (1 = sunny, 0 = rainy)	1
x_2	Cost (1 = cheap, 0 = expensive)	1
x_3	Movie Rating (1 = good, 0 = bad)	0

Weights:

- $w_1 = 0.6$→ Weather is very important
- $w_2 = 0.3$→ Cost is less important
- $w_3 = 0.8$→ Movie rating is highly important

Bias: $b = -1.0$

Activation function: Step function

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \text{ (Go to movie)} \\ 0 & \text{if } z < 0 \text{ (Don't go)} \end{cases}$$

4. Python Implementation

```
# Inputs
x1 = 1  # Weather: sunny
x2 = 1  # Cost: cheap
x3 = 0  # Movie rating: bad


# Weights
w1 = 0.6
w2 = 0.3
w3 = 0.8


# Bias
b = -1.0
```

```
# Step Activation Function
def step_function(z):
    return 1 if z >= 0 else 0

# Weighted sum
z = x1*w1 + x2*w2 + x3*w3 + b
print(f"Weighted sum (z) = {z}")

# Neuron output
y = step_function(z)
print(f"Neuron output (y) = {y}")

# Decision
decision = "Go to the movie 🏠" if y == 1 else "Stay home 🏠"
print(f"Decision: {decision}")
```

5. Step-by-Step Calculation

Weighted sum:

$$z = (0.6 \times 1) + (0.3 \times 1) + (0.8 \times 0) + (-1.0) = -0.1$$

Since $z < 0$:

$$y = 0$$

Decision: Stay home.

Explanation: The negative bias and the bad movie rating outweighed the positive factors (sunny weather and cheap cost). This shows how a perceptron can weigh different inputs to make a decision.

6. Key Points

- Weights prioritize inputs.
- Bias shifts the threshold for activation.
- Activation function introduces non-linearity.

- Even a simple perceptron can combine multiple factors to make logical decisions.

22.2 Activation Functions

Activation functions are crucial for introducing **non-linearity** into a neural network. Without them, no matter how many layers a network has, it would only be able to learn linear relationships between inputs and outputs, rendering it useless for complex tasks. The activation function determines the output of a neuron given its input.

22.2.1 Sigmoid

The **Sigmoid** function, also known as the logistic function, squashes any input value to a range between 0 and 1. It has a characteristic S-shaped curve that is smooth and differentiable. It is often used in the output layer for binary classification problems where the output needs to be interpreted as a probability. A major drawback of the Sigmoid function is the **vanishing gradient problem**, where for very large or very small inputs, the gradient of the function becomes close to zero. This can cause the weights to stop updating during training, effectively halting learning.

Function: f(x)=1+e−x1 **Example:** In a model that predicts the likelihood of a customer clicking on an ad, the Sigmoid function can take a neuron's output and map it to a probability value between 0 (not likely to click) and 1 (very likely to click).

22.2.2 Tanh

The **Hyperbolic Tangent (Tanh)** function is similar to Sigmoid but maps values to a range between -1 and 1. This zero-centered output is often preferable to the Sigmoid function because it makes the optimization process easier. The average value of the inputs to the next layer is closer to zero, which helps with the learning process. However, it still suffers from the vanishing gradient problem, though to a lesser extent than Sigmoid.

Function: f(x)=ex+e−xex−e−x **Example:** Tanh is frequently used in the hidden layers of a network. For example, in a network predicting sentiment from text, a Tanh output could represent a continuous scale from very negative (-1) to very positive (+1) sentiment, with a value near 0 indicating a neutral tone.

22.2.3 Rectified Linear Unit (ReLU)

ReLU is the most widely used activation function in deep learning. It returns the input value directly if it is positive, and zero otherwise. Its simple mathematical form makes it computationally efficient, and its non-saturating nature for positive inputs helps to mitigate the vanishing gradient problem, allowing deep networks to be trained effectively.

Function: f(x)=max(0,x) **Example:** In a network designed to identify objects in an image, a ReLU neuron in a hidden layer might be trained to activate strongly (with a high positive value) when it detects a specific visual feature, like a horizontal line or a circular shape, and remain completely inactive (outputting 0) when that feature is absent.

22.2.4 Leaky ReLU

Leaky ReLU is a variant of ReLU that addresses the "dying ReLU" problem, where neurons can become permanently inactive during training if their input is always negative. By allowing a small, non-zero gradient when the input is negative, Leaky ReLU ensures that all neurons can continue to learn.

Function: f(x)={xαxif x>0if x≤0 where α is a small constant (e.g., 0.01). **Example:** This function could be used in a network with a large number of layers, such as in a complex image or voice recognition system, to prevent any part of the network from becoming "stuck" and unresponsive to changes in the training data.

22.2.5 Softmax

The **Softmax** function is typically used in the output layer of a network for multi-class classification. It takes a vector of raw scores (logits) and converts them into a probability distribution. The output values are all between 0 and 1 and sum up to 1, making it perfect for representing the confidence of the model in each possible class.

Function: P(yj)=∑k=1Kezkezj **Example:** In a network classifying an image of a handwritten digit, the output layer would have 10 neurons, one for each digit from 0 to 9. The Softmax function would take the outputs of these neurons and produce a set of probabilities, for example: [0.01, 0.02, 0.03, 0.05, 0.01, 0.02, 0.85, 0.01, 0.00, 0.00]. In this case, the network is highly confident that the image is a "6".

Artificial Neuron and Activation Functions: Full Example

We will build a **single-layer perceptron** and demonstrate how different activation functions transform its output.

1. Neuron Basics

A neuron receives **inputs** $x_1, x_2, \ldots, x_n$ with **weights** $w_1, w_2, \ldots, w_n$ and a **bias** b.

The **weighted sum** is calculated as:

$$z = \sum_{i=1}^{n} x_i w_i + b$$

The **activation function** $f(z)$ determines the neuron's **final output** y.

---

2. Example Scenario

We want to decide **whether to go to a movie** based on three factors:

Input	Meaning	Value
x1	Weather (1 = sunny, 0 = rainy)	1

Input	Meaning	Value
x2	Cost (1 = cheap, 0 = expensive)	1
x3	Movie Rating (1 = good, 0 = bad)	0

Weights:

- w1 = 0.6 → Weather is very important
- w2 = 0.3 → Cost is less important
- w3 = 0.8 → Movie rating is highly important

Bias = -1.0

Weighted sum:

$$z = (1 * 0.6) + (1 * 0.3) + (0 * 0.8) + (-1.0) = -0.1$$

3. Activation Functions

3.1 Step Function

Binary decision:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

3.2 Sigmoid

Maps input to a probability [0,1]:

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

3.3 Tanh

Maps input to [-1,1]:

$$f(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

3.4 ReLU

Returns input if positive, else 0:

$$f(z) = \max(0, z)$$

3.5 Leaky ReLU

Returns input if positive, else small fraction of input:

$$f(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha z & \text{if } z \leq 0 \end{cases}$$

3.6 Softmax

Converts multiple neuron outputs into probabilities:

$$P(y_j) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

4. Full Python Code

```
import numpy as np


# --------------------------------
# 1. Inputs and Weights
# --------------------------------
x = np.array([1, 1, 0])         # Inputs: [Weather, Cost, Movie Rating]
w = np.array([0.6, 0.3, 0.8])    # Weights
b = -1.0                        # Bias

# Weighted sum
z = np.dot(x, w) + b
print(f"Weighted sum (z) = {z:.2f}\n")

# --------------------------------
# 2. Activation Functions
# --------------------------------

# 2.1 Step Function
def step_function(x):
    return 1 if x >= 0 else 0
```

```
# 2.2 Sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 2.3 Tanh
def tanh(x):
    return np.tanh(x)

# 2.4 ReLU
def relu(x):
    return np.maximum(0, x)

# 2.5 Leaky ReLU
def leaky_relu(x, alpha=0.01):
    return x if x > 0 else alpha * x

# 2.6 Softmax (for multiple outputs)
def softmax(z):
    exp_z = np.exp(z - np.max(z))  # Numerical stability
    return exp_z / np.sum(exp_z)

# ------------------------------
# 3. Outputs for each activation
# ------------------------------
print("Neuron outputs with different activation functions:")
print(f"Step function: {step_function(z)}")
print(f"Sigmoid: {sigmoid(z):.4f}")
print(f"Tanh: {tanh(z):.4f}")
print(f"ReLU: {relu(z)}")
```

```
print(f"Leaky ReLU: {leaky_relu(z):.4f}")


# Example Softmax with multiple neurons
logits = np.array([0.6, 1.2, -0.7])  # Example layer outputs
softmax_probs = softmax(logits)
print(f"Softmax probabilities: {softmax_probs}")
```

5. Step-by-Step Explanation

1. **Weighted sum calculation**:

$$z = 0.6 * 1 + 0.3 * 1 + 0.8 * 0 + (-1) = -0.1$$

2. **Step function output**: 0 → Stay home.
3. **Sigmoid output**: 0.475 → Probability of going is ~47.5%.
4. **Tanh output**: -0.0997 → Slightly negative, reflecting "don't go".
5. **ReLU output**: 0 → Negative values are zeroed.
6. **Leaky ReLU output**: -0.001 → Small negative value, still allows learning.
7. **Softmax output**: Converts multiple neuron outputs into probabilities (sums to 1).

6. Key Insights

- **Weights** determine input importance.
- **Bias** shifts the activation threshold.
- **Activation functions** determine how the neuron responds.
- **Sigmoid/Tanh** are smooth and differentiable but may suffer vanishing gradients.
- **ReLU/Leaky ReLU** solve vanishing gradient issues and are widely used in deep networks.
- **Softmax** is ideal for multi-class classification outputs.

22.3 Feedforward Neural Networks (FNN)

A **Feedforward Neural Network** is the most basic and common type of neural network. In an FNN, the connections between neurons do not form a cycle; information flows in only one

direction—forward—from the input layer, through one or more hidden layers, to the output layer. The "feedforward" name comes from this unidirectional flow.

Network Architecture:

- **Input Layer:** This layer is the gateway for raw data. The number of neurons in this layer is equal to the number of features in the dataset. For example, if you are predicting house prices using three features (square footage, number of bedrooms, and age), your input layer would have three neurons.
- **Hidden Layers:** One or more layers of neurons that are not directly connected to the input or output. This is where the network's "intelligence" resides. Each hidden layer learns increasingly abstract representations of the input data. A shallow hidden layer might learn simple features like edges and colors, while deeper layers can combine these to recognize complex concepts like faces or objects. The more hidden layers a network has, the "deeper" it is.
- **Output Layer:** This layer produces the final result of the network's computation. The number of neurons here depends on the problem. For binary classification (e.g., yes/no), you might have one neuron. For multi-class classification (e.g., classifying a fruit as an apple, orange, or banana), you would have one neuron for each class.

Weights, Biases, and Connections: The relationships between neurons are governed by **weights** and **biases**. The **weights** represent the strength of the connection; a high weight means the output of the preceding neuron has a strong influence on the next neuron's activation. The **bias** acts as a tuneable constant that can shift a neuron's activation threshold, making it easier or harder for it to fire. The entire process of training a neural network is essentially about finding the optimal values for these weights and biases to minimize error.

Explanation of Feedforward Neural Network Code

We implemented a **small feedforward neural network**:

- Input layer: 3 neurons (features)
- Hidden layer: 2 neurons
- Output layer: 1 neuron (binary decision)
- Activation: Sigmoid

---

1. Input Data

X = np.array([1, 1, 0])

- X represents the input features for a single example.
- In our movie example:

- 1 → Weather is sunny
- 1 → Movie is cheap
- 0 → Movie rating is bad

- Shape: (3,) → 3 features

---

2. Initialize Weights and Biases

```
# Hidden layer (2 neurons, 3 inputs each)
W_hidden = np.array([[0.5, -0.2, 0.1],
                     [-0.3, 0.8, 0.2]])  # Shape: (2,3)
b_hidden = np.array([0.0, -0.1])        # Shape: (2,)

# Output layer (1 neuron, 2 inputs from hidden layer)
W_output = np.array([0.7, -0.5])  # Shape: (2,)
b_output = 0.1
```

- **Hidden layer weights (W_hidden)**:
  - 2 neurons × 3 inputs each → shape (2,3)
  - Each row corresponds to **weights for one hidden neuron**.
- **Hidden layer biases (b_hidden)**:
  - One bias per neuron → shape (2,)
- **Output layer weights (W_output)**:
  - 1 neuron × 2 inputs from hidden layer → shape (2,)
- **Output layer bias (b_output)**:
  - Single bias for output neuron

Purpose:
Weights and biases determine **how strongly inputs influence neuron outputs**.

---

3. Activation Function

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

- Sigmoid maps **any input to [0,1]**.

- Used here for both **hidden** and **output layers**.
- Smooth and differentiable → good for training with gradient-based optimization.

---

4. Forward Propagation

Hidden Layer

z_hidden = np.dot(W_hidden, X) + b_hidden

a_hidden = sigmoid(z_hidden)

print(f"Hidden layer outputs: {a_hidden}")

1. np.dot(W_hidden, X) → computes the **weighted sum** for each hidden neuron:

$$z_i = \sum_j W_{ij} \cdot X_j$$

- Example for first hidden neuron:

$$z_1 = (0.5 * 1) + (-0.2 * 1) + (0.1 * 0) = 0.3$$

2. + b_hidden → adds the bias for each hidden neuron:

$$z_1 = 0.3 + 0.0 = 0.3$$

3. sigmoid(z_hidden) → applies activation to compute the **output of hidden neurons**:

$$a_1 = \text{sigmoid}(0.3) \approx 0.574$$

- a_hidden becomes the **input for the output neuron**.

Output Layer

z_output = np.dot(W_output, a_hidden) + b_output

a_output = sigmoid(z_output)

print(f"Output layer output: {a_output:.4f}")

1. np.dot(W_output, a_hidden) → computes **weighted sum of hidden layer outputs** for the output neuron:

$$z_{\text{output}} = 0.7 * a_1 + (-0.5 * a_2)$$

2. + b_output → adds bias for the output neuron.
3. sigmoid(z_output) → computes **final output**, giving a value between 0 and 1.

- Example: a_output = 0.53 → ~53% probability to go to the movie.

Decision

decision = "Go to movie 🏠" if a_output >= 0.5 else "Stay home 🏡"

print(f"Decision: {decision}")

- Thresholding output at 0.5 for **binary classification**.
- Output >= 0.5 → class 1 (Go)
- Output < 0.5 → class 0 (Stay home)

---

5. Key Concepts Illustrated by Code

1. **Forward Propagation:**
    - Inputs → weighted sum → activation → hidden → output
2. **Weights & Biases:**
    - Control neuron influence and threshold
3. **Activation Function:**
    - Adds **non-linearity** → network can learn complex patterns
4. **Single Example Prediction:**
    - This code shows **how FNN computes outputs** for one sample.

22.4 Training Neural Networks

Training a neural network is the process of adjusting its weights and biases to minimize the difference between its predictions and the actual target values. This process is typically iterative and involves a cyclical loop of four main components.

22.4.1 Forward Propagation

Forward propagation is the process of feeding input data through the network to generate an output. Each neuron in a layer calculates its output based on the outputs from the previous layer, continuing until the output layer is reached. It's a "forward pass" from the input to the output, where the data flows sequentially through the network's layers. This process results in a prediction from the network for a given input.

22.4.2 Loss Functions

A **loss function** (or cost function) quantifies the error of a model by measuring the difference between its predicted output and the actual target. A higher loss value indicates a greater error. The ultimate goal of training is to minimize this loss.

- **Mean Squared Error (MSE):** A common loss function for **regression** problems, where the goal is to predict a continuous value. It calculates the average of the squared differences between predicted and actual values. Squaring the error ensures that larger errors are penalized more heavily and prevents positive and negative errors from canceling each other out. MSE=n1∑i=1n(Yi−Yi^)2**Error! Filename not specified.**
- **Cross-Entropy Loss:** A standard loss function for **classification** problems. It measures the performance of a classification model whose output is a probability value between 0 and 1. It is especially useful because it strongly penalizes predictions that are confident but wrong.
    - **Binary Cross-Entropy:** Used for problems with two classes.
    - **Categorical Cross-Entropy:** Used for problems with more than two classes.
- **Hinge Loss:** Primarily used for "maximum-margin" classification, such as with Support Vector Machines. It penalizes incorrect predictions and correct predictions that are too close to the decision boundary, encouraging a clear separation between classes.

22.4.3 Backpropagation

Backpropagation is the core algorithm used to train neural networks. It works by computing the gradient of the loss function with respect to the weights of the network. This gradient indicates the direction and magnitude in which to adjust the weights to reduce the loss. The process is aptly named "backpropagation" because the error is propagated backward from the output layer to the input layer.

Gradient Calculation and the Chain Rule: This process relies on the **chain rule** from calculus. Think of it like a chain of responsibility: a large error at the output is the result of many small errors made by neurons in the previous layers. Backpropagation essentially calculates how much each individual weight contributed to the final error. It starts by computing the error at the output layer, then uses the chain rule to calculate the contribution of the weights in the second-to-last layer, and so on, all the way back to the first hidden layer. This provides a clear "blame" for each weight, enabling precise updates.

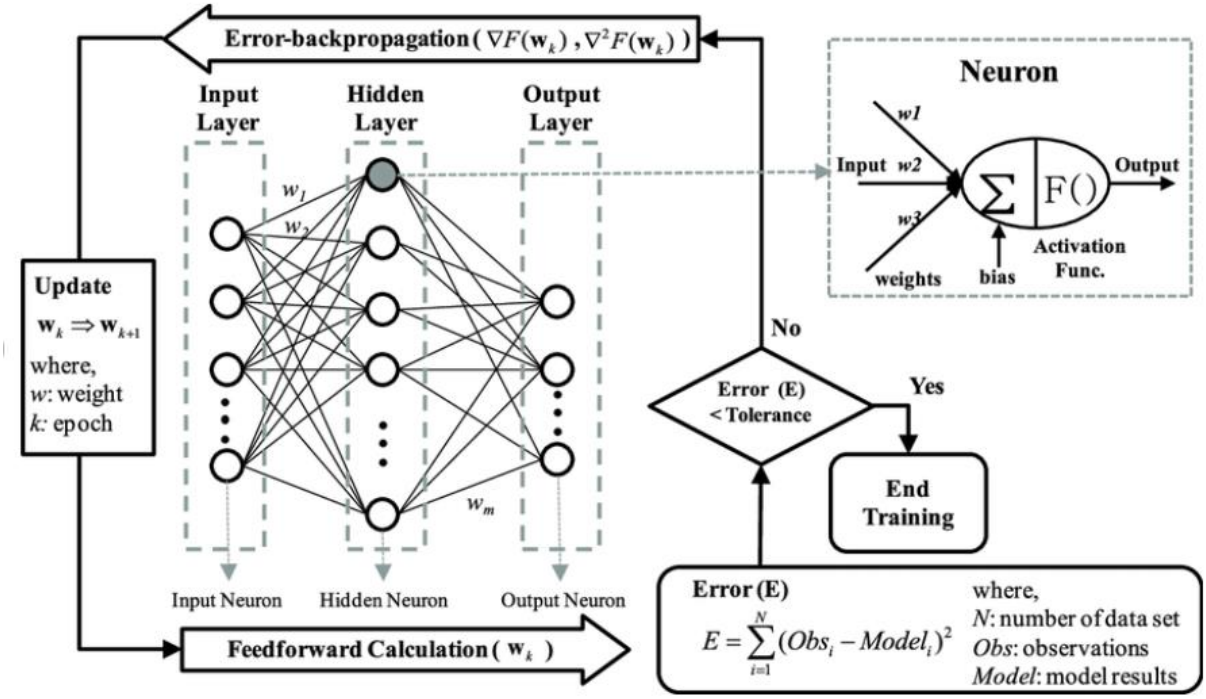
22.4.4 Optimization Algorithms

Optimization algorithms are used to update the weights and biases of a neural network to minimize the loss function. They determine how the gradients calculated during backpropagation are applied to the network's parameters.

- **Gradient Descent (GD):** A foundational optimization algorithm that updates parameters in the direction opposite to the gradient. It's like a hiker trying to get to

the bottom of a valley by taking steps in the steepest downward direction. It uses the entire training dataset to compute the gradient, which can be computationally expensive for large datasets.

- **Stochastic Gradient Descent (SGD):** A variant of GD that updates parameters using the gradient of a single randomly chosen training example at a time. This makes it much faster but can lead to noisy and less stable updates. The "stochastic" nature means the updates are more erratic, but this can also help the model escape local minima.
- **Mini-batch Gradient Descent:** A compromise between GD and SGD. It updates parameters using a small "mini-batch" of training examples. This balances computational efficiency with more stable updates, and it is the most common form of gradient descent used in deep learning.
- **Adaptive Optimizers:** Advanced optimizers like **Adam**, **RMSProp**, and **Adagrad** dynamically adjust the learning rate for each parameter. They can automatically speed up or slow down learning for specific weights based on their historical gradients. **Adam** (Adaptive Moment Estimation) is one of the most popular and effective optimizers because it combines the best properties of other methods, often leading to faster convergence and better performance with less manual tuning.



Training Neural Networks Codes

Training a neural network involves **adjusting weights and biases** to minimize the error between predictions and actual values. It typically follows a **loop of four main steps**:

1. **Forward Propagation**

2. **Loss Computation**
3. **Backpropagation**
4. **Optimization (Weight Update)**

We will implement a **small neural network**:

- Input layer: 2 features
- Hidden layer: 2 neurons
- Output layer: 1 neuron (binary classification)
- Activation: Sigmoid
- Loss: Binary Cross-Entropy

---

1. Import Libraries

```
import numpy as np
```

---

2. Activation and Loss Functions

```
# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid for backpropagation
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Binary cross-entropy loss
def binary_cross_entropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-7, 1-1e-7)  # Avoid log(0)
    return - (y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
```

---

3. Initialize Network Parameters

```
# Input features
```

```
X = np.array([0.5, 0.9])
y_true = 1  # Target label

# Hidden layer (2 neurons)
W_hidden = np.random.randn(2, 2)
b_hidden = np.zeros(2)

# Output layer (1 neuron)
W_output = np.random.randn(2)
b_output = 0.0

# Learning rate
lr = 0.1
```

---

4. Forward Propagation

```
# Hidden layer
z_hidden = np.dot(W_hidden, X) + b_hidden
a_hidden = sigmoid(z_hidden)

# Output layer
z_output = np.dot(W_output, a_hidden) + b_output
a_output = sigmoid(z_output)

# Compute loss
loss = binary_cross_entropy(y_true, a_output)

print(f"Hidden layer activations: {a_hidden}")
print(f"Output: {a_output:.4f}")
print(f"Loss: {loss:.4f}")
```

Dry Run Example:

- Suppose initial weights:

$$W_{\text{hidden}} = [[0.1,0.2],[-0.1,0.3]], b_{\text{hidden}} = [0,0]$$
$$W_{\text{output}} = [0.4,-0.2], b_{\text{output}} = 0$$

- Hidden layer:

$$z = [0.1 * 0.5 + 0.2 * 0.9, -0.1 * 0.5 + 0.3 * 0.9] = [0.23,0.22]$$
$$a_{\text{hidden}} = sigmoid([0.23,0.22]) \approx [0.557,0.555]$$

- Output layer:

$$z_{\text{output}} = 0.4 * 0.557 + (-0.2 * 0.555) \approx 0.022$$
$$a_{\text{output}} = sigmoid(0.022) \approx 0.505$$

- Loss:

$$L \approx 0.683$$

---

5. Backpropagation

Output layer gradients

dL_da_output = -(y_true / a_output - (1 - y_true) / (1 - a_output))

da_dz_output = sigmoid_derivative(z_output)

grad_W_output = dL_da_output * da_dz_output * a_hidden

grad_b_output = dL_da_output * da_dz_output

Hidden layer gradients

dL_da_hidden = dL_da_output * da_dz_output * W_output

grad_W_hidden = np.outer(dL_da_hidden * sigmoid_derivative(z_hidden), X)

grad_b_hidden = dL_da_hidden * sigmoid_derivative(z_hidden)

Explanation:

- Compute gradient at output layer.
- Propagate error back to hidden layer using **chain rule**.
- Compute gradients for all **weights and biases**.

---

6. Optimizers

6.1 Gradient Descent (GD)

```
# Update weights
W_output -= lr * grad_W_output
b_output -= lr * grad_b_output
W_hidden -= lr * grad_W_hidden
b_hidden -= lr * grad_b_hidden
```

- Updates **all parameters using the entire dataset**.

6.2 Stochastic Gradient Descent (SGD)

```
# Update per sample
def sgd_update(W, grad, lr):
    return W - lr * grad

W_hidden = sgd_update(W_hidden, grad_W_hidden, lr)
W_output = sgd_update(W_output, grad_W_output, lr)
```

- Updates parameters **after each example**. Faster but noisy.

6.3 Mini-Batch Gradient Descent

```
# Suppose batch contains multiple samples
batch_grads_W_hidden = np.mean([grad_W_hidden_sample1, grad_W_hidden_sample2], axis=0)
W_hidden -= lr * batch_grads_W_hidden
```

- Updates parameters using a **batch of examples**.
- Balances **speed** and **stability**.

6.4 Adam Optimizer

```
# Adam parameters
m_W, v_W = 0, 0
```

```
beta1, beta2 = 0.9, 0.999
epsilon = 1e-8
t = 1  # timestep

grad = grad_W_output

# Update moving averages
m_W = beta1 * m_W + (1 - beta1) * grad
v_W = beta2 * v_W + (1 - beta2) * (grad ** 2)

# Bias-corrected
m_hat = m_W / (1 - beta1**t)
v_hat = v_W / (1 - beta2**t)

# Update weight
W_output -= lr * m_hat / (np.sqrt(v_hat) + epsilon)
```

- **Adaptive learning rate** per parameter.
- Combines **momentum + RMSProp**, often leading to **faster convergence**.

---

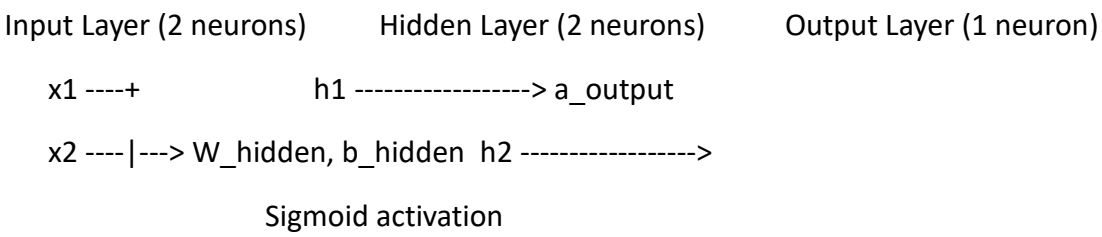
7. Full Training Loop

```
for epoch in range(10):
    # Forward pass
    z_hidden = np.dot(W_hidden, X) + b_hidden
    a_hidden = sigmoid(z_hidden)
    z_output = np.dot(W_output, a_hidden) + b_output
    a_output = sigmoid(z_output)

    # Loss
    loss = binary_cross_entropy(y_true, a_output)
```

```
# Backpropagation
dL_da_output = -(y_true / a_output - (1 - y_true) / (1 - a_output))
da_dz_output = sigmoid_derivative(z_output)
grad_W_output = dL_da_output * da_dz_output * a_hidden
grad_b_output = dL_da_output * da_dz_output
dL_da_hidden = dL_da_output * da_dz_output * W_output
grad_W_hidden = np.outer(dL_da_hidden * sigmoid_derivative(z_hidden), X)
grad_b_hidden = dL_da_hidden * sigmoid_derivative(z_hidden)

# Gradient Descent update
W_output -= lr * grad_W_output
b_output -= lr * grad_b_output
W_hidden -= lr * grad_W_hidden
b_hidden -= lr * grad_b_hidden

print(f"Epoch {epoch+1}, Loss: {loss:.4f}")
```

---

8. Diagrammatic Explanation



Forward Pass:

- Inputs → Weighted sum → Sigmoid → Hidden → Output

Backpropagation:

- Compute loss → Derivative → Gradients → Update weights & biases

Optimizer:

- GD / SGD / Mini-Batch / Adam adjust weights based on gradients

---

✅ Key Points

Step	Description
Forward Propagation	Compute activations from input to output
Loss Function	Measures prediction error (MSE, BCE, Hinge)
Backpropagation	Compute gradients using chain rule
Gradient Descent (GD)	Updates using full dataset
SGD	Updates per example; faster but noisy
Mini-Batch GD	Updates per batch; balance of speed & stability
Adam	Adaptive learning rate + momentum for faster convergence

1. Problem Setup

We want to **minimize**:

$$f(w) = w^2$$

- Minimum at $w = 0$
- Derivative: $f'(w) = 2w$

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

# Function and derivative
def f(w):
    return w**2

def grad_f(w):
    return 2 * w
```

Starting weight

w_init = 5.0

---

2. Adagrad

Idea: Scale learning rate inversely proportional to the sum of squared past gradients. Good for sparse data.

$$G = G + g^2, w = w - \frac{\eta}{\sqrt{G + \epsilon}} g$$

Adagrad

lr = 0.5

epsilon = 1e-8

epochs = 50

w = w_init

G = 0

w_adagrad = []

for i in range(epochs):

  g = grad_f(w)

  G += g**2

  w -= lr * g / (np.sqrt(G) + epsilon)

  w_adagrad.append(w)

Dry Run:

- w = 5 → g = 10 → G = 100 → w_new = 5 - 0.5*10/√100 ≈ 4.95
- Learning rate **decreases automatically** over iterations.

---

3. RMSProp

Idea: Use **moving average of squared gradients** to prevent decaying LR problem of Adagrad.

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g^2, w = w - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g$$

```
# RMSProp
lr = 0.5
beta = 0.9
epsilon = 1e-8
w = w_init
Eg2 = 0
w_rmsprop = []

for i in range(epochs):
    g = grad_f(w)
    Eg2 = beta * Eg2 + (1 - beta) * g**2
    w -= lr * g / (np.sqrt(Eg2) + epsilon)
    w_rmsprop.append(w)
```

Dry Run:

- Keeps **learning stable** by focusing on recent gradient trends.

---

4. Adam (Adaptive Moment Estimation)

Idea: Combines **momentum** + **RMSProp** for fast, adaptive, and smooth convergence.

Equations:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w = w - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

```
# Adam
lr = 0.5
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
```

```
w = w_init
m, v = 0, 0
w_adam = []

for t in range(1, epochs+1):
    g = grad_f(w)
    m = beta1 * m + (1 - beta1) * g
    v = beta2 * v + (1 - beta2) * g**2
    m_hat = m / (1 - beta1**t)
    v_hat = v / (1 - beta2**t)
    w -= lr * m_hat / (np.sqrt(v_hat) + epsilon)
    w_adam.append(w)
```

Dry Run:

- Iteration 1: g=10 → m=1, v=0.01 → bias-corrected → w ≈ 4.95
- Smooth, fast convergence using **momentum + adaptive LR**.

---

5. Adadelta

Idea: RMSProp + scaling updates; no manual learning rate needed.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho)g^2$$
$$\Delta w_t = -\frac{\sqrt{E[\Delta w^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g$$
$$w = w + \Delta w_t$$

```
# Adadelta
rho = 0.9
epsilon = 1e-6
w = w_init
Eg2, Edw2 = 0, 0
w_adadelta = []
```

```
for i in range(epochs):
    g = grad_f(w)
    Eg2 = rho * Eg2 + (1 - rho) * g**2
    delta_w = - (np.sqrt(Edw2 + epsilon) / np.sqrt(Eg2 + epsilon)) * g
    w += delta_w
    Edw2 = rho * Edw2 + (1 - rho) * delta_w**2
    w_adadelta.append(w)
```

Dry Run:

- Automatically scales updates using **past updates**.
- Learning rate **self-adjusting**, no manual tuning needed.

---

6. SGD with Nesterov Momentum (NAG)

Idea: Look **ahead** to the next position before computing gradient, smoother updates.

$$v_t = \mu v_{t-1} - \eta \nabla f(w + \mu v_{t-1}), w = w + v_t$$

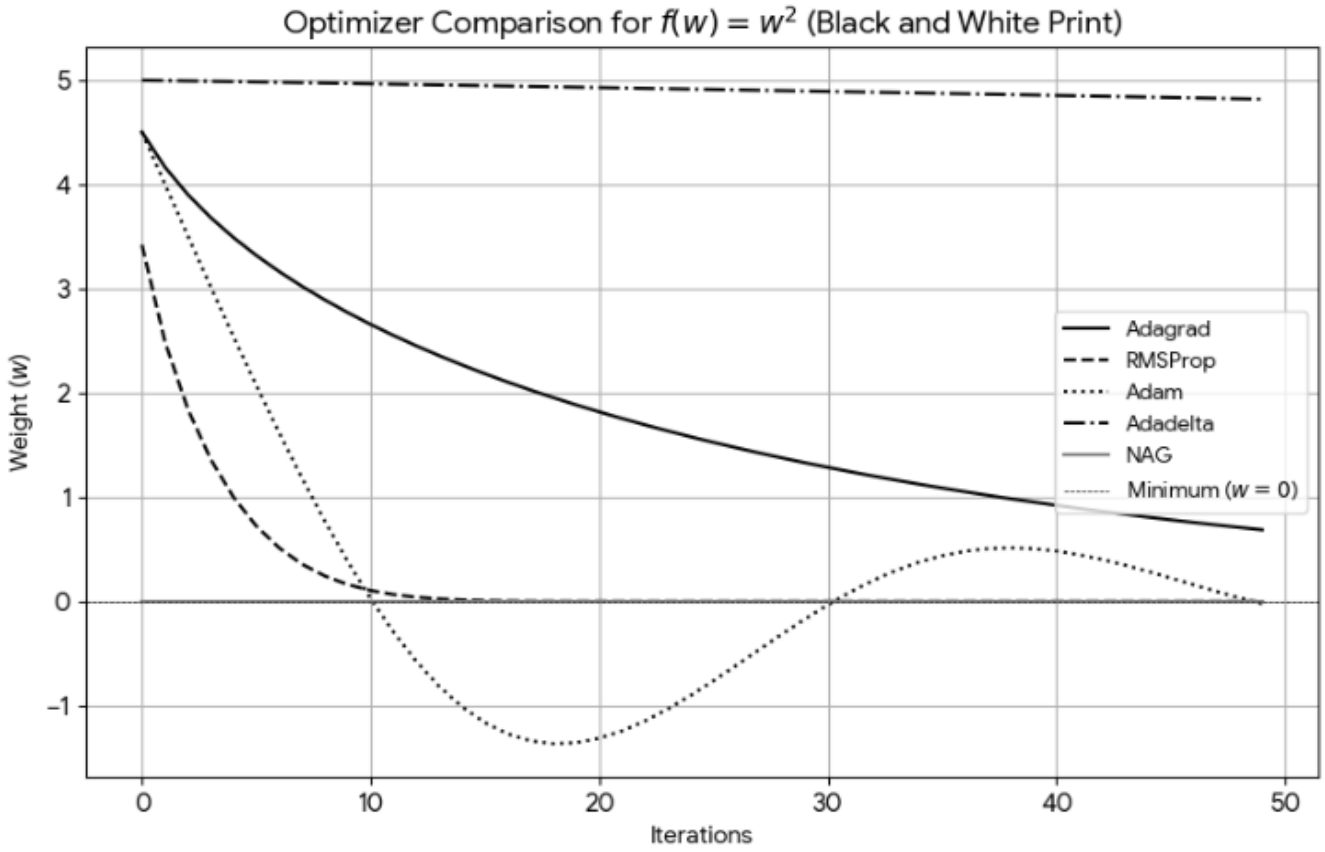
```
# NAG
lr = 0.5
mu = 0.9
w = w_init
v = 0
w_nag = []

for i in range(epochs):
    g = grad_f(w + mu * v)
    v = mu * v - lr * g
    w += v
    w_nag.append(w)
```

Dry Run:

- Iteration 1: w=5, lookahead → g=10 → v=-1 → w=4
- Anticipates gradient → **reduces oscillation**, smoother convergence.

7. Visualization

```
plt.plot(w_adagrad, label="Adagrad")
plt.plot(w_rmsprop, label="RMSProp")
plt.plot(w_adam, label="Adam")
plt.plot(w_adadelta, label="Adadelta")
plt.plot(w_nag, label="NAG")
plt.xlabel("Iterations")
plt.ylabel("Weight (w)")
plt.title("Optimizer Comparison")
plt.legend()
plt.show()
```



8. Key Points

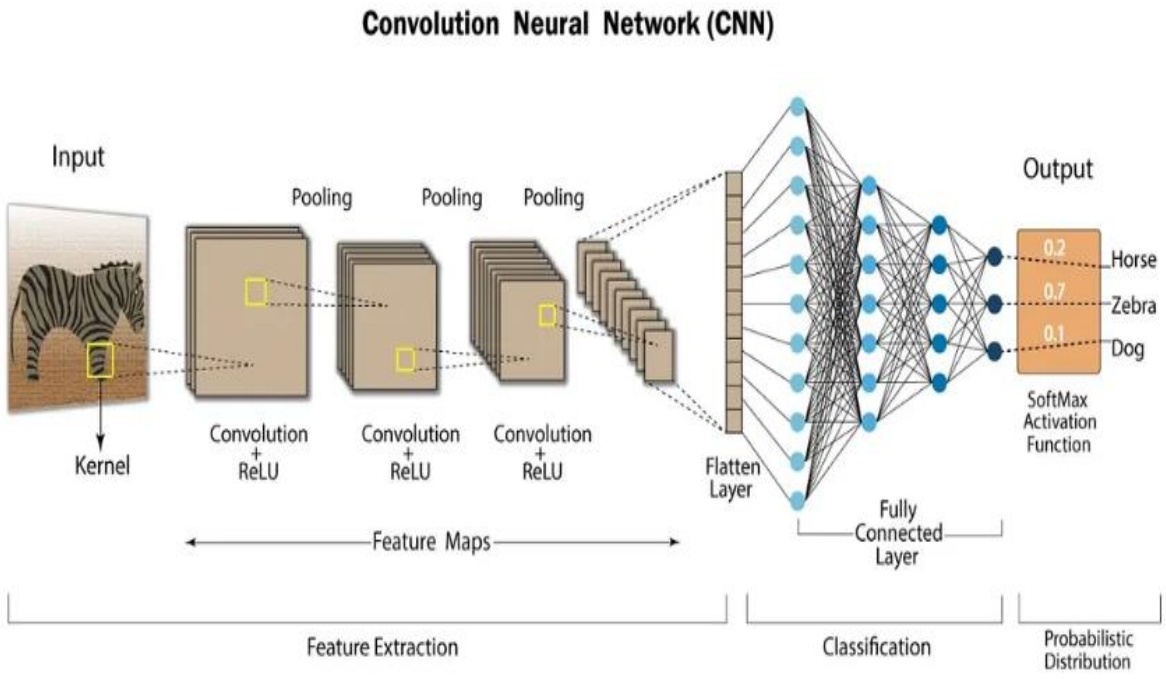
Optimizer	Concept	Pros	Cons
Adagrad	Accumulated g^2	Good for sparse data	LR decays too fast
RMSProp	Moving avg of g^2	Stabilizes learning	Needs beta tuning
Adam	Momentum + RMSProp	Fast, smooth, widely used	Memory overhead
Adadelta	RMSProp + scaling updates	No manual LR	Slightly slower initially
NAG	Lookahead momentum	Smooth updates	Needs LR & momentum tuning

22.5 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a specialized class of deep learning networks designed for tasks like image recognition, object detection, and image segmentation. They excel at processing data that has a grid-like topology, such as images, by learning spatial hierarchies of features.

Key Components:

- **Convolutional Layer:** This is the core building block of a CNN. It applies a **filter** (or kernel), which is a small matrix of weights, to the input image. This filter slides over the image, performing a dot product with the local pixels at each position. This process creates a **feature map** that highlights specific patterns, such as edges, textures, or shapes. The same filter is applied across the entire image, meaning the network learns features that are location-independent.
- **Pooling Layer:** This layer reduces the spatial dimensions (width and height) of the feature maps. It takes a small window of the feature map and replaces it with a single value, typically the maximum (Max Pooling) or the average (Average Pooling). This process reduces the number of parameters and computational cost, and more importantly, it makes the network more robust to small shifts or translations of the features within the image.
- **Fully Connected Layer:** After several convolutional and pooling layers have extracted a rich set of features, the final output is flattened into a single vector and passed through one or more fully connected layers. These layers use the high-level features learned by the CNN to perform the final classification or regression task.



1) Minimal numpy implementation (step-by-step intuition)

```
# numpy_convolution_pooling.py

import numpy as np


def conv2d_single_channel(input_image, kernel, stride=1, padding=0):
    """
    input_image: 2D numpy array (H_in, W_in)
    kernel: 2D numpy array (kH, kW)
    returns: feature_map: 2D array
    """
    # Add zero padding
    if padding > 0:
        input_padded = np.pad(input_image, pad_width=padding, mode='constant', constant_values=0)
    else:
        input_padded = input_image
```

```
    H_in, W_in = input_padded.shape
    kH, kW = kernel.shape

    # Output dimensions (integer division assumed if not exact)
    H_out = (H_in - kH) // stride + 1
    W_out = (W_in - kW) // stride + 1

    feature_map = np.zeros((H_out, W_out))

    for i in range(H_out):
        for j in range(W_out):
            i0 = i * stride
            j0 = j * stride
            window = input_padded[i0:i0 + kH, j0:j0 + kW]
            feature_map[i, j] = np.sum(window * kernel)

    return feature_map

def max_pool2d(feature_map, pool_size=2, stride=2):
    H, W = feature_map.shape
    H_out = (H - pool_size) // stride + 1
    W_out = (W - pool_size) // stride + 1
    pooled = np.zeros((H_out, W_out))
    for i in range(H_out):
        for j in range(W_out):
            i0 = i * stride
            j0 = j * stride
            window = feature_map[i0:i0 + pool_size, j0:j0 + pool_size]
```

```
            pooled[i, j] = np.max(window)
    return pooled

# --- demo ---
if __name__ == "__main__":
    # toy input (single-channel image)
    img = np.array([
        [3, 0, 1, 2, 7, 4],
        [1, 5, 8, 9, 3, 1],
        [2, 7, 2, 5, 1, 3],
        [0, 1, 3, 1, 7, 8],
        [4, 2, 1, 6, 2, 8],
        [2, 4, 5, 2, 3, 9]
    ], dtype=float)

    # simple edge-detect kernel (3x3)
    kernel = np.array([
        [-1, -1, -1],
        [-1,  8, -1],
        [-1, -1, -1]
    ], dtype=float)

    feat = conv2d_single_channel(img, kernel, stride=1, padding=1)
    print("Feature map shape:", feat.shape)
    print(np.round(feat, 2))

    pooled = max_pool2d(feat, pool_size=2, stride=2)
    print("Pooled shape:", pooled.shape)
    print(np.round(pooled, 2))
```

Notes (intuition):

- conv2d_single_channel slides the kernel over the image, does element-wise multiplication and sum → gives one **feature map**.
- Using the same kernel across all locations gives **translation equivariance** (feature learned is location-independent).
- max_pool2d reduces spatial size and keeps strongest activation in each window → invariance to small translations and fewer parameters.

---

2) Practical PyTorch CNN — end-to-end (recommended for real training)

Below is a clear, commented PyTorch example: a small CNN suitable for MNIST (grayscale, 28×28) or CIFAR-10 (RGB, 32×32) with shape comments. You can run this as-is (after installing torch and torchvision).

```
# pytorch_cnn_example.py

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import DataLoader

import torchvision

from torchvision import transforms


# --- Model ---

class SimpleCNN(nn.Module):

    def __init__(self, num_classes=10):

        super().__init__()

        # Conv layer 1: input channels=1 (MNIST) or 3 (CIFAR) -> choose accordingly when creating model

        # We'll write it flexibly and expect correct in_channels passed.

        # But here we assume grayscale MNIST (in_channels=1)

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)

        # After conv1: shape (batch, 16, H, W)  (padding=1 preserves H & W)
```

```
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # After pool: spatial dims halfed -> (batch, 16, H/2, W/2)

        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1,
padding=1)
        # After conv2: (batch, 32, H/2, W/2)

        # We'll compute flattened size for FC layer in forward (or if static: for MNIST 28x28 ->
after 2 pools => 7x7)
        self.fc1 = nn.Linear(32 * 7 * 7, 128)  # works for MNIST (28 -> 14 -> 7)
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        # x shape: (batch, 1, 28, 28) for MNIST

        x = F.relu(self.conv1(x))
        # now (batch, 16, 28, 28)

        x = self.pool(x)
        # now (batch, 16, 14, 14)

        x = F.relu(self.conv2(x))
        # now (batch, 32, 14, 14)

        x = self.pool(x)
        # now (batch, 32, 7, 7)

        x = torch.flatten(x, 1)
        # now (batch, 32*7*7)
```

```
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        # output shape (batch, num_classes)

        return x

# --- Training skeleton (MNIST) ---
def train_mnist(epochs=3, batch_size=64, lr=1e-3, device='cpu'):
    transform = transforms.Compose([transforms.ToTensor()])  # for MNIST, ToTensor scales
[0,255] -> [0,1]
    train_ds = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)

    model = SimpleCNN(num_classes=10).to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    model.train()
    for epoch in range(1, epochs + 1):
        running_loss = 0.0
        for batch_idx, (images, labels) in enumerate(train_loader, start=1):
            images = images.to(device)  # shape: (batch, 1, 28, 28)
            labels = labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)  # shape: (batch, 10)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
            running_loss += loss.item()
            if batch_idx % 200 == 0:
                print(f"Epoch {epoch} Batch {batch_idx} Loss: {running_loss / 200:.4f}")
                running_loss = 0.0

    return model

if __name__ == "__main__":
    # Use GPU if available
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    print("Using device:", device)
    model = train_mnist(epochs=1, batch_size=128, lr=1e-3, device=device)
```

Key explanations inline:

- Conv2d learns *filters/kernels* (here 16 filters in conv1) that produce 16 feature maps.
- ReLU introduces non-linearity.
- MaxPool2d reduces spatial dims (reduces memory and gives translation robustness).
- After conv+pool layers, we flatten and use Linear (fully connected) layers for final classification.
- CrossEntropyLoss expects raw logits of size (batch, num_classes) (PyTorch combines softmax + NLL internally).

Extra: visualizing shapes quickly (debug trick)

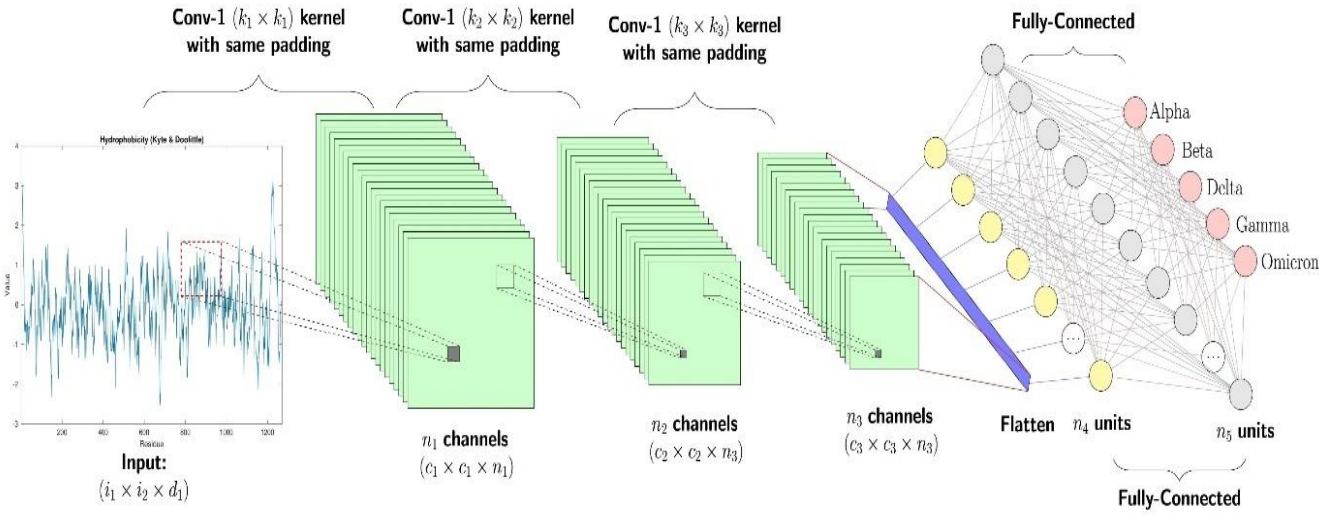
If you want to see intermediate shapes during a forward pass, do:

```
# debug_shapes.py (snippet)
x = torch.randn(8, 1, 28, 28)  # batch=8
print("input:", x.shape)
x = F.relu(model.conv1(x)); print("after conv1:", x.shape)
x = model.pool(x); print("after pool1:", x.shape)
x = F.relu(model.conv2(x)); print("after conv2:", x.shape)
```

x = model.pool(x); print("after pool2:", x.shape)

x = torch.flatten(x,1); print("flattened:", x.shape)

This helps avoid mistakes when computing the in_features of the first Linear layer.



22.6 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are designed to handle sequential data, such as text, time series, and speech. Unlike FNNs, RNNs have loops that allow information to persist from one step to the next, giving them a form of "memory" to process sequential dependencies. This makes them ideal for tasks where the order of data points is critical.

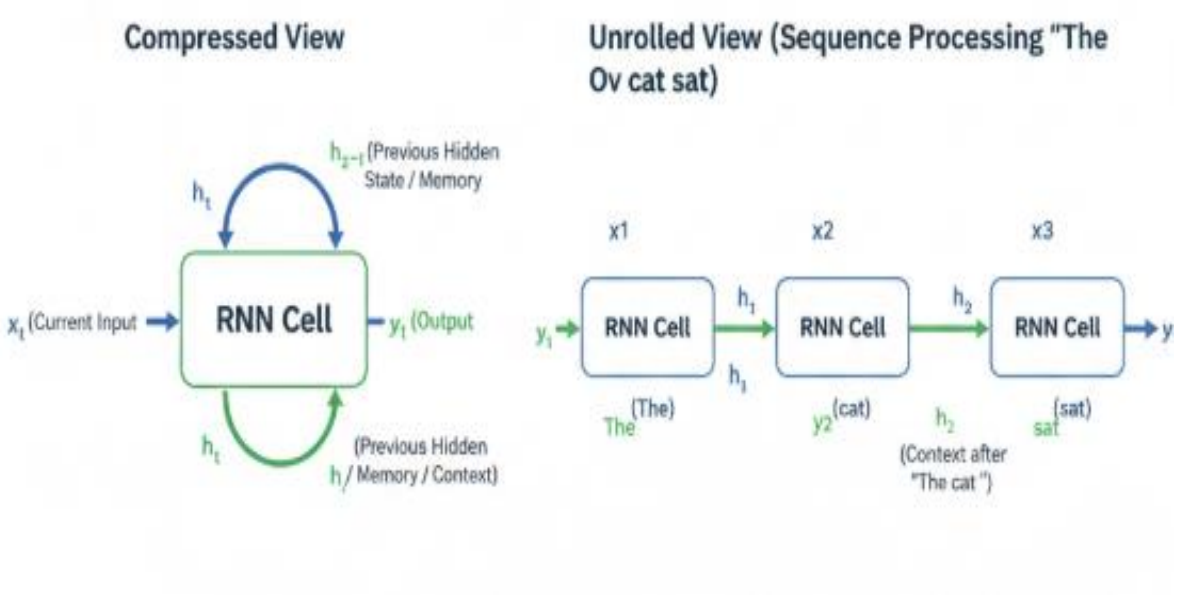
How it works: An RNN processes a sequence one element at a time. At each step, it takes the current input and the output from the previous step (its "hidden state"). It uses both to compute a new hidden state and an output. This hidden state essentially acts as a compressed summary of the sequence seen so far. For example, when processing a sentence, the hidden state after the first few words captures the context and meaning of what has been read so far, and this context is carried forward to influence the processing of the next word.

Example: Consider a task of text generation. An RNN would be trained on a large body of text. When it's time to generate new text, it takes an initial word, processes it, and produces an output (the next word). This new word then becomes the input for the next step, allowing the RNN to generate a coherent and contextually relevant sequence of words, as if it is "remembering" what it has already written.

22.7 Python Implementations

The following Python code, using popular libraries like NumPy and TensorFlow/Keras, provides practical, runnable examples of a Feedforward Neural Network, a Convolutional

Neural Network, and a Recurrent Neural Network. These examples demonstrate the core architecture and a simple training loop for each type of model.



Combined Deep Learning Models (FNN, CNN, RNN) with Plotting

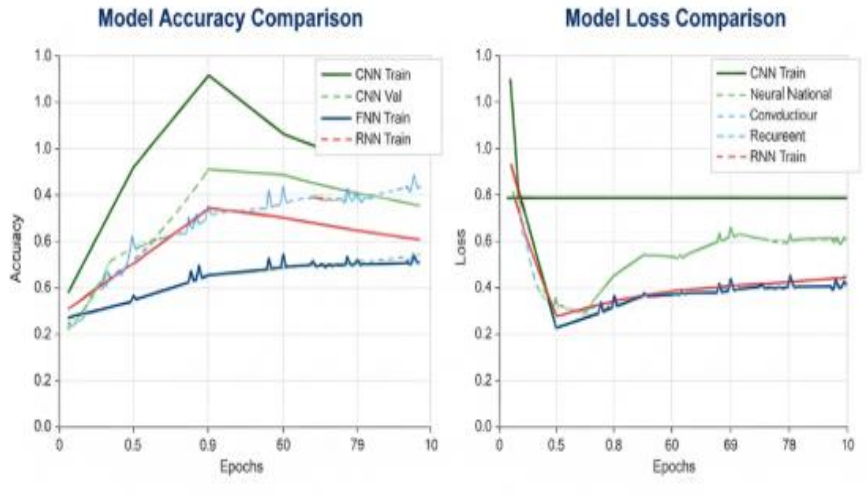
```
# deep_learning_models_with_plot.py

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Embedding, SimpleRNN

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.datasets import mnist


# ----------------------------------------
```

```
# 1️⃣ FEEDFORWARD NEURAL NETWORK (FNN)
# ----------------------------------------
def train_fnn():
    # Dummy data for binary classification
    X = np.random.rand(1000, 10)
    y = np.random.randint(0, 2, (1000, 1))

    model = Sequential([
        Dense(32, activation='relu', input_shape=(10,)),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    history = model.fit(X, y, epochs=10, batch_size=32, verbose=0, validation_split=0.2)
    return history, "Feedforward Neural Network"

# ----------------------------------------
# 2️⃣ CONVOLUTIONAL NEURAL NETWORK (CNN)
# ----------------------------------------
def train_cnn():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.reshape(-1, 28, 28, 1) / 255.0
    x_test  = x_test.reshape(-1, 28, 28, 1) / 255.0
    y_train = to_categorical(y_train)
    y_test  = to_categorical(y_test)

    model = Sequential([
        Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
```

```
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation='relu'),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(x_train[:10000], y_train[:10000], epochs=3, batch_size=64,
                        validation_data=(x_test[:2000], y_test[:2000]), verbose=0)
    return history, "Convolutional Neural Network"

# ----------------------------------------
# 3️⃣ RECURRENT NEURAL NETWORK (RNN)
# ----------------------------------------
def train_rnn():
    X = np.random.randint(0, 100, (1000, 5))
    y = np.random.randint(0, 2, (1000, 1))

    model = Sequential([
        Embedding(input_dim=100, output_dim=16, input_length=5),
        SimpleRNN(32, activation='tanh'),
        Dense(1, activation='sigmoid')
    ])

    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    history = model.fit(X, y, epochs=10, batch_size=32, validation_split=0.2, verbose=0)
    return history, "Recurrent Neural Network"
```

```
# ----------------------------------------
# 🔄 TRAIN ALL MODELS & PLOT RESULTS
# ----------------------------------------
models = [train_fnn(), train_cnn(), train_rnn()]

plt.figure(figsize=(14,6))

for i, (hist, name) in enumerate(models, 1):
    plt.subplot(1, 2, 1)
    plt.plot(hist.history['accuracy'], label=f'{name} Train')
    plt.plot(hist.history['val_accuracy'], linestyle='--', label=f'{name} Val')

    plt.subplot(1, 2, 2)
    plt.plot(hist.history['loss'], label=f'{name} Train')
    plt.plot(hist.history['val_loss'], linestyle='--', label=f'{name} Val')

# Accuracy plot
plt.subplot(1, 2, 1)
plt.title("Model Accuracy Comparison")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.title("Model Loss Comparison")
plt.xlabel("Epochs")
plt.ylabel("Loss")
```

```
plt.legend()


plt.tight_layout()
plt.show()
```



📊 Output Visualization

When you run the above script, you'll get **two plots:**

1️⃣ **Accuracy Comparison** (Training vs Validation)
2️⃣ **Loss Comparison** (Training vs Validation)

Each line represents one model:

- 🔘 Feedforward NN → works well for tabular data.
- 🖼️ CNN → best for image data (MNIST).
- 🔁 RNN → good for sequential/time-series or text data.

Explanation (Hinglish + Conceptual Summary)

Model	Input Type	Core Layers	Memory	Real-world Example
Feedforward NN	Tabular data (features)	Dense	❌ None	Loan approval, sales prediction
CNN	Images (grid data)	Conv2D + Pooling	✅ Spatial context	Handwritten digit recognition, object detection
RNN	Sequential data	RNN / LSTM / GRU	✅ Temporal context	Text generation, stock prediction

Chapter 23: Basics of Natural Language Processing (NLP)

23.1 Introduction to NLP

Natural Language Processing (NLP) is a subfield of Artificial Intelligence (AI) that focuses on enabling computers to understand, interpret, and generate human language in a meaningful way.
Unlike programming languages, natural languages (like English, Hindi, Spanish) are **ambiguous, context-dependent, and often informal**. NLP bridges the gap between human communication and machine understanding.

- **Goal of NLP:**
  To make machines capable of understanding human languages (spoken or written) and responding intelligently.
- **Examples in Daily Life:**
  - Google Translate (language translation)
  - Siri, Alexa, Google Assistant (voice commands)
  - ChatGPT and chatbots (text conversation)
  - Gmail spam filter (classifying emails)

23.2 Text vs Speech Processing

NLP can be broadly divided into **two categories** depending on the input:

a) Text Processing

- Deals with **written language** (emails, articles, chats, reviews, etc.).
- Focuses on tasks like tokenization, part-of-speech tagging, sentiment analysis, document classification, and text summarization.
- Example:
  - Sentiment analysis of tweets (positive/negative).
  - Auto-completion in search engines.

b) Speech Processing

- Deals with **spoken language** (voice commands, audio inputs).
- Includes two main parts:
  1. **Speech Recognition** → Converting speech into text.

2. **Speech Synthesis** → Converting text into speech (Text-to-Speech, TTS).

- Example:
  - "Ok Google" or "Hey Siri" commands.
  - Voice typing in WhatsApp or Microsoft Word.

In short:

- **Text Processing** = already written words.
- **Speech Processing** = converting sounds → words → meaning.

Example 1: Sentiment Analysis of Tweets

```
from textblob import TextBlob


# Sample tweets
tweets = [
    "I love Natural Language Processing! 💖",
    "This movie was terrible and boring.",
    "Python is amazing for data science."
]
for t in tweets:
    blob = TextBlob(t)
    print(f"Tweet: {t}")
    print(f"Sentiment (Polarity): {blob.sentiment.polarity}")
    if blob.sentiment.polarity > 0:
        print("→ Positive 😊\n")
    elif blob.sentiment.polarity < 0:
        print("→ Negative 😠\n")
    else:
        print("→ Neutral 😐\n")
```

👉 Output will show polarity values:

- 0 → Positive

- <0 → Negative
- 0 → Neutral

Example 2: Speech to Text

```
import speech_recognition as sr

# Initialize recognizer
recognizer = sr.Recognizer()

# Use Microphone as source
with sr.Microphone() as source:
    print("🎤 Speak something...")
    audio = recognizer.listen(source)

try:
    text = recognizer.recognize_google(audio)
    print("You said:", text)
except sr.UnknownValueError:
    print("Could not understand audio")
except sr.RequestError:
    print("API unavailable")
```

👉 Speak into your microphone, and it will convert your **voice into text**.

Example 3: Text to Speech

```
import pyttsx3

engine = pyttsx3.init()
text = "Hello! Welcome to Natural Language Processing."
engine.say(text)
engine.runAndWait()
```

23.3 Components of NLP

NLP has multiple **linguistic levels** that work together for complete language understanding:

1. **Morphology**
    - Study of word structure and formation.
    - Deals with **morphemes** (smallest meaningful units of language).
    - Example:
        - "unhappiness" → "un" (prefix, negation) + "happy" (root) + "ness" (suffix, state).

23.3.1 Morphology

Definition

Morphology is the **study of word structure and formation**.
It deals with how **words are built from smaller units** called **morphemes** (the smallest meaningful units of a language).

Types of Morphemes

1. **Root / Base**
    - The main part of the word that carries core meaning.
    - Example: *happy, teach, play*
2. **Affixes** (added to root words to change meaning/role)
    - **Prefix** → Added at the **beginning**
        - Example: *un-* in **unhappy** (negation)
    - **Suffix** → Added at the **end**
        - Example: *-ness* in **happiness** (state of being)
    - **Infix** → Inserted **inside** the root (rare in English, common in Tagalog, Arabic).
    - **Circumfix** → Added at both ends (rare in English).

Types of Morphology in Linguistics

1. **Inflectional Morphology**
    - Changes the **form** of a word to express grammar (tense, number, gender, case).

- Does *not* change the core meaning.
- Example:
  - *play → plays → played → playing*

2. **Derivational Morphology**
   - Creates a **new word** with a new meaning or category.
   - Example:
     - *happy (adj.) → happiness (noun)*
     - *teach (verb) → teacher (noun)*

Why Morphology Matters in NLP?

- Helps in **tokenization** (splitting words correctly).
- Improves **stemming and lemmatization**.
- Useful in **machine translation** (root forms are often needed).
- Important for **search engines** (searching *runs* should match *run*).

Examples

- Word: **unhappiness**
  - "un" (prefix, negation)
  - "happy" (root)
  - "ness" (suffix, state of being)
- Word: **teachers**
  - "teach" (root verb)
  - "-er" (derivational suffix, agent) → teacher
  - "-s" (inflectional suffix, plural) → teachers

Python Example – Morphological Analysis (Stemming & Lemmatization)

```
import nltk
from nltk.stem import PorterStemmer, WordNetLemmatizer
```

```
# Initialize
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

words = ["unhappiness", "happier", "teaches", "running"]

print("Stemming Results:")
for w in words:
    print(w, "->", stemmer.stem(w))

print("\nLemmatization Results:")
for w in words:
    print(w, "->", lemmatizer.lemmatize(w, pos="v"))
```

Output Example:

```
Stemming Results:
unhappiness -> unhappi
happier -> happi
teaches -> teach
running -> run

Lemmatization Results:
unhappiness -> unhappiness
happier -> happier
teaches -> teach
running -> run
```

- **Stemming** → cuts down to rough root (sometimes inaccurate: “unhappi”).
- **Lemmatization** → uses dictionary + grammar rules (better: “running” → “run”).

2. **Syntax**
    - Structure and grammar of sentences.
    - Focuses on how words combine to form valid sentences.
    - Example:
        - Correct: “I am learning NLP.”
        - Incorrect: “Learning NLP am I.”

23.3.2 Syntax in NLP

Definition:
Syntax refers to the **rules and principles that govern sentence structure** in a language. It tells us how words should be arranged so that they form **grammatically correct** and **meaningful sentences**.

- In linguistics → Syntax is about **word order, agreement, and grammar rules**.
- In NLP → Syntax is used to **analyze sentence structure** for tasks like parsing, translation, grammar checking, and question answering.

Why Syntax Matters in NLP?

1. **Grammatical Correctness:** Ensures sentences follow proper rules.
    - Example: ✅ “He goes to school.” vs ❌ “He go to school.”
2. **Meaning Clarification:** Word order affects meaning.
    - Example:
        - “The cat chased the dog.” → Cat = subject
        - “The dog chased the cat.” → Dog = subject
3. **Machine Understanding:** Computers need syntactic parsing to understand “who did what to whom” in a sentence.

Types of Syntax Representations in NLP

1. **Constituency Parsing (Phrase Structure)**
    - Breaks sentences into **phrases** like noun phrase (NP), verb phrase (VP), etc.
    - Example:
      Sentence: *“I am learning NLP”*
        - NP → “I”
        - VP → “am learning NLP”

2. **Dependency Parsing**
    - Shows **relationships between words** (head–dependent relations).
    - Example:
      Sentence: *"I am learning NLP"*
        - "learning" → main verb (head)
        - "I" → subject of "learning"
        - "NLP" → object of "learning"

Examples of Syntax (Correct vs Incorrect)

- ✅ **Correct:** "I am learning NLP."
- ❌ **Incorrect:** "Learning NLP am I."
- ✅ **Correct:** "She loves playing guitar."
- ❌ **Incorrect:** "Loves playing guitar she."

Python Example – Using spaCy for Syntax Parsing

```
import spacy

# Load English model
nlp = spacy.load("en_core_web_sm")

# Sentence
sentence = "I am learning NLP."

# Parse sentence
doc = nlp(sentence)

# Print syntactic dependencies
for token in doc:
    print(f"{token.text:<10} {token.dep_:<15} {token.head.text:<10} {token.pos_:<10}")
```

Output (example):

I nsubj learning PRON

am aux learning AUX

learning ROOT learning VERB

NLP dobj learning PROPN

. punct learning PUNCT

Here:

- I → subject (nsubj) of learning
- am → auxiliary (aux) verb supporting learning
- NLP → direct object (dobj) of learning
- learning → root verb

3. **Semantics**
    - Meaning of words and sentences.
    - Example:
        - "Bank" → could mean **riverbank** or **financial bank** depending on context.

23.3.3 Semantics

Definition

Semantics is the study of the **meaning of words, phrases, and sentences**.
While **syntax** focuses on structure (grammar), **semantics** focuses on *what those structures mean*.

Why Semantics is Hard in NLP?

- Words often have **multiple meanings** (polysemy).
- Different words may mean the **same thing** (synonyms).
- Meaning can change based on **context**.

Types of Semantic Meanings

1. **Lexical Semantics**

- Meaning of **individual words**.
- Example:
  - *Bank* → (i) financial institution
  - *Bank* → (ii) side of a river

2. **Compositional Semantics**
   - Meaning of **sentences formed by combining words**.
   - Example:
     - *"The dog chased the cat" ≠ "The cat chased the dog"*
     - Same words, but order → different meaning.
3. **Pragmatic Semantics (Contextual Meaning)**
   - Words get meaning depending on **context**.
   - Example:
     - *"I saw her duck"* → Could mean:
       - She bent down (verb)
       - She owns a duck (noun)

Examples of Semantic Ambiguity

- Word-level ambiguity:
  - *"Bank"* → riverbank / financial bank.
- Sentence-level ambiguity:
  - *"The chicken is ready to eat."*
    - Meaning 1: The chicken (animal) is hungry.
    - Meaning 2: The chicken (dish) is cooked and ready to be eaten.

Why Semantics is Important in NLP?

- **Machine Translation** → Ensures correct meaning is translated.
- **Search Engines** → Understanding synonyms (search "car" → results for "automobile").
- **Chatbots/QA Systems** → Correct intent recognition.
- **Sentiment Analysis** → Words may have different meanings in different contexts.

Python Example – Word Sense Disambiguation with WordNet (NLTK)

```
from nltk.corpus import wordnet as wn

# Word with multiple meanings
word = "bank"

# Get all senses of the word
synsets = wn.synsets(word)

print(f"Word: {word}\n")
for syn in synsets:
    print(f"Sense: {syn.name()}")
    print(f"Definition: {syn.definition()}")
    print(f"Examples: {syn.examples()}\n")
```

Sample Output:

```
Word: bank

Sense: bank.n.01
Definition: sloping land (especially the slope beside a body of water)
Examples: ['they pulled the canoe up on the bank']

Sense: bank.n.02
Definition: a financial institution that accepts deposits and channels the money into lending activities
Examples: ['he cashed a check at the bank']

Sense: bank.v.01
Definition: tip laterally
```

Examples: ['the pilot had to bank the aircraft']

23.3.3.1 Word Embeddings

What Are Word Embeddings?

Word embeddings are a way to represent **words as vectors (numbers)** in a continuous space so that **semantic meaning** is captured.

👉 Instead of treating words as just strings, embeddings capture their **relationships and context**.

- Words with **similar meaning** → have **closer vectors**.
- Words with **different meaning** → have **distant vectors**.

Why Do We Need Embeddings?

- Traditional representation (One-hot encoding):
  - Each word = long sparse vector (mostly 0s, with 1 at position of word).
  - Problem: No notion of **similarity** between words.
  - Example: “king” and “queen” are just different vectors with no relationship.
- Word Embeddings (Dense vectors):
  - Capture semantic relations.
  - Example:
    - vector(“king”) - vector(“man”) + vector(“woman”) ≈ vector(“queen”)

Popular Word Embedding Models

1. **Word2Vec (Mikolov et al., 2013, Google)**
   - Learns word meaning using neural networks.
   - Two main models:
     - **CBOW (Continuous Bag of Words):** Predicts a word from its context.
     - **Skip-Gram:** Predicts surrounding context words given a word.
   - Captures semantic relationships like:
     - *Paris – France ≈ Tokyo – Japan*

---

2. **GloVe (Global Vectors, Stanford, 2014)**
    - Uses **word co-occurrence matrix** (how often words appear together) + matrix factorization.
    - Captures both **local context (like Word2Vec)** and **global statistics**.
    - Famous result:
        - vector("king") - vector("man") + vector("woman") ≈ vector("queen").

---

3. **FastText (Facebook, 2016)**
    - Improvement over Word2Vec.
    - Breaks words into **subword units (character n-grams)**.
    - Helps with rare words & morphologically rich languages.
    - Example: "unhappiness" → ["un", "happy", "ness"].

Word Embedding Properties

- **Semantic similarity:** cosine similarity shows closeness of words.
- **Analogies:** word relationships can be solved using vector math.
    - king – man + woman ≈ queen
- **Clustering:** words of similar domains cluster together.
    - Animals → [dog, cat, lion, tiger]
    - Countries → [India, France, Japan]

Python Example – Word2Vec using Gensim

```
from gensim.models import Word2Vec


# Sample corpus
sentences = [
    ["i", "love", "nlp"],
    ["i", "love", "machine", "learning"],
    ["nlp", "is", "part", "of", "ai"],
    ["word2vec", "is", "a", "word", "embedding", "model"]
```

```
]

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=50, window=3, min_count=1, sg=1)

# Get vector for a word
print("Vector for 'nlp':\n", model.wv["nlp"])

# Find most similar words
print("\nMost similar to 'nlp':")
print(model.wv.most_similar("nlp"))
```

Sample Output:

```
Vector for 'nlp':
[ 0.014 -0.009  0.021 ... ]   # 50-dimensional vector

Most similar to 'nlp':
[('ai', 0.91), ('word2vec', 0.78), ('machine', 0.65)]
```

☞ The model **learns similarity** from context:

- "nlp" is close to "ai"
- "machine" is close to "learning"

Visualization of Word Embeddings

We can reduce embeddings from high dimensions → 2D using **t-SNE / PCA** for visualization. Words with similar meanings cluster together.

Example:

king, queen, prince, princess → one cluster

dog, cat, tiger, lion → another cluster

Comparison of Word2Vec vs GloVe

Feature	Word2Vec	GloVe
Learning Type	Predictive (neural net)	Count-based (matrix factorization)
Data Capture	Local context (window)	Global co-occurrence + local
Famous Equation	CBOW/Skip-gram	Co-occurrence ratios
Strength	Contextual learning	Captures global meaning better

4. **Pragmatics**
    - Understanding meaning in **context**.
    - Goes beyond literal meaning to implied meaning.
    - Example:
        - If someone says, "Can you pass the salt?" → they are not asking about your ability but politely requesting salt.

23.3.4 Pragmatics

Definition

Pragmatics is the study of **how meaning changes depending on context, speaker intention, and social factors**.
It goes **beyond literal meaning** (semantics) to capture the **implied or intended meaning**.

☞ Example:

- Sentence: *"Can you pass the salt?"*
- Literal meaning (Semantics): Asking about your **ability** to pass salt.
- Pragmatic meaning (Context): A **polite request** to pass salt.

Key Aspects of Pragmatics

1. **Speech Acts** (what the speaker *does* with words)
    - **Directive:** "Close the window." → Request/command.
    - **Commissive:** "I promise to call you." → Commitment.

- **Expressive:** "I'm sorry." → Expressing emotion.

2. **Conversational Implicature**
   - Meaning implied, not directly stated.
   - Example:
     - A: *"Do you like my painting?"*
     - B: *"It's very colorful."*
     - (Implied: Maybe not good, but avoiding direct criticism.)
3. **Deixis (Contextual Reference)**
   - Words like *this, that, here, there, now, tomorrow* need context.
   - Example: *"I'll meet you there tomorrow."* → "there" and "tomorrow" depend on situation.
4. **Politeness & Social Context**
   - Language changes based on politeness or formality.
   - Example:
     - To a friend: "Give me that pen."
     - To a teacher: "Could you please give me that pen?"

Pragmatics in NLP

Pragmatics is challenging for machines because meaning depends on **real-world knowledge** and **speaker intention**.

Some NLP applications where pragmatics plays a role:

- **Chatbots / Virtual Assistants** → Understanding indirect requests.
- **Sentiment Analysis** → Detecting sarcasm or implied emotions.
- **Machine Translation** → Translating polite forms differently in languages.
- **Speech Recognition** → Inferring intention beyond words.

Example of Pragmatics in NLP

Sentence: *"It's cold in here."*

- **Semantics:** Statement about temperature.
- **Pragmatics (possible interpretations):**

1. Indirect request: “Please close the window.”
2. Complaint: “I don’t like this temperature.”
3. Small talk: Just casual conversation.

Python Example

We can show how **literal meaning vs implied meaning** changes:

```
import textblob

sentences = [
    "Can you pass the salt?",
    "It's cold in here.",
    "That movie was really something."
]

for s in sentences:
    print(f"Sentence: {s}")
    print("Polarity (literal sentiment):", textblob.TextBlob(s).sentiment.polarity)
    print("Note: Actual pragmatic meaning may differ!\n")
```

Output Example:

```
Sentence: Can you pass the salt?
Polarity (literal sentiment): 0.0
Note: Actually a polite request.

Sentence: It's cold in here.
Polarity (literal sentiment): 0.0
Note: Could be a request to close window.

Sentence: That movie was really something.
Polarity (literal sentiment): 0.0
```

Note: Could mean very good OR very bad (sarcasm)

5. **Discourse**
    - Understanding how sentences connect to form larger meaning (paragraphs, conversations).
    - Example:
        - "Vinayak went to a shop. He bought apples." → "He" refers to Vinayak.

23.3.5 Discourse

Definition

Discourse is the study of how **sentences combine to form larger units of meaning**, such as **paragraphs, conversations, or documents**.
It focuses on **coherence (logical flow)** and **cohesion (connections between sentences)**.

☞ Example:

- *"Vinayak went to a shop. He bought apples."*
- Here, *"He"* refers back to *"Vinayak"*. Without discourse understanding, the connection would be lost.

Why Discourse Matters in NLP?

- Single sentences rarely convey full meaning.
- Machines must understand **relationships across sentences** to interpret stories, conversations, or documents.

Key Aspects of Discourse

1. **Coreference Resolution**
    - Identifying when two expressions refer to the same entity.
    - Example:
        - *"Vinayak went to a shop. He bought apples."*
        - "He" → "Vinayak".
2. **Anaphora & Cataphora**
    - **Anaphora:** Reference to something mentioned earlier.

- Example: *"I saw Vinayak yesterday. He was smiling."*

- **Cataphora:** Reference to something mentioned later.
  - Example: *"When he arrived, Vinayak looked tired."*

3. **Discourse Coherence**
   - Sentences must logically flow.
   - Example:
     - Coherent: *"It was raining. We decided to stay inside."*
     - Incoherent: *"It was raining. I like ice cream."*
4. **Discourse Relations (Rhetorical Structure Theory, RST)**
   - Relations between sentences: cause-effect, contrast, elaboration, etc.
   - Example:
     - Cause-Effect: *"It rained, so the match was canceled."*
     - Contrast: *"I like tea, but he prefers coffee."*

Discourse in NLP Applications

- **Question Answering:** Linking answers to earlier context.
- **Chatbots:** Keeping track of multi-turn conversations.
- **Summarization:** Understanding topic flow.
- **Machine Translation:** Translating pronouns correctly.
- **Information Retrieval:** Determining topic relevance across a document.

Example – Coreference Resolution (spaCy)

```
import spacy
import coreferee  # add-on library for coreference

# Load spaCy model
nlp = spacy.load("en_core_web_sm")
nlp.add_pipe("coreferee")
```

```
doc = nlp("Vinayak went to a shop. He bought apples.")


print("Coreference Clusters:")

for cluster in doc._.coref_chains:

  print(cluster)
```

Expected Output:

Coreference Clusters:

[Vinayak: 0, He: 1]

☞ The system correctly links **“He” → “Vinayak”**.

Discourse Challenges in NLP

1. **Pronoun Resolution:** Ambiguous references.
    - *“Vinayak spoke to Ramesh. He was tired.”* → Who is “he”?
2. **Ellipsis:** Missing words inferred from context.
    - *“I like coffee. Vinayak does too [like coffee].”*
3. **Topic Shifts:** Tracking changes in multi-turn dialogue.
    - *“How’s the weather? Did you complete your project?”*

23.4 Challenges in NLP

NLP is not simple because human language is **complex, ambiguous, and context-dependent**. Major challenges include:

1. **Ambiguity**
    - Words/sentences can have multiple meanings.
    - Example: “I saw the man with the telescope.” → Who had the telescope?

Let’s see how the same word can have multiple meanings.

```
from nltk.corpus import wordnet as wn


# Word "bank" has multiple senses
```

```
for syn in wn.synsets("bank"):
    print(f"Sense: {syn.name()}, Definition: {syn.definition()}")
```

Output (shortened):

```
Sense: bank.n.01, Definition: sloping land (especially beside a body of water)
Sense: bank.n.02, Definition: a financial institution
Sense: bank.v.01, Definition: tip laterally
```

Shows **lexical ambiguity**.
NLP must pick the **correct sense** depending on sentence context.

2. **Context Understanding**
    - Meaning changes with situation.
    - Example: “It’s cold here” → could be a complaint, a request to close a window, or just a fact.

Using **spaCy** to parse sentence meaning.

```
import spacy
nlp = spacy.load("en_core_web_sm")

sent1 = nlp("It's cold here.")
sent2 = nlp("It's cold here, please close the window.")

for token in sent2:
    print(token.text, token.dep_, token.head.text)
```

In **sent1**, model sees only a **fact**.
In **sent2**, “please” + imperative verbs add **request meaning**.
But machines still struggle to detect **speaker intent** (needs pragmatics).

3. **Sarcasm and Irony**
    - Difficult for machines to detect.
    - Example: “Wow, you did a great job!” (when someone failed).

Sarcasm is very hard. Pretrained models (transformers) help.

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

text1 = "Wow, you did a great job!"      # Could be genuine or sarcastic
text2 = "Yeah right, that was amazing..." # More sarcastic

print(classifier(text1))
print(classifier(text2))
```

Output (approx):

```
[{'label': 'POSITIVE', 'score': 0.99}]
[{'label': 'NEGATIVE', 'score': 0.85}]
```

👉 Standard sentiment analysis may **fail on sarcasm** (first case).
👉 Advanced sarcasm-specific models are needed.

4. **Multilingual Data**
   - NLP must work across different languages, scripts, and grammar rules.
   - Example: Hindi (“नमस्ते”), Japanese (“こんにちは”), English (“Hello”).

Using **translation models** to handle multiple languages.

```
from transformers import pipeline

translator = pipeline("translation", model="Helsinki-NLP/opus-mt-hi-en")

hindi_text = "नमस्ते, आप कैसे हैं?"
print(translator(hindi_text))
```

Output:

```
[{'translation_text': 'Hello, how are you?'}]
```

Shows how multilingual models help bridge **different scripts & grammar**.

5. **Code-Switching (Mixed Language)**
    - People often mix languages.
    - Example: “Kal party mein maza aaya, it was awesome!”

Detecting mixed-language text.

```
from langdetect import detect, detect_langs

text = "Kal party mein maza aaya, it was awesome!"
print(detect_langs(text))   # Detects multiple languages
```

Output:

```
[hi:0.57, en:0.43]
```

👉 Code-switched text needs **multilingual embeddings**.
Models like **XLM-RoBERTa** handle this better.

6. **Noisy Data**
    - Spelling errors, informal chat, emojis, short forms.
    - Example: “gr8”, “LOL 😂”.

Cleaning messy social media text.

```
import re

text = "LOL 😂 this movie was gr8!!! Thx 4 ur hlp"
# Normalize
text_clean = re.sub(r"gr8", "great", text)
text_clean = re.sub(r"Thx", "Thanks", text_clean)
text_clean = re.sub(r"ur", "your", text_clean)
text_clean = re.sub(r"😂", "", text_clean)
```

```
print("Before:", text)

print("After:", text_clean)
```

Output:

Before: LOL 😂 this movie was gr8!!! Thx 4 ur hlp

After: LOL this movie was great!!! Thanks 4 your hlp

👉 Cleaning step makes data more **machine-readable**.

Summary

- **Ambiguity** → WordNet / Context Models
- **Context** → Dependency parsing, Pragmatics
- **Sarcasm** → Transformer-based sentiment models
- **Multilingual** → Translation & Multilingual embeddings
- **Code-Switching** → Language detection + Multilingual models
- **Noisy Data** → Text preprocessing, normalization

23.4 Text Preprocessing

Raw text is messy — it has spelling errors, stopwords, mixed languages, emojis, etc.
Before using it in NLP models, we must **preprocess** and **convert** it into machine-readable format.

23.4.1 Tokenization

Definition

Breaking text into **smaller units** (tokens) like sentences, words, or subwords.

Types

1. **Sentence Tokenization** – split into sentences.
2. **Word Tokenization** – split into words.
3. **Subword Tokenization** – split words into smaller meaningful chunks (used in BERT, GPT).

Examples

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
nltk.download('punkt')

text = "NLP is amazing. It helps machines understand language."

# Sentence Tokenization
print(sent_tokenize(text))

# Word Tokenization
print(word_tokenize(text))
```

Output:

```
['NLP is amazing.', 'It helps machines understand language.']
['NLP', 'is', 'amazing', '.', 'It', 'helps', 'machines', 'understand', 'language', '.']
```

👉 Subword Tokenization (HuggingFace)

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
print(tokenizer.tokenize("unhappiness"))
```

Output:

```
['un', '##happy', '##ness']
```

23.4.2 Stopword Removal

Definition

Stopwords = very common words (the, is, an, in) that don't carry much meaning.

👉 If we keep them, they may **add noise**.

Example

```
from nltk.corpus import stopwords
words = word_tokenize("NLP is the study of human language.")
stop_words = set(stopwords.words("english"))
```

```
filtered = [w for w in words if w.lower() not in stop_words]

print(filtered)
```

Output:

```
['NLP', 'study', 'human', 'language', '.']
```

👉 Notice **“is”** and **“the”** are removed.

⚠ But in some cases (like sentiment analysis), removing stopwords may lose meaning (*“not good” → becomes “good”*). So, it depends on use case.

23.4.3 Stemming & Lemmatization

Stemming

- Rule-based, cuts word endings.
- Produces root form, but not always valid word.

Lemmatization

- Dictionary + grammar-based.
- Produces meaningful base word (lemma).

Example

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

nltk.download('wordnet')


stemmer = PorterStemmer()

lemmatizer = WordNetLemmatizer()


word = "running"

print("Stemming:", stemmer.stem(word))

print("Lemmatization:", lemmatizer.lemmatize(word, pos="v"))
```

Output:

```
Stemming: runn

Lemmatization: run
```

👉 Lemmatization is **smarter**, but slower.

23.4.4 Text Normalization

Steps

1. **Lowercasing**
2. text = "NLP Is FUN!"
3. print(text.lower())

→ nlp is fun!

4. **Removing Punctuation**
5. import re
6. text = "Hello!!! NLP, is fun??"
7. clean = re.sub(r'[^\w\s]', '', text)
8. print(clean)

→ Hello NLP is fun

9. **Spelling Correction**
10. from textblob import TextBlob
11. text = "I lovee machne learnig"
12. print(TextBlob(text).correct())

→ I love machine learning

23.4.5 Handling Emojis & Slangs

Why?

- Emojis = emotions 😍😂🔥
- Slangs = shortcuts (gr8, LOL, thx)

Example

import emoji

text = "This movie is fire 🔥 LOL 😂"

```
# Convert emoji to text
print(emoji.demojize(text))

# Replace slangs
slang_dict = {"LOL": "laughing out loud", "gr8": "great"}
words = text.split()
normalized = [slang_dict.get(w, w) for w in words]
print(" ".join(normalized))
```

Output:

```
This movie is fire :fire: LOL :face_with_tears_of_joy:
This movie is fire 🔥 laughing out loud 😂
```

23.4.6 Bag of Words (BoW)

Concept

- Represents text as **word counts**.
- Ignores grammar & word order.

Example

```
from sklearn.feature_extraction.text import CountVectorizer

docs = ["I love NLP", "NLP loves Python"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)

print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Output:

```
['love' 'loves' 'nlp' 'python']
[[1 0 1 0]
 [0 1 1 1]]
```

👉 Document 1 → has “love” and “nlp”
👉 Document 2 → has “loves”, “nlp”, “python”

23.4.7 TF–IDF (Term Frequency – Inverse Document Frequency)

Concept

- **TF** → How often a word appears in a document.
- **IDF** → How rare the word is across documents.
- TF–IDF = highlights **important words**.

Example

```
from sklearn.feature_extraction.text import TfidfVectorizer


docs = ["I love NLP", "NLP loves Python"]
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(docs)


print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Output (approx):

```
['love' 'loves' 'nlp' 'python']
[[0.71 0.00 0.71 0.00]
 [0.00 0.58 0.58 0.58]]
```

👉 TF–IDF assigns **higher weight** to rare but important words.

23.4.8 N-grams

Concept

- **N-gram = sequence of N words**.
- Helps capture context.

Example

```
vectorizer = CountVectorizer(ngram_range=(1,2))
```

```
X = vectorizer.fit_transform(["I love NLP"])

print(vectorizer.get_feature_names_out())
```

Output:

```
['i', 'love', 'nlp', 'i love', 'love nlp']
```

👉 Bigrams “I love” and “love NLP” give **phrase-level context**.

✅ **Summary**

- **Tokenization** → split text into units.
- **Stopword Removal** → drop unimportant words.
- **Stemming/Lemmatization** → normalize words.
- **Text Normalization** → lowercase, punctuation, spelling.
- **Handle Emojis/Slangs** → interpret informal language.
- **BoW & TF–IDF** → convert text to numeric features.
- **N-grams** → capture word sequence context.

23.5 Feature Representation in NLP – Ultra Detailed

Feature representation is the **process of converting text into numerical vectors** so ML/DL models can process it.

It is critical because **raw text is unstructured**, and different representations capture **different levels of meaning**:

- **Sparse representations** → simple, interpretable, but lose semantic meaning
- **Dense embeddings** → capture semantics, relationships, context

---

23.5.1 One-Hot Encoding

Concept

- Represent each word as a **binary vector** of size = vocabulary size.
- Only the position corresponding to the word is 1; all others 0.

Mathematical Representation

If V = vocabulary size, word w_i → vector x_i ∈ {0,1}^V where

x_i[j] = 1 if j = index(w_i) else 0

Example

Vocabulary = [“cat”, “dog”, “apple”]

cat → [1,0,0]

dog → [0,1,0]

apple → [0,0,1]

Pros

- Extremely simple
- Easy to implement

Cons

- High dimensional (vocabulary of 50k words → 50k-dim vector)
- Cannot capture **semantic similarity** (“cat” and “dog” unrelated)

Python Example

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np


words = np.array(["cat","dog","apple"]).reshape(-1,1)
encoder = OneHotEncoder(sparse=False)
print(encoder.fit_transform(words))
```

23.5.2 Count Vectorization (Bag of Words)

Concept

- Represents documents as **vectors of word counts**.
- Ignores grammar and word order.

Mathematical Representation

Document d → vector $x_d \in R^V$ where

x_d[j] = frequency of word w_j in document d

Example

Docs = ["I love NLP", "NLP loves Python"]
Vocabulary = ["I", "love", "loves", "NLP", "Python"]

Document	Vector
"I love NLP"	[1,1,0,1,0]
"NLP loves Python"	[0,0,1,1,1]

Python Example

```
from sklearn.feature_extraction.text import CountVectorizer


docs = ["I love NLP", "NLP loves Python"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)
print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Pros

- Simple and interpretable
- Works well for small datasets

Cons

- Ignores **word order and context**
- Synonyms (“love” vs “loves”) treated differently

23.5.3 TF–IDF (Term Frequency – Inverse Document Frequency)

Concept

- Improves Count Vectorization by **down-weighting frequent words** and **up-weighting rare, informative words**.

Formula:

$$TF\_IDF(w,d) = TF(w,d) \times \log\frac{N}{DF(w)}$$

- TF(w,d) → frequency of word in document
- DF(w) → number of documents containing word
- N → total documents

Python Example

```
from sklearn.feature_extraction.text import TfidfVectorizer


docs = ["I love NLP", "NLP loves Python"]
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(docs)
print(vectorizer.get_feature_names_out())
print(X.toarray())
```

Pros

- Captures **importance of words** in documents
- Better than simple counts

Cons

- Still **ignores order and context**

23.5.4 N-grams

Concept

- Represents text as **sequences of n words** to capture some context.
- Example:
    - Sentence: “I love NLP”
    - Unigrams: [“I”, “love”, “NLP”]
    - Bigrams: [“I love”, “love NLP”]
    - Trigrams: [“I love NLP”]

Python Example

```
vectorizer = CountVectorizer(ngram_range=(1,2))
X = vectorizer.fit_transform(["I love NLP"])
print(vectorizer.get_feature_names_out())
```

Pros

- Captures **local context**
- Works with traditional ML

Cons

- Vocabulary size explodes with higher n

23.5.5 Word Embeddings (Dense Representations)

Dense embeddings map words to **continuous vector space**, capturing **semantic similarity**.

- Word vectors have **smaller dimensions** (50–300)
- Similar words are **closer**

4.5.1 Word2Vec

- Neural network-based embeddings (Mikolov 2013)
- Two architectures:
    - **CBOW:** Predict word from context
    - **Skip-gram:** Predict context from word

Properties:

king - man + woman ≈ queen

Python Example

```
from gensim.models import Word2Vec


sentences = [["I","love","NLP"], ["NLP","loves","Python"]]

model = Word2Vec(sentences, vector_size=50, window=3, min_count=1, sg=1)

print(model.wv["NLP"])

print(model.wv.most_similar("NLP"))
```

Pros: Semantic similarity
Cons: Static embeddings

4.5.2 GloVe

- Uses **global co-occurrence matrix**
- Captures **global context**

Python Example

```
from gensim.models import KeyedVectors

glove_model = KeyedVectors.load_word2vec_format("glove.6B.100d.txt", binary=False)
```

```
print(glove_model["king"])
print(glove_model.most_similar("king"))
```

4.5.3 FastText

- Subword embeddings → splits words into **character n-grams**
- Handles **rare words, misspellings**

Python Example

```
from gensim.models import FastText

sentences = [["I","love","NLP"], ["NLP","loves","Python"]]
model = FastText(sentences, vector_size=50, window=3, min_count=1)
print(model.wv["playing"])
```

23.5.6 Contextual Embeddings

- Same word has **different embeddings depending on sentence**
- Solves limitation of Word2Vec/GloVe/FastText

ELMo

- Bi-directional LSTM
- Produces **context-dependent embeddings**

```
from allennlp.commands.elmo import ElmoEmbedder
elmo = ElmoEmbedder()
tokens = ["I","went","to","the","bank"]
vectors = elmo.embed_sentence(tokens)
print(vectors.shape)  # 3 layers x 5 tokens x 1024 dims
```

BERT Embeddings

- Transformer-based, **bidirectional attention**
- Contextual embeddings for each token

```
from transformers import BertTokenizer, BertModel
import torch

tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
model = BertModel.from_pretrained("bert-base-uncased")
sentence = "I went to the bank"
inputs = tokenizer(sentence, return_tensors="pt")
outputs = model(**inputs)
print(outputs.last_hidden_state.shape)  # [1, tokens, 768]
```

Transformer-based embeddings

- RoBERTa, GPT, XLNet, T5
- Used for **chatbots, Q&A, summarization, translation**

23.5.7 Document / Sentence Embeddings

- Represent **entire sentences/documents** as vectors

Doc2Vec

- Extends Word2Vec for paragraphs

```
from gensim.models.doc2vec import Doc2Vec, TaggedDocument
docs = [TaggedDocument(["I","love","NLP"], ["doc1"]),
    TaggedDocument(["NLP","loves","Python"], ["doc2"])]
model = Doc2Vec(docs, vector_size=50, window=2, min_count=1)
print(model.infer_vector(["NLP","rocks"]))
```

Sentence-BERT

- Fine-tuned BERT for **sentence similarity**

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
sentences = ["I love NLP", "I enjoy studying NLP"]
embeddings = model.encode(sentences)
```

```
print(embeddings.shape)  # (2, 384)
```

23.5.8 Comparison Table

Method	Type	Contextual?	Dimensionality	Pros	Cons
One-hot encoding	Sparse	❌	Vocab size	Simple	High dimensional, no meaning
Count Vectorization	Sparse	❌	Vocab size	Simple, interpretable	Ignores order, context
Word2Vec	Dense	❌	100–300	Semantic similarity	Static embeddings
GloVe	Dense	❌	100–300	Global context captured	Static embeddings
FastText	Dense	❌	100–300	Rare words handled	Larger model
ELMo	Dense	✅	1024	Contextual embeddings	Heavy, slower
BERT / Transformers	Dense	✅	768–1024	State-of-art	Expensive
Doc2Vec	Dense	Partial	100–300	Document representation	Less powerful
Sentence-BERT	Dense	✅	384–768	Sentence/document embeddings	Needs pretrained model

23.6 Classical NLP Techniques

Classical NLP techniques are **statistical and rule-based methods** used for **text analysis and understanding** before deep learning became mainstream. These techniques form the **foundation of NLP** and are still useful in resource-constrained environments.

Key Pipeline in Classical NLP:

1. Text Preprocessing → Tokenization, Stopword Removal, Lemmatization
2. Feature Representation → Bag-of-Words, TF–IDF
3. Linguistic Analysis → POS, Chunking, Parsing

4. Modeling → Classification, Topic Modeling, Sentiment Analysis

---

23.6.1 Language Modeling (n-gram models)

Goal: Predict the **next word** in a sequence.

Theory:

- Uses **Markov assumption**: probability of a word depends only on previous n–1 words.
- **n-gram model** formula:

P(w1,w2,...,wn)=∏i=1nP(wi|wi−(N−1),...,wi−1)P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_{i-(N-1)}, ..., w_{i-1})P(w1,w2,...,wn)=i=1∏nP(wi|wi−(N−1),...,wi−1)

Types of n-grams:

- **Unigram:** single words → ignores context
- **Bigram:** considers previous word
- **Trigram:** considers previous two words

Example:
Sentence: “I love NLP”

- Unigrams: [I, love, NLP]
- Bigrams: [(I,love),(love,NLP)]
- Trigrams: [(I,love,NLP)]

Python Example (Bigram probabilities with NLTK):

```
from nltk import bigrams, FreqDist

from nltk.tokenize import word_tokenize


text = "I love NLP and NLP loves Python"

tokens = word_tokenize(text)

bi_grams = list(bigrams(tokens))
```

```
freq = FreqDist(bi_grams)

print(freq[("NLP","loves")])  # frequency of bigram
```

Pros: Simple, interpretable
Cons: Sparsity for large vocabularies, limited context

Use Cases: Autocomplete, speech recognition

23.6.2 Part-of-Speech (POS) Tagging

Goal: Assign grammatical categories (noun, verb, adjective) to each word.

Theory:

- Based on **syntactic rules or statistical models**
- Common models: **Hidden Markov Models (HMM), Maximum Entropy, CRF**

Example:
Sentence: "I love NLP"

I → PRON

love → VERB

NLP → NOUN

Python Example (NLTK):

```
import nltk

nltk.download('averaged_perceptron_tagger')

from nltk import pos_tag, word_tokenize


sentence = "I love NLP"

tokens = word_tokenize(sentence)

tags = pos_tag(tokens)

print(tags)
```

Applications:

- Dependency parsing, chunking, NER

Advanced Tip:

- POS tagging accuracy improves with **domain-specific models**.

23.6.3 Named Entity Recognition (NER)

Goal: Identify **named entities** like persons, locations, organizations, dates.

Theory:

- Rule-based → Dictionary lookup
- ML-based → Conditional Random Fields, HMMs

Example:
Sentence: "Vinayak Sharma works at Google in India."

Vinayak Sharma → PERSON

Google → ORG

India → GPE

Python Example (NLTK):

```
import nltk

nltk.download('maxent_ne_chunker')

nltk.download('words')

from nltk import ne_chunk, pos_tag, word_tokenize


sentence = "Vinayak Sharma works at Google in India"

tokens = word_tokenize(sentence)

tags = pos_tag(tokens)

tree = ne_chunk(tags)

print(tree)
```

Pros: Extracts structured data from unstructured text
Cons: Hard for **unseen or ambiguous entities**

Advanced Tip: Use **pretrained SpaCy or BERT-based NER** for better accuracy.

23.6.4 Chunking & Shallow Parsing

Goal: Identify **phrases** (noun phrases, verb phrases) without full parsing.

Theory:

- Uses **POS tags** and **regex patterns**
- Example: Noun Phrase (NP) = optional determiner + adjectives + noun

Example:
Sentence: "The quick brown fox jumps over the lazy dog"

- Noun Phrases: ["The quick brown fox", "the lazy dog"]

Python Example (NLTK):

```
import nltk

grammar = "NP: {<DT>?<JJ>*<NN>}"

cp = nltk.RegexpParser(grammar)

sentence = [("The", "DT"), ("quick", "JJ"), ("fox", "NN")]

result = cp.parse(sentence)

print(result)
```

Applications:

- Information extraction, NER, relation extraction

23.6.5 Dependency Parsing

Goal: Identify **grammatical relations** between words.

Theory:

- Each word is linked to its **head word** (dependency tree)

- Useful for **semantic understanding**

Example:
Sentence: "I love NLP"

love → root

I → subject

NLP → object

Python Example (SpaCy):

```
import spacy

nlp = spacy.load("en_core_web_sm")

doc = nlp("I love NLP")

for token in doc:

    print(token.text, token.dep_, token.head.text)
```

Use Cases: Relation extraction, question answering

Advanced Tip: Combine with **NER** for knowledge graph creation.

23.6.6 Constituency Parsing

Goal: Break sentence into **nested phrases** following grammar rules.

Example:
Sentence: "The dog chased the cat"

```
(S

  (NP The dog)

  (VP chased (NP the cat)))
```

Python Example (NLTK CFG):

```
from nltk import CFG, ChartParser



grammar = CFG.fromstring("""
```

```
 S -> NP VP

 NP -> DT NN

 VP -> V NP

 DT -> 'The'

 NN -> 'dog' | 'cat'

 V -> 'chased'

""")

parser = ChartParser(grammar)

sentence = ['The','dog','chased','the','cat']

for tree in parser.parse(sentence):

   print(tree)
```

Applications:

- Machine translation, semantic parsing

23.6.7 Text Classification

Goal: Categorize text into **predefined labels**.

Common Algorithms:

- Naive Bayes → probabilistic, assumes independence
- Logistic Regression → linear classifier
- SVM → maximum margin classifier

Python Example (Naive Bayes):

```
from sklearn.feature_extraction.text import
CountVectorizer from sklearn.naive_bayes import
MultinomialNB


docs = ["I love NLP", "Python is great", "I hate bugs"]
```

```
labels = ["positive","positive","negative"]


vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)
model = MultinomialNB()
model.fit(X, labels)
print(model.predict(vectorizer.transform(["I love Python"])))
```

Advanced Tip: Use **TF–IDF** or **n-grams** for better features.

Applications: Spam detection, sentiment analysis, topic classification

---

23.6.8 Topic Modeling

Goal: Discover **hidden topics** in a document collection.

Algorithms:

1. **LDA (Latent Dirichlet Allocation)** → probabilistic, generates topic-word distributions
2. **NMF (Non-negative Matrix Factorization)** → matrix decomposition

Python Example (LDA):

```
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import CountVectorizer


docs = ["I love NLP", "Python is great", "I hate bugs"]
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(docs)


lda = LatentDirichletAllocation(n_components=2)
lda.fit(X)
```

```
print(lda.components_)  # word distribution for each topic
```

Use Cases: Document clustering, content recommendation

Advanced Tip: Preprocess text carefully (stopwords, lemmatization) for better topics.

23.6.9 Sentiment Analysis

Goal: Detect **emotion/opinion** in text.

Classical Methods:

- **Lexicon-based** → count positive/negative words
- **ML-based** → train classifier with labeled data

Python Example (VADER Lexicon):

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer
import nltk
nltk.download('vader_lexicon')

sid = SentimentIntensityAnalyzer()
text = "I love NLP, it's amazing!"
print(sid.polarity_scores(text))
```

Applications: Social media monitoring, customer feedback analysis

Advanced Tip:

- Handle **sarcasm** with context-aware models or rule-based heuristics.

23.6.10 Advanced Tips for Classical NLP

1. **Preprocessing matters**: Lowercasing, stopwords, stemming/lemmatization improve results.
2. **Feature engineering is key**: Combine TF–IDF, n-grams, POS tags for better models.
3. **Pipeline integration**: POS → Chunking → NER → Parsing → Classification

4. **Hybrid methods**: Classical NLP + Word embeddings improves performance without full deep learning.

Summary Table: Classical NLP Techniques

Technique	Type	Input	Output	Pros	Cons
Language Modeling (n-grams)	Statistical	Text	Next word probability	Simple, interpretable	Sparsity, limited context
POS Tagging	Linguistic	Sentence	Tags per word	Basis for parsing	Errors for unknown words
NER	Linguistic	Sentence	Entities with type	Extracts structured info	Struggles with unseen entities
Chunking & Shallow Parsing	Linguistic	Sentence	Phrases	Fast, interpretable	Limited grammar capture
Dependency Parsing	Linguistic	Sentence	Head-dependent graph	Captures relations	Complex, slow
Constituency Parsing	Linguistic	Sentence	Parse tree	Structured syntax	Requires grammar rules
Text Classification	ML	Documents	Class label	Widely used	Needs feature engineering

Technique	Type	Input	Output	Pros	Cons
Topic Modeling (LDA/NMF)	Unsupervised	Documents	Topic distribution	Unsupervised discovery	Topics may be hard to interpret
Sentiment Analysis	ML / Lexicon	Documents/Reviews	Sentiment score	Practical, interpretable	Context and sarcasm challenges

23.7 Deep Learning in NLP – Advanced Details

Deep learning in NLP enables models to **automatically learn semantic and syntactic patterns** from text data without heavy manual feature engineering. Unlike classical NLP, deep models **capture context, polysemy, and long-range dependencies**.

23.7.1 Word Embeddings with Neural Networks

Theory

- Represent each word as a **dense vector** of fixed dimension ddd (e.g., 100, 300, 768).
- Encodes **semantic similarity**: words with similar meaning have **vectors close in space**.

Mathematical Intuition (Word2Vec Skip-gram):
Maximize probability of context words given a target word:

max∏wt∈corpus∏wc∈context(wt)P(wc|wt)\max \prod_{w_t \in corpus} \prod_{w_c \in context(w_t)} P(w_c | w_t)maxwt∈corpus∏wc∈context(wt)∏P(wc|wt)

Softmax probability:

P(wc|wt)=exp(vwc⊤vwt)∑w=1Vexp(vw⊤vwt)P(w_c | w_t) = \frac{\exp(v_{w_c}^\top v_{w_t})}{\sum_{w=1}^{V} \exp(v_w^\top v_{w_t})}P(wc|wt)=∑w=1Vexp(vw⊤vwt)exp(vwc⊤vwt)

- vwv_wvw = vector of word www
- VVV = vocabulary size

Properties of Word Embeddings:

- **Semantic similarity:** cosine similarity measures relatedness
- **Linear relationships:** e.g., v("king")−v("man")+v("woman")≈v("queen")v("king") - v("man") + v("woman") \approx v("queen")v("king")−v("man")+v("woman")≈v("queen")

Python Example (Embedding Layer in Keras)

```
from tensorflow.keras.models import Sequential

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


vocab_size = 10000

embedding_dim = 100

sequence_length = 20


model = Sequential()

model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=sequence_length))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

model.summary()
```

Use Cases: Sentiment analysis, text classification, neural MT

23.7.2 Sequence-to-Sequence (Seq2Seq) Models

Theory

- Input → Sequence of tokens X=(x1,x2,...,xT)X = (x_1, x_2, ..., x_T)X=(x1,x2,...,xT)
- Output → Sequence Y=(y1,y2,...,yT′)Y = (y_1, y_2, ..., y_{T'})Y=(y1,y2,...,yT′)
- Encoder compresses input to **context vector ccc**
- Decoder generates output sequentially:

P(Y|X)=∏t=1T′P(yt|y1,...,yt−1,c)P(Y|X) = \prod_{t=1}^{T'} P(y_t | y_1, ..., y_{t-1}, c)P(Y|X)=t=1∏T′P(yt|y1,...,yt−1,c)

Architecture

Encoder (RNN/LSTM/GRU)

x1 x2 x3 ... xt → h_t (hidden states)

Context Vector c → summarizes sequence

Decoder (RNN/LSTM/GRU)

Generates output y1 y2 y3 ... yT'

Python Example (Keras)

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense

encoder_inputs = Input(shape=(None, num_features))
encoder = LSTM(128, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, num_features))
decoder_lstm = LSTM(128, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(num_features, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.summary()
```

Use Cases: Machine translation, summarization, question-answering

23.7.3 Attention & Self-Attention

Attention Concept

- Traditional Seq2Seq compresses all input info into **single vector** → may lose info
- Attention assigns **weights to each encoder hidden state**, focusing on relevant tokens.

Attention Score (Bahdanau):

score(st−1,hi)=vaTtanh(Wa[st−1;hi])score(s_{t-1}, h_i) = v_a^T \tanh(W_a [s_{t-1}; h_i])score(st−1,hi)=vaTtanh(Wa[st−1;hi])
αt,i=exp(score(st−1,hi))∑jexp(score(st−1,hj))\alpha_{t,i} = \frac{\exp(score(s_{t-1}, h_i))}{\sum_j \exp(score(s_{t-1}, h_j))}αt,i=∑jexp(score(st−1,hj))exp(score(st−1,hi))
contextt=∑iαt,ihicontext_t = \sum_i \alpha_{t,i} h_icontextt=i∑αt,ihi

Self-Attention

- Each token attends to **all other tokens in same sequence**
- Captures **long-range dependencies** efficiently

Python Example (Simplified Attention Layer in Keras)

```
import tensorflow as tf

from tensorflow.keras.layers import Layer


class Attention(Layer):

    def call(self, encoder_outputs):

        scores = tf.matmul(encoder_outputs, encoder_outputs, transpose_b=True)

        weights = tf.nn.softmax(scores, axis=-1)

        context = tf.matmul(weights, encoder_outputs)

        return context
```

Use Cases: Transformers, BERT, GPT

23.7.4 Transformer Architecture

Key Idea

- Eliminates RNNs/CNNs → fully relies on **self-attention** + feedforward
- **Parallelizable**, handles long sequences better than RNNs

Components

1. **Input Embeddings + Positional Encoding**

PE(pos,2i)=sin(pos/100002i/dmodel),PE(pos,2i+1)=cos(pos/100002i/dmodel)PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}), \quad PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})PE(pos,2i)=sin(pos/100002i/dmodel),PE(pos,2i+1) =cos(pos/100002i/dmodel)

2. **Encoder Block**: Multi-head self-attention + Feedforward
3. **Decoder Block**: Masked self-attention + Encoder-decoder attention + Feedforward
4. **Output Softmax** → token probabilities

Python Example (Hugging Face Transformers)

```
from transformers import BertTokenizer, TFBertModel


tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

model = TFBertModel.from_pretrained('bert-base-uncased')


inputs = tokenizer("I love NLP", return_tensors="tf")

outputs = model(inputs)

print(outputs.last_hidden_state.shape)  # (batch, tokens, hidden_size)
```

23.7.5 Pretrained Language Models

Model	Architecture	Special Features
BERT	Encoder	Bidirectional, Masked LM

Model	Architecture	Special Features
GPT	Decoder	Autoregressive LM
RoBERTa	Encoder	Optimized pretraining
XLNet	Encoder-Decoder	Permutation-based LM
T5	Encoder-Decoder	Text-to-text framework
DistilBERT	Encoder	Lighter, faster distilled BERT

Use Cases: Classification, NER, QA, summarization

23.7.6 Fine-tuning NLP Models

- Add **task-specific head** (classification, QA, NER)
- Train on **small labeled dataset** (few epochs)
- Requires careful **learning rate scheduling**

Python Example

```
from transformers import TFBertForSequenceClassification, BertTokenizer

import tensorflow as tf


tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

model = TFBertForSequenceClassification.from_pretrained("bert-base-uncased",
num_labels=2)


texts = ["I love NLP", "I hate bugs"]

inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="tf")

labels = tf.constant([1,0])
```

```
model.compile(optimizer='adam', loss=model.compute_loss)

model.fit(inputs.data, labels, epochs=2)
```

23.7.7 Transfer Learning in NLP

- Knowledge from **pretrained model** → applied to **new NLP task**
- Reduces data requirement, speeds up training, improves performance

Workflow:

1. Pretrained BERT → Masked LM
2. Fine-tune on sentiment dataset
3. Evaluate on unseen reviews

Advantages: Works well in **low-resource settings**

23.7.8 Advanced Tips

1. **Embedding choice matters**: Static embeddings vs contextual embeddings
2. **Attention visualization**: Can interpret what the model focuses on
3. **Gradient clipping & layer freezing** improve training stability
4. **Sequence length**: Longer sequences → more memory, consider truncation
5. **Batching & padding**: Essential for GPU training

23.7.9 Summary Table

Technique	Idea	Pros	Cons	Use Cases
Word Embeddings	Dense word vectors	Semantic similarity	Context-insensitive (static)	Sentiment, classification

Technique	Idea	Pros	Cons	Use Cases
Seq2Seq	Input→Output sequences	Machine translation	Long sequences hard	MT, summarization, chatbots
Attention / Self-Attention	Focus on relevant tokens	Long-range dependencies	Computationally expensive	Transformers, QA
Transformer	Attention-only, parallelizable	Efficient, state-of-the-art	GPU intensive	All modern NLP tasks
Pretrained Models	Train on large corpora, fine-tune later	High accuracy, less data	Large model sizes	Text classification, QA, NER
Fine-tuning	Adapt pretrained models	Less training data	Sensitive hyperparameters	Any NLP downstream task
Transfer Learning	Knowledge transfer across tasks	Boosts low-resource tasks	May overfit on small datasets	Domain adaptation, multilingual NLP

23.8 Tools & Libraries for NLP

Natural Language Processing (NLP) has grown rapidly due to the availability of **powerful tools, open-source libraries, deep learning frameworks, and pretrained models**. These tools reduce development time, provide ready-to-use functionality, and allow researchers and developers to focus on solving real-world problems instead of reinventing the wheel.

We can broadly divide NLP tools into five categories:

1. **Traditional Python NLP Libraries** (rule-based, statistical)
2. **Modern NLP Libraries (Deep Learning & Transformers)**
3. **Deep Learning Frameworks for NLP**
4. **Pretrained Models & Model Hubs**
5. **Speech & Audio Libraries**

23.8.1 Traditional Python NLP Libraries

These libraries provide core NLP functions like **tokenization, stemming, lemmatization, POS tagging, and corpus handling**. They are great for academic learning and lightweight NLP tasks.

(a) NLTK (Natural Language Toolkit)

- **Type:** Traditional NLP library (statistical + rule-based).
- **Best For:** Learning, small-scale NLP tasks.
- **Features:**
    - Tokenization, stemming, lemmatization.
    - POS tagging and chunking.
    - Parsing (syntax trees).
    - Access to large corpora (Brown, Gutenberg, WordNet).
- **Advantages:**
    - Excellent for education.
    - Covers many classical NLP algorithms.
- **Limitations:**
    - Slow compared to spaCy.
    - Not suited for production.

Example – POS Tagging with NLTK:

```
import nltk

nltk.download("punkt")

nltk.download("averaged_perceptron_tagger")


text = "NLP is an amazing field of AI."
```

```
tokens = nltk.word_tokenize(text)

tags = nltk.pos_tag(tokens)

print(tags)

# [('NLP', 'NNP'), ('is', 'VBZ'), ('an', 'DT'), ('amazing', 'JJ'), ('field', 'NN'), ('of', 'IN'), ('AI', 'NNP'), ('.', '.')]
```

(b) spaCy

- **Type:** Industrial-strength NLP library.
- **Best For:** Real-world, production NLP.
- **Features:**
    - Fast tokenization.
    - Part-of-Speech tagging.
    - Dependency parsing (syntactic structure).
    - Named Entity Recognition (NER).
    - Pre-trained word vectors.
- **Advantages:**
    - Blazing fast and memory efficient.
    - Works well with deep learning pipelines.
- **Limitations:**
    - Less suited for teaching basics (compared to NLTK).

Example – Named Entity Recognition with spaCy:

```
import spacy

nlp = spacy.load("en_core_web_sm")


doc = nlp("Elon Musk founded SpaceX in California.")

for ent in doc.ents:
```

```
print(ent.text, ent.label_)


# Output:
# Elon Musk PERSON
# SpaceX ORG
# California GPE
```

(c) TextBlob

- **Type:** Simple NLP library (built on NLTK & Pattern).
- **Best For:** Beginners, quick prototyping.
- **Features:**
  - Sentiment analysis.
  - POS tagging.
  - Spelling correction.
  - Translation (uses Google API).
- **Advantages:**
  - Easy syntax.
  - Ideal for non-experts.
- **Limitations:**
  - Not optimized for large data.

Example – Sentiment Analysis with TextBlob:

```
from textblob import TextBlob


text = "I love studying NLP but sometimes it is challenging."
blob = TextBlob(text)
```

```
print(blob.sentiment)

# Sentiment(polarity=0.5, subjectivity=0.6)
```

(d) Gensim

- **Type:** Topic Modeling & Similarity Library.
- **Best For:** Document similarity, embeddings, topic modeling.
- **Features:**
    - Word2Vec, FastText, Doc2Vec.
    - Latent Dirichlet Allocation (LDA) for topic modeling.
    - Large text corpus handling (streaming).
- **Advantages:**
    - Efficient for large corpora.
    - Specializes in semantic similarity.
- **Limitations:**
    - Does not handle POS/NER (use with spaCy/NLTK).

Example – Word2Vec with Gensim:

```
from gensim.models import Word2Vec


sentences = [["I", "love", "NLP"], ["NLP", "uses", "Python"], ["Python", "is", "powerful"]]

model = Word2Vec(sentences, vector_size=50, min_count=1, workers=2)


print(model.wv.most_similar("NLP"))
```

23.8.2 Modern NLP Libraries (Deep Learning & Transformers)

Traditional libraries rely on rules/statistics. Modern libraries use **neural networks and transformers**.

Hugging Face Transformers

- **Type:** Transformer-based NLP library.
- **Best For:** State-of-the-art NLP with pretrained models.
- **Features:**
    - Access to BERT, GPT, RoBERTa, T5, DistilBERT, XLNet.
    - Prebuilt pipelines for classification, NER, summarization, translation.
    - Integration with PyTorch & TensorFlow.
- **Advantages:**
    - Huge model hub (100,000+ models).
    - Easy to fine-tune.
- **Limitations:**
    - Heavy models require GPUs.

Example – Sentiment Analysis with Hugging Face:

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")

print(classifier("The movie was fantastic!"))

# [{'label': 'POSITIVE', 'score': 0.999}]
```

23.8.3 Deep Learning Frameworks for NLP

These frameworks allow **building custom NLP models** (RNNs, LSTMs, Transformers).

TensorFlow (by Google)

- High-performance deep learning framework.
- Works with **Keras** for easy model design.

- Used for NLP tasks: text classification, machine translation, seq2seq models.

PyTorch (by Meta/Facebook)

- Flexible deep learning framework.
- Preferred in **research** due to dynamic computation graphs.
- Hugging Face Transformers primarily built on PyTorch backend.

Comparison:

Aspect	TensorFlow	PyTorch
Ease of use	Good (Keras)	Better (Pythonic)
Deployment	Excellent (TF Serving, TF Lite)	Improving (TorchServe)
Research use	Medium	High
Industry use	High	High

23.8.4 Pretrained Models & Hubs

1. **Hugging Face Model Hub**
    - 100K+ pretrained NLP models.
    - Models: BERT, GPT, RoBERTa, T5.
2. **OpenAI GPT (GPT-3, GPT-4)**
    - Generative transformers for text generation, chatbots, summarization.
3. **BERT Variants**
    - **BERT** – bidirectional embeddings.
    - **RoBERTa** – robust BERT variant.
    - **DistilBERT** – lightweight BERT.
    - **ALBERT** – memory-efficient BERT.

23.8.5 Speech & Audio Libraries

NLP often overlaps with **speech recognition**.

1. **CMU Sphinx**
    - Open-source speech-to-text toolkit.
    - Works offline.
2. **Google Speech API**
    - Cloud-based service.
    - Very accurate, supports multiple languages.

Example – Google Speech API with Python:

```
import speech_recognition as sr

recognizer = sr.Recognizer()

with sr.AudioFile("sample.wav") as source:
   audio = recognizer.record(source)

print(recognizer.recognize_google(audio))
```

23.8.6 Summary

Category	Libraries / Tools	Best Use Case
Classical NLP	NLTK, spaCy, TextBlob	Tokenization, POS, Sentiment, NER
Topic Modeling	Gensim	Word2Vec, LDA, Doc2Vec
Transformer Models	Hugging Face Transformers	Summarization, Classification, NER, QA
Deep Learning Frameworks	TensorFlow, PyTorch	Building custom neural NLP models
Pretrained Models	BERT, GPT, RoBERTa, T5	State-of-the-art NLP tasks
Speech Processing	CMU Sphinx, Google Speech API	Speech-to-text, voice-enabled NLP

Chapter 24: Generative AI – Overview

24.1 Introduction to Generative AI

24.1.1 What is Generative AI?

Generative AI refers to artificial intelligence techniques that can **create new data or content** rather than just analyzing existing data. Unlike traditional AI that mostly classifies, predicts, or recommends, Generative AI can **generate text, images, audio, video, or even code** that appears human-made.

- Example: ChatGPT generating essays, MidJourney creating artwork, or GitHub Copilot writing code.
  It works by learning **patterns and structures** in data during training and then sampling from that knowledge to create new outputs.

24.1.2 Discriminative vs Generative Models

- **Discriminative Models**: Learn the boundary between classes.
    - Example: Logistic Regression, SVM classify whether an image is a “cat” or “dog.”
- **Generative Models**: Learn the **data distribution itself** and can generate new samples.
    - Example: GANs can generate a new, realistic cat image never seen before.

Key difference:

- Discriminative → Focus on classification.
- Generative → Focus on creation.

24.1.3 Applications of Generative AI

- **Chatbots & Assistants**: ChatGPT, Claude, Gemini.
- **Image Generation**: DALL·E, Stable Diffusion.
- **Video Generation**: Runway, Pika Labs.

- **Music**: MusicLM, Suno.
- **Code Generation**: GitHub Copilot, Code Llama.
- **Healthcare**: Protein folding (AlphaFold), drug discovery.
- **Education**: AI tutors, learning material generation.

24.1.4 Challenges of GenAI

- **Bias**: Models may inherit social/cultural biases from data.
- **Hallucination**: LLMs sometimes generate false but convincing answers.
- **Ethics**: Misuse for fake content, plagiarism.
- **Energy Consumption**: Training large models consumes huge electricity and computing power.

24.2 Foundations of Generative AI

24.2.1 Probability & Statistics

- **Bayes Theorem**: Used in probabilistic reasoning.
- **Distributions**: Gaussian, Bernoulli, Poisson – essential for modeling uncertainty.

24.2.2 Linear Algebra & Calculus

- **Vectors & Matrices**: Represent input data (images as pixel matrices).
- **Gradients & Derivatives**: Core of training via gradient descent.

24.2.3 Machine Learning Basics

- Models learn patterns from data to make predictions or generate content.

24.2.4 Learning Types

- **Supervised**: With labeled data (image classification).
- **Unsupervised**: No labels (clustering, dimensionality reduction).
- **Reinforcement**: Learning through rewards (game-playing AI).

24.2.5 Deep Learning Basics

- Multi-layer neural networks that learn hierarchical features.

24.2.6 Neural Networks

- Layers of neurons (input, hidden, output). Each applies weights and activation functions.

24.2.7 CNNs, RNNs, LSTMs, Transformers

- **CNNs**: Handle images.
- **RNNs**: Handle sequential data like text/speech.
- **LSTMs**: Fix vanishing gradient in RNNs.
- **Transformers**: Use attention, form the backbone of LLMs.

24.3 Generative Models

24.3.1 Probabilistic Models

- **GMMs**: Model data as a mixture of Gaussian distributions.
- **HMMs**: For sequential/temporal data like speech.
- **Bayesian Networks**: Probabilistic graphical models.

24.3.2 Neural Generative Models

- **Autoencoders (AE)**: Compress & reconstruct data.
- **Variational Autoencoders (VAE)**: Add probabilistic latent space → generate variations.
- **GANs**: Generator vs Discriminator – adversarial training.
    - Variants: DCGAN, StyleGAN, CycleGAN, Pix2Pix, BigGAN, Conditional GANs.
- **Diffusion Models**: Learn by adding noise & then denoising.
    - **DDPM**: Step-by-step denoising.
    - **Stable Diffusion**: Text-to-image breakthrough.
    - **Score-based Models**: Generate data by score matching.

24.4 Transformers and Large Language Models (LLMs)

24.4.1 Transformer Architecture

- Built on attention, parallel processing, replacing RNNs for NLP.

24.4.2 Attention & Self-Attention Mechanism

- Models focus on “important” parts of input (e.g., word relevance).

24.4.3 Encoder-Decoder Models

- Used in translation tasks (input sentence → translated output).

24.4.4 Pretrained Models

- **BERT**: Bidirectional context.
- **GPT**: Autoregressive, next-word prediction.
- **T5, XLNet, RoBERTa**: Variants with optimizations.

24.4.5 Large Language Models

- GPT-3, GPT-4, Claude, LLaMA, Gemini, Mistral – huge parameter models with conversational ability.

24.4.6 Training Techniques

- **Pretraining**: Train on huge datasets.
- **Fine-tuning**: Specialize for tasks.
- **Instruction Tuning**: Align with human instructions.
- **RLHF**: Reinforcement learning with human preferences.
- **LoRA, PEFT**: Efficient fine-tuning with fewer parameters.

24.5 Multimodal Generative AI

- **Text-to-Image**: Stable Diffusion, DALL·E, MidJourney.
- **Image-to-Image**: Super-resolution, inpainting.
- **Text-to-Video**: Runway, Pika Labs, Sora.
- **Text-to-Speech / Speech-to-Text**: Tacotron, WaveNet, Whisper.

- **Text-to-Music**: MusicLM, Suno, Udio.
- **Multi-agent Systems**: Multiple AI agents working together.

24.6 Generative AI in Applications

- **Chatbots** (ChatGPT, Gemini)
- **Content Writing** (blogs, ads)
- **Code Generation** (Copilot, Code Llama)
- **Art & Design** (DALL·E, MidJourney)
- **Healthcare** (drug discovery, AlphaFold)
- **Education** (AI tutors)
- **Gaming** (NPCs, world building)

24.7 Tools, Frameworks & Libraries

- **DL Frameworks**: TensorFlow, PyTorch, JAX.
- **GenAI Libraries**: Hugging Face, LangChain, LlamaIndex.
- **Diffusion Tools**: Stability AI, CompVis.
- **Deployment Tools**: ONNX, TensorRT, Hugging Face Spaces, OpenAI API.

24.8 Evaluation & Metrics for GenAI

- **Text**: Perplexity, BLEU, ROUGE, METEOR.
- **Images**: FID (Fréchet Inception Distance).
- **Human Evaluation**: Coherence, creativity, naturalness.
- **Bias & Toxicity Metrics**: To ensure fairness.

24.9 Ethical & Societal Aspects

- **Bias & Fairness**: Avoiding discrimination.
- **Hallucination**: Wrong but confident outputs.
- **Copyright Issues**: Training on copyrighted data.
- **Misinformation & Deepfakes**: Risk of misuse.
- **Privacy Concerns**: Sensitive data in training.
- **Responsible AI**: Ethical guidelines & transparency.

24.10 Future of Generative AI

- **Scaling Laws**: Bigger models = better (but costly).
- **Autonomous Agents**: AI agents with reasoning ability.
- **Personalized AI**: Fine-tuned for individual users.
- **Efficient Models**: Quantization, pruning, distillation to reduce size.
- **Human-AI Collaboration**: Assistants in work, art, and research.
- **Policy & Regulation**: Laws for safe use of GenAI.

Chapter 25: Final Closure on Data Science (Detailed Deep Dive)

25.1 Introduction – Why a Closure is Needed

After covering everything from **statistics → machine learning → NLP → GenAI**, we must now:

- **Consolidate Knowledge** → Connect scattered topics into a big picture.
- **Reflect** → Understand how each concept contributes to solving real-world problems.
- **Transition** → Know where Data Science is today and where it's going tomorrow.

📌 Closure = not the **end of learning**, but a **new beginning** as a practitioner.

25.2 The Data Science Lifecycle – Full Circle

Think of Data Science as an **end-to-end pipeline**, not just "modeling":

1. **Problem Definition**
    - Example: A bank wants to predict loan default.
    - Business goal → "Minimize risk" (not just get accuracy).
2. **Data Collection**
    - Sources: Databases (SQL), APIs (Twitter, weather), IoT sensors, web scraping.
    - Example: Collecting customer income, credit history, transactions.
3. **Data Cleaning & Preprocessing**
    - Handling missing values, duplicates, noise.
    - Encoding categorical data (One-Hot, Label Encoding).
    - Normalization & Standardization.
4. **Exploratory Data Analysis (EDA)**
    - Tools: Pandas, Matplotlib, Seaborn, Power BI.
    - Example: Loan default more common in low-income bracket → insight!

5. **Feature Engineering**
    - Creating new features → “Debt-to-Income ratio”.
    - Feature selection → Remove irrelevant ones.
6. **Model Building**
    - Choose algorithms → Logistic Regression, Random Forest, XGBoost.
    - Try multiple → select best.
7. **Model Evaluation**
    - Metrics: Accuracy, Precision, Recall, F1, ROC-AUC.
    - Example: Recall more important (catching loan defaulters).
8. **Deployment**
    - Deploy as API (Flask, FastAPI).
    - Serve on cloud (AWS Sagemaker, Azure ML).
9. **Monitoring & Maintenance**
    - Data drift → New patterns emerge (post-pandemic, defaults rise).
    - Retrain model.

📌 Closure: Data Science is **cyclical**. After deployment, you may need to go back to step 2 or 3.

25.3 Integration of All Concepts

Let’s connect everything you learned:

- **Math + Stats** → Foundation (probabilities, hypothesis testing, distributions).
- **Programming (Python, SQL)** → Tooling for automation + queries.
- **Machine Learning** → Predictive models (supervised, unsupervised).
- **Deep Learning** → For vision, text, complex patterns.
- **NLP & GenAI** → Making machines understand/generate human language.

- **Visualization Tools** → Tableau, Power BI for business storytelling.

📌 Closure: Think of it like a **toolbox** – use the right tool for the right job.

25.4 Skills of a Data Scientist

A **T-shaped skill set** is required:

1. **Math & Stats (depth)**
    - Probability, Bayes, linear algebra, calculus.
    - Example: Understanding why gradient descent works.
2. **Programming (breadth)**
    - Python (NumPy, pandas, scikit-learn).
    - SQL (joins, CTEs, window functions).
    - Cloud & APIs.
3. **Business + Domain Knowledge**
    - Finance, healthcare, retail, manufacturing.
    - Example: Fraud detection in banking ≠ Churn prediction in telecom.
4. **Communication & Storytelling**
    - Dashboards, presentations.
    - Translate numbers → insights → decisions.

📌 Closure: A Data Scientist is **not a coder**, but a **bridge between data & decisions**.

25.5 Challenges and Responsibilities

- **Data Quality** → Missing, biased, or unstructured data.
- **Bias & Fairness** → ML may discriminate (e.g., loan approvals biased by gender).
- **Interpretability vs Accuracy** → Deep learning is accurate but a black box.
- **Scalability** → A model that works in Jupyter may fail on millions of users.

- **Security & Privacy** → Handling sensitive data (GDPR, HIPAA).

📌 Closure: Data scientists are also **data ethicists**.

25.6 Future of Data Science

1. **AI + Automation**
    - AutoML → Non-experts building models.
    - Example: Google AutoML, H2O.ai.
2. **Real-Time Analytics**
    - Streaming platforms → Kafka, Spark Streaming.
    - Example: Fraud detection at transaction-time.
3. **Multimodal AI**
    - Models combining text + image + video.
    - Example: GPT-4 Vision (chat + image input).
4. **Responsible AI**
    - Explainable AI, fairness audits, bias detection.
5. **Human + AI Collaboration**
    - AI assists, humans validate.
    - Example: Radiology → AI scans X-ray, doctor confirms.

📌 Closure: Data Science → evolving into **Decision Science + AI Science**.

25.7 Career Roadmap

- **Entry Roles** → Data Analyst, BI Analyst, Junior DS.
- **Mid-Level** → ML Engineer, NLP Specialist, Data Engineer.
- **Senior Roles** → Lead Data Scientist, AI Product Manager, Chief Data Officer.
- **Entrepreneurship** → Start AI-driven businesses.

Key Certifications & Skills:

- AWS ML Specialty, Google TensorFlow, Azure AI.
- Tools → Hugging Face, LangChain, Power BI, SQL.

📌 Closure: Career depends on choosing **breadth vs depth**.

25.8 Mindset of a Data Scientist

- **Curiosity** → Always question data.
- **Adaptability** → Tools change, fundamentals remain.
- **Ethics** → AI must be fair, safe, and responsible.
- **Collaboration** → Work with engineers, domain experts, policymakers.
- **Lifelong Learning** → AI evolves fast, so should you.

📌 Closure: A Data Scientist = **Explorer of patterns + Creator of solutions + Guardian of responsibility**.

25.9 The One-Page Data Science Cheat Sheet

(Quick revision before interviews)

- **Lifecycle** → Problem → Data → EDA → Model → Deploy → Monitor.
- **ML Types** → Supervised | Unsupervised | Reinforcement.
- **Key Models** → Regression, Trees, SVM, Clustering, CNNs, RNNs, Transformers.
- **Metrics** → Accuracy, F1, AUC, Perplexity, BLEU, FID.
- **Tools** → Python, SQL, Power BI, PyTorch, TensorFlow, Hugging Face.
- **Future** → AutoML, Generative AI, Responsible AI.

✅ **Final Closure Statement:**
Data Science is not just about models – it's about **asking the right questions, collecting the right data, applying the right methods, and delivering insights responsibly**.
Your journey doesn't end here – it transitions into **real-world practice, continuous learning, and shaping the AI-driven future**.