Data Science Course Phase-Wise Content

This comprehensive online course is designed for students and professionals who want to gain end-to-end expertise in Data Science. The curriculum covers the **entire data science lifecycle** — including data collection, cleaning, preparation, analysis, visualization, predictive modeling, and deployment. Learners will engage with **interactive online modules, hands-on projects, and real-world case studies**, ensuring both conceptual understanding and practical application.

Phase 1: Foundations of Programming & Analytics

I. Python Programming for Data Science

Objectives

- Understand the basics of Python programming.
- Learn how to write Python scripts for data analysis and machine learning.
- Gain confidence in using Python for problem-solving.

Topics Covered & Explanations

1. Variables, Data Types, Operators

- **Explanation:**
  Variables are containers used to store data. Python supports multiple data types like integers, floats, strings, and booleans. Operators are symbols that perform operations on variables.
- **Example Code:**

```
# Variables and Data Types
name = "KenStack"
age = 20
score = 85.5
is_student = True
```

```
print(name, age, score, is_student)

# Operators
a, b = 10, 3
print("Addition:", a + b)
print("Division:", a / b)
print("Modulus:", a % b)
```

2. Lists, Tuples, Dictionaries, Sets

1. List

- **Definition:** A **list** is an ordered collection of elements.
- **Key Points:**
  - Maintains **insertion order** (the order in which you add items).
  - **Mutable** → you can add, remove, or update elements after creating the list.
  - Can hold **different data types** in one list (e.g., integers, strings, floats).
  - Supports indexing (`list[0]`) and slicing (`list[1:3]`).

```
fruits = ["apple", "banana", "cherry"]

print(fruits[0])       # Access by index

fruits.append("mango") # Add item

fruits[1] = "orange"   # Update item

print(fruits)
```

2. Tuple

- **Definition:** A **tuple** is an ordered collection of elements like a list, but **immutable**.
- **Key Points:**
  - Once created, you **cannot change** (add, remove, or update) its elements.

- Faster than lists (since they are fixed).
- Often used for **fixed data** (e.g., coordinates, dates).
- Supports indexing and slicing just like lists.

```
coordinates = (10.5, 20.3)

print(coordinates[0])    # Access by index



# coordinates[0] = 15   ❌ Error (cannot modify tuple)
```

3. Dictionary

- **Definition:** A **dictionary** stores data in **key–value pairs**.
- **Key Points:**
    - Keys must be **unique** and **immutable** (e.g., string, number, tuple).
    - Values can be of any type.
    - Data is retrieved using **keys**, not indexes.
    - Useful for representing **real-world objects** (e.g., a student profile).
- **Example:**

```
student = {

    "name": "Kalaivani",

    "age": 21,

    "course": "Data Science"

}

print(student["name"])          # Access by key

student["age"] = 22             # Update value

student["grade"] = "A"          # Add new key-value pair
```

```
print(student)
```

4. Set

- **Definition:** A **set** is an **unordered collection** of **unique elements**.
- **Key Points:**
  - No duplicate values allowed.
  - Does **not maintain order** (items may appear in different order each time).
  - Useful for mathematical operations like **union, intersection, difference**.
  - Faster membership testing (`in`) compared to lists.
- **Example:**

```
numbers = {1, 2, 2, 3, 4}

print(numbers)  # {1, 2, 3, 4} (duplicates removed)



# Set operations

a = {1, 2, 3}

b = {3, 4, 5}

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

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

print(a - b)       # {1, 2}
```

☞ **Quick Summary for Learners:**

- **List:** Best when you need an **ordered, editable collection**.
- **Tuple:** Best when you need an **ordered, fixed collection**.
- **Dictionary:** Best for **mapping relationships** (like attributes of an object).
- **Set:** Best for **unique values** and **mathematical set operations**.

3. Conditional Statements, Loops

1. Conditional Statements in Python

- Conditional statements allow your program to **make decisions** based on certain conditions.
- The keywords used are:
  - **`if`** → Checks the first condition.
  - **`elif`** → Stands for *else if*, used when there are multiple conditions.
  - **`else`** → Runs if none of the above conditions are true.

```
marks = 75

if marks >= 90:

    print("Grade A")

elif marks >= 60:

    print("Grade B")

else:

    print("Grade C")                          # Output: Grade B
```

Step-by-Step Explanation

1. **`if marks >= 90:`**
   - The program checks if the value of `marks` is **greater than or equal to 90**.
   - If this is **True**, it prints `"Grade A"` and **stops checking further conditions**.
   - If it's **False**, the program moves to the next condition.
2. **`elif marks >= 60:`**
   - Since `marks` is **75**, this condition is **True** (because $75 \geq 60$).
   - The program executes `print("Grade B").`
3. **`else:`**
   - This runs only if **all previous conditions are False**.
   - In this case, since one condition (`elif`) was already true, the program **skips the else block**.

2. Loops in Python

- Loops are used when you want to **repeat a block of code multiple times**.
- Python mainly provides two types of loops:
  - **`for` loop** → Iterates over a sequence (like list, tuple, string, or range).
  - **`while` loop** → Repeats code as long as a condition is true.

(i). For Loop

Syntax:

```
for variable in sequence:

    # code block
```

Example Code:

```
# Printing numbers from 1 to 5

for i in range(1, 6):

    print("Iteration:", i)
```

Step-by-Step Explanation:

1. `range(1, 6)` generates numbers → 1, 2, 3, 4, 5.
2. The variable `i` takes each number one by one.
3. The print statement executes for each value.

Output:

```
Iteration: 1
Iteration: 2
Iteration: 3
```

```
Iteration: 4
Iteration: 5
```

(ii). While Loop

Syntax:

```
while condition:
    # code block
```

Example Code:

```
# Printing numbers from 1 to 3
count = 1
while count <= 3:
    print("Count:", count)
    count += 1
```

Step-by-Step Explanation:

1. The loop starts with `count = 1`.
2. Condition `count <= 3` is checked:
   - If True → runs the code block.
   - If False → loop ends.
3. After each iteration, `count` increases by 1 (`count += 1`).
4. The loop stops once `count` becomes 4 (condition is False).

Output:

```
Count: 1
Count: 2
Count: 3
```

4. Functions, Modules

(i). Functions

- **Definition:** A function is a **block of reusable code** that performs a specific task.
- **Why use functions?**
  - Avoid repeating code.
  - Improve readability and organization.
  - Easy to debug and maintain.

Syntax:

```
def function_name(parameters):

    # code block

    return result
```

Example Code:

```
# Function to greet a person

def greet(name):

    return f"Hello, {name}!"



print(greet("KenStack"))

print(greet("Kalaivani"))
```

Step-by-Step Explanation:

1. `def greet(name):` → Defines a function named `greet` with a parameter `name`.
2. `return f"Hello, {name}!"` → Sends back a message with the given name.
3. `greet("KenStack")` → Calls the function and prints `"Hello, KenStack!"`.

Output:

```
Hello, KenStack!

Hello, Kalaivani!
```

(ii). Modules

- **Definition:** A module is a **Python file (.py) containing functions, variables, and classes**.
- **Why use modules?**
  - Organize code into separate files.
  - Reuse code across projects.
  - Access Python's built-in libraries.

Example Code:

```
# Using a built-in module (math)

import math

print("Square root of 16:", math.sqrt(16))

print("Value of pi:", math.pi)
```

Step-by-Step Explanation:

1. `import math` → Brings in Python's built-in `math` module.
2. `math.sqrt(16)` → Calls the square root function → result is 4.0.
3. `math.pi` → Returns the value of π (3.14159...).

Output:

```
Square root of 16: 4.0

Value of pi: 3.141592653589793
```

5. File Handling

- **Definition:** File handling allows you to **read and write data to files**.

- **Why use file handling?**
    - Store program output permanently.
    - Read input data from files (e.g., CSV, text).
    - Useful for data science when working with datasets.

Modes:

- `"r"` → Read (default)
- `"w"` → Write (overwrites file if it exists)
- `"a"` → Append (adds to existing file)

Example Code:

```
# Writing to a file

with open("sample.txt", "w") as file:

    file.write("Welcome to Data Science with Python!")



# Reading from a file

with open("sample.txt", "r") as file:

    content = file.read()

    print(content)
```

Step-by-Step Explanation:

1. `with open("sample.txt", "w") as file:` → Opens (or creates) a file in write mode.
2. `file.write("...")` → Writes text inside the file.
3. `with open("sample.txt", "r") as file:` → Opens the same file in read mode.
4. `file.read()` → Reads the content and stores it in `content`.

5. `print(content)` → Displays the stored content.

Output:

```
Welcome to Data Science with Python!
```

☞ Quick Summary for Learners:

- **Functions** → Reusable blocks of code.
- **Modules** → Organized collections of functions and variables.
- **File Handling** → Reading/writing external files for storing and retrieving data.

6. Introduction to OOPs (Object-Oriented Programming)

Introduction to OOPs (Object-Oriented Programming)

- **Definition:**
  OOP (Object-Oriented Programming) is a programming paradigm that organizes code into **objects** (real-world entities) and **classes** (blueprints of objects).
- **Why OOP?**
  - Makes code **modular** (divided into parts).
  - Promotes **reusability** (write once, use many times).
  - Easier to **manage large projects**.
  - Mimics real-world entities (e.g., a *Student* object, a *Car* object).

Key Concepts of OOP

1. **Class:** Blueprint or template to create objects.
2. **Object:** Instance of a class (actual entity).
3. **Attributes:** Variables that hold object data.
4. **Methods:** Functions that define object behavior.

Example Code: Student Class

```
# Define a class
class Student:
    # Constructor (special method to initialize object)
    def __init__(self, name, course):
        self.name = name
        self.course = course

    # Method (behavior)
    def show_details(self):
        print(f"Name: {self.name}, Course: {self.course}")

# Create objects (instances of class)
s1 = Student("Anitha", "Data Science")
s2 = Student("Rahul", "AI & ML")

# Call method on objects
s1.show_details()
s2.show_details()
```

Step-by-Step Explanation

1. **`class Student:`**
    - Defines a blueprint called `Student`.
2. **`def __init__(self, name, course):`**
    - Special method (constructor) that runs automatically when creating an object.
    - `self` refers to the current object.
    - `name` and `course` are attributes assigned to each student.
3. **`def show_details(self):`**
    - A method that prints the details of the student.
4. **`s1 = Student("Anitha", "Data Science")`**
    - Creates an object `s1` with name "Anitha" and course "Data Science".

5. **`s1.show_details()`**
    - Calls the method → prints student details.

Output

```
Name: Anitha, Course: Data Science
Name: Rahul, Course: AI & ML
```

Why OOP is Useful in Data Science?

- You can create **classes for datasets** (e.g., CSV loader class).
- You can model **real-world entities** like `Customer`, `Product`, `Model`.
- Helps in **machine learning projects** by structuring data and model pipelines.

☞ **Quick Summary for Learners:**

- **Class:** Blueprint.
- **Object:** Real instance.
- **Attributes:** Properties (data).
- **Methods:** Behaviors (functions).

Conclusion – Phase 1: Foundations of Programming & Analytics

In this phase, learners built a **strong foundation in Python programming** — the most widely used language in Data Science. Step by step, they explored:

- **Variables, Data Types, and Operators** → to represent and manipulate data.
- **Collections (List, Tuple, Dictionary, Set)** → to organize and manage datasets efficiently.
- **Conditional Statements & Loops** → to control program flow and perform iterations.

- **Functions and Modules** → to write reusable, modular code and leverage Python's libraries.
- **File Handling** → to read from and write to external files, a skill essential for working with datasets.
- **Object-Oriented Programming (OOPs)** → to model real-world entities and build structured, scalable applications.

By completing this phase, learners are now ready to **apply Python in analytics and data manipulation tasks**. These skills serve as the foundation for advanced topics in the upcoming phases — such as **data analysis, visualization, and machine learning**.

II. Statistics & Probability for Data Science

Objectives

- Build a strong statistical foundation for analyzing and interpreting data.
- Understand key probability and distribution concepts that power machine learning algorithms.
- Learn to test hypotheses and make data-driven decisions.

1. Types of Data & Sampling

- **Types of Data:**
  - **Qualitative (Categorical):** e.g., Gender (Male/Female), Colors (Red/Blue).
  - **Quantitative (Numerical):** e.g., Age, Salary, Temperature.
    - **Discrete:** Countable numbers (e.g., number of students).
    - **Continuous:** Measurable values (e.g., height, weight).
- **Sampling:**
  - Process of selecting a subset of data from a population.
  - **Methods:** Random Sampling, Stratified Sampling, Systematic Sampling.

Example:
If a university has 10,000 students, instead of asking all, we may survey 200 students chosen randomly.

---

2. Measures of Central Tendency (Mean, Median, Mode)

- **Mean:** Average value.
- **Median:** Middle value in sorted data.
- **Mode:** Most frequent value.

Example in Python:

```
import statistics as stats


data = [10, 20, 20, 30, 40]
print("Mean:", stats.mean(data))
print("Median:", stats.median(data))
print("Mode:", stats.mode(data))
```

Output:

```
Mean: 24
Median: 20
Mode: 20
```


3. Measures of Dispersion (Range, Variance, Standard Deviation)

- **Range:** Difference between max and min values.
- **Variance:** How far data points spread out from the mean.
- **Standard Deviation (σ):** Square root of variance, used often in analytics.

Example in Python:

```
import statistics as stats


data = [10, 20, 30, 40, 50]
print("Range:", max(data) - min(data))
print("Variance:", stats.variance(data))
print("Standard Deviation:", stats.stdev(data))
```

4. Probability Concepts & Bayes Theorem

- **Probability:** Likelihood of an event happening ($0 \leq P \leq 1$).
- **Bayes Theorem:** Updates probability based on new evidence.

Formula:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Example:

- Suppose 1% of emails are spam.
- If a filter catches spam 90% of the time, Bayes Theorem helps calculate the probability that an email is actually spam **given** it was flagged.

5. Probability Distributions (Binomial, Normal, Poisson)

- **Binomial Distribution:** Discrete, used for success/failure problems (e.g., coin toss).
- **Normal Distribution (Bell Curve):** Continuous, most natural events (e.g., heights, exam scores).
- **Poisson Distribution:** Models rare events (e.g., number of accidents per day).

Example in Python (Normal Distribution):

```
import numpy as np

import matplotlib.pyplot as plt


data = np.random.normal(50, 10, 1000)  # mean=50, sd=10

plt.hist(data, bins=30, edgecolor='black')

plt.title("Normal Distribution")

plt.show()
```

6. Hypothesis Testing (t-test, z-test, Chi-square test)

- **Hypothesis Testing:** A way to test assumptions about data.
  - **Null Hypothesis (H_0):** No effect or no difference.
  - **Alternative Hypothesis (H_1):** There is an effect or difference.
- **Tests:**
  - **t-test:** Compares means of two groups (small samples).
  - **z-test:** Compares means (large samples, known variance).
  - **Chi-square test:** Tests independence between categorical variables.

Example:
Checking if a new teaching method improves scores compared to traditional teaching.

Conclusion – Statistics & Probability for Data Science

In this phase, learners gain a **solid grounding in statistics and probability** — the language of data science. By mastering data types, measures of central tendency & dispersion, probability concepts, distributions, and hypothesis testing, they'll be able to **analyze datasets, draw insights, and validate findings scientifically**.

These concepts serve as the **mathematical backbone** for the upcoming phases in **Data Analysis, Visualization, and Machine Learning**.

III: Excel for Data Analysis

Objectives

- Learn how to analyze, organize, and visualize data using Microsoft Excel.
- Gain hands-on skills with formulas, functions, pivot tables, and dashboards.
- Understand how Excel can be used as a quick and powerful data analysis tool before moving to advanced tools like SQL or Python.

1. Functions: VLOOKUP, IF, COUNTIF, etc.

- **VLOOKUP:** Searches for a value in the first column and returns a value from the same row.
    - **Example:**
      Formula:

```
=VLOOKUP(101, A2:C10, 2, FALSE)
```

This looks for ID `101` in column A and returns the value from column B.

- **IF Function:** Returns one value if a condition is TRUE, another if FALSE.
    - **Example:**

```
=IF(B2>=60, "Pass", "Fail")
```

Checks if marks ≥ 60 → returns "Pass", otherwise "Fail".

- **COUNTIF:** Counts the number of cells that meet a condition.
    - **Example:**

```
=COUNTIF(C2:C20, "Data Science")
```

2. Pivot Tables, Charts, Data Cleaning

- **Pivot Tables:** Summarize large datasets easily.
  - Example: Show total sales **by product** or **by region**.
- **Charts:** Visualize data trends.
  - Types: Column, Line, Pie, Bar, Scatter.
  - Example: Monthly sales chart to see growth.
- **Data Cleaning:**
  - Remove duplicates (`Data → Remove Duplicates`).
  - Handle missing values (use filter, replace with average/blank).
  - Trim extra spaces (`=TRIM(A2)`).

3. Conditional Formatting, Dashboards

- **Conditional Formatting:** Highlight cells based on rules.
  - Example: Highlight scores ≥ 90 in green, < 50 in red.
- **Dashboards:** Combine multiple charts, tables, and KPIs in a single view.
  - Use **Pivot Tables + Charts + Slicers** to create interactive dashboards.
  - Example: A **Sales Dashboard** showing total sales, best-selling product, and regional performance.

Conclusion - Excel for Data Analysis

In this phase, learners will master **Excel as a data analysis tool** — from using powerful functions like `VLOOKUP` and `IF`, to building pivot tables and interactive dashboards. By the end, they'll be able to **clean, analyze, and visualize datasets efficiently** and present insights in a professional way.

This knowledge acts as a **bridge between basic data handling and advanced analytics tools** like SQL, Power BI, or Python.

Phase 2: Data Analysis & Visualization

IV. Data Analysis with Python

Objectives:

- Perform data cleaning, manipulation, and analysis using popular Python libraries.

1. NumPy for Numerical Computation

✍️ What is NumPy?

NumPy (Numerical Python) is a fundamental package for scientific computing with Python. It provides a powerful n-dimensional array object (ndarray) and efficient tools for performing vectorized operations like linear algebra, statistical analysis, and more.

🔑 Why NumPy?

- **Faster than native Python lists (due to C-based backend)**
- **Supports vectorized operations**
- **Useful in data analysis, ML, and scientific computing**

✅ Code Examples

```
import numpy as np


# Creating arrays

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

b = np.array([5, 6, 7, 8])


# Element-wise operations

print(a + b)        # [6 8 10 12]

print(a * b)        # [5 12 21 32]

print(np.sqrt(a))   # [1. 1.41 1.73 2.]


# Statistical operations

print(np.mean(a))   # 2.5

print(np.std(a))    # 1.118...
```

2. Pandas for Data Manipulation

✍️ What is Pandas?

Pandas is a Python library designed for data manipulation and analysis. It provides two primary data structures:

- **Series – 1D labeled array**
- **DataFrame – 2D labeled table (similar to Excel or SQL table)**

🔑 Why Pandas?

- **Easy handling of tabular data**
- **Fast filtering, transformation, and summarization**
- **Integrated with NumPy and other Python libraries**

✅ Code Examples

```
import pandas as pd


# Create DataFrame

data = {

    'Name': ['Alice', 'Bob', 'Charlie'],

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

    'Salary': [50000, 60000, 70000]

}

df = pd.DataFrame(data)


# Access columns

print(df['Name'])


# Add new column

df['Bonus'] = df['Salary'] * 0.10


# Summary statistics

print(df.describe())
```

3. Handling Missing Values and Duplicates

✍️ Why Handle Missing Data?

Real-world data is rarely clean. Missing or duplicated values can distort analysis, cause errors in models, and lead to misleading insights.

🔑 Common Strategies

- **Drop missing rows**
- **Fill missing values with:**
  - **Mean/median (for numeric)**
  - **Mode (for categorical)**
  - **Fixed values**

✅ Code Examples

```
# Create DataFrame with missing data

data = {

    'Name': ['Alice', 'Bob', 'Bob', 'Charlie', None],

    'Age': [25, None, None, 35, 40],

    'Salary': [50000, 60000, 60000, 55000, None]

}

df = pd.DataFrame(data)


# Check missing values

print(df.isnull().sum())


# Fill missing Age with mean

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


# Fill Salary with fixed value

df['Salary'].fillna(0, inplace=True)
```

```
# Remove duplicate rows
df.drop_duplicates(inplace=True)


print(df)
```

4. Data Aggregation & Grouping

✍️ What is Grouping?

Grouping allows you to **split a dataset** into groups, **apply operations** (e.g., average, sum), and **combine the results**.

Useful for:

- Summarizing data by category
- Finding trends within groups (e.g., sales by region, revenue by month)

✅ Code Examples

```
data = {
    'Department': ['HR', 'IT', 'HR', 'IT', 'Finance'],
    'Employee': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Salary': [50000, 60000, 55000, 62000, 70000]
}
df = pd.DataFrame(data)

# Group by Department and get average salary
grouped = df.groupby('Department')
print(grouped['Salary'].mean())

# Multiple aggregations
print(grouped['Salary'].agg(['mean', 'min', 'max', 'count']))
```

5. Merging & Joining DataFrames

✍️ Why Merge?

You often need to **combine multiple datasets**—e.g., merging customer info with transaction data.

Pandas supports SQL-style joins:

- inner – only matching keys
- left – all from left, match if possible
- right – all from right
- outer – all from both, fill missing with NaN

✅ Code Examples

```
# DataFrames to merge
df1 = pd.DataFrame({
    'EmpID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'EmpID': [1, 2, 4],
    'Department': ['HR', 'IT', 'Finance']
})

# Inner join
merged_inner = pd.merge(df1, df2, on='EmpID', how='inner')
print(merged_inner)

# Left join
merged_left = pd.merge(df1, df2, on='EmpID', how='left')
print(merged_left)
```

```
# Concatenation
df3 = pd.DataFrame({
    'EmpID': [5],
    'Name': ['David']
})
print(pd.concat([df1, df3], ignore_index=True))
```

🎓 Capstone Mini-Project: HR Data Cleanup and Analysis

📁 Dataset

```
raw_data = {
    'Name': ['Alice', 'Bob', 'Bob', 'Charlie', None],
    'Age': [25, None, None, 35, 40],
    'Department': ['HR', 'IT', 'IT', 'HR', 'Finance'],
    'Salary': [50000, 60000, 60000, 55000, None]
}
df = pd.DataFrame(raw_data)
```

✅ Tasks:

1. Fill missing Age with mean.
2. Fill missing Salary with group-wise (department) average.
3. Drop duplicates and null names.
4. Group by department and compute:
    - Average salary
    - Headcount
5. Merge with department-head DataFrame and export final report.

V. Data Visualization with Python

Objectives:

- Create insightful and interactive visualizations.

1. Matplotlib: Bar, Line, Pie, Histogram

✍️ **What is Matplotlib?**

Matplotlib is Python's most popular **2D plotting library**, especially good for static and publication-quality charts.

✅ **Importing**

```
import matplotlib.pyplot as plt

import numpy as np
```

🔶 **Line Plot**

```
x = np.arange(1, 11)

y = x ** 2



plt.plot(x, y, color='blue', marker='o')

plt.title("Line Plot Example")

plt.xlabel("X Values")

plt.ylabel("Y = X^2")

plt.grid(True)

plt.show()
```

🔶 **Bar Plot**

```
categories = ['A', 'B', 'C', 'D']

values = [23, 45, 56, 78]
```

```
plt.bar(categories, values, color='green')

plt.title("Bar Plot Example")

plt.xlabel("Categories")

plt.ylabel("Values")

plt.show()
```

🔶 Pie Chart

```
labels = ['Python', 'Java', 'C++', 'Ruby']

sizes = [40, 30, 20, 10]



plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)

plt.title("Pie Chart Example")

plt.axis('equal')  # Equal aspect ratio

plt.show()
```

🔶 **Histogram**

```
data = np.random.randn(1000)


plt.hist(data, bins=30, color='skyblue', edgecolor='black')
plt.title("Histogram Example")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

📘 **2. Seaborn: Box Plot, Heatmap, Pairplot**

✍️ **What is Seaborn?**

Seaborn is a high-level interface built on Matplotlib that makes beautiful and **statistically insightful plots** easily.

✅ Importing

```
import seaborn as sns
import pandas as pd
```

✅ Sample Dataset

```
df = sns.load_dataset('tips')
```

🔶 Box Plot

```
sns.boxplot(x='day', y='total_bill', data=df)
plt.title("Box Plot: Total Bill by Day")
plt.show()
```

🔶 Heatmap

```
corr = df.corr(numeric_only=True)
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```

🔶 Pairplot

```
sns.pairplot(df, hue='sex')
plt.suptitle("Pairplot Example", y=1.02)
plt.show()
```

📘 3. Plotly (Optional): Interactive Visualizations

✍️ What is Plotly?

Plotly allows you to create **interactive** visualizations with zoom, hover, tooltips, and dynamic filtering. Works in notebooks and web apps.

✅ Install and Import

```
pip install plotly
import plotly.express as px
```

🔶 Interactive Line Chart

```
df = px.data.gapminder().query("country == 'India'")
fig = px.line(df, x='year', y='gdpPercap', title='GDP per Capita in India')
fig.show()
```

🔶 Interactive Bar Chart

```
df = px.data.tips()
fig = px.bar(df, x='day', y='total_bill', color='sex', barmode='group')
fig.show()
```

📘 4. Customizing Plots & Graphs

✍️ Why Customize?

Customization improves the clarity and appearance of plots, especially for presentations or reports.

✅ Custom Elements (Matplotlib)

```
x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.figure(figsize=(8, 5))
plt.plot(x, y, color='red', linestyle='--', linewidth=2, marker='o', label='sin(x)')
plt.title("Customized Sine Wave", fontsize=14, color='blue')
plt.xlabel("X Axis", fontsize=12)
plt.ylabel("Y Axis", fontsize=12)
plt.legend()
```

```
plt.grid(True)
plt.tight_layout()
plt.show()
```

✅ Customizing with Seaborn

```
sns.set(style='whitegrid')  # Themes: white, darkgrid, ticks, etc.
sns.boxplot(x='day', y='tip', data=df, palette='pastel')
```

🧪 **Capstone Mini-Project: Data Visualization Dashboard (Matplotlib + Seaborn)**

🎯 **Goal:**

Use a dataset (e.g., Titanic, Sales, Tips) to create a **mini dashboard** with:

- Bar chart: total sales by category
- Line chart: monthly trend
- Pie chart: product shares
- Box plot: distribution of prices
- Heatmap: correlation of features

VI. SQL for Data Science

Objectives:

- Learn to query and manipulate structured data using SQL.

📘 1. SELECT, WHERE, GROUP BY, HAVING, ORDER BY

✍️ Explanation

Clause	Purpose
`SELECT`	Choose columns to display

Clause	Purpose
`WHERE`	Filter rows based on condition
`GROUP BY`	Group rows based on column(s)
`HAVING`	Filter groups (used after `GROUP BY`)
`ORDER BY`	Sort results in ascending/descending order

✅ Examples

```
-- Select specific columns
SELECT name, department, salary FROM employees;

-- Filter rows
SELECT * FROM employees WHERE salary > 50000;

-- Group and aggregate
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;

-- Filter grouped results
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 60000;

-- Sort results
SELECT name, salary FROM employees
ORDER BY salary DESC;
```

📘 2. JOINS (INNER, LEFT, RIGHT, FULL)

✍️ Explanation

Joins are used to **combine data from multiple tables** based on a related column (usually a foreign key).

Type	Description
INNER	Returns only matching rows in both tables
LEFT	All rows from left table + matched rows from right
RIGHT	All rows from right table + matched rows from left
FULL	All rows from both tables (with NULLs for missing)

✅ **Example Schema**

```
-- employees table
EmpID | Name    | DeptID
------+---------+--------
1     | Alice   | 101
2     | Bob     | 102
3     | Charlie | 103

-- departments table
DeptID | DeptName
-------+----------
101    | HR
102    | IT
104    | Finance
```

✅ Code Examples

```
-- INNER JOIN
SELECT e.Name, d.DeptName
FROM employees e
INNER JOIN departments d ON e.DeptID = d.DeptID;

-- LEFT JOIN
SELECT e.Name, d.DeptName
```

```
FROM employees e
LEFT JOIN departments d ON e.DeptID = d.DeptID;

-- RIGHT JOIN
SELECT e.Name, d.DeptName
FROM employees e
RIGHT JOIN departments d ON e.DeptID = d.DeptID;

-- FULL JOIN
SELECT e.Name, d.DeptName
FROM employees e
FULL OUTER JOIN departments d ON e.DeptID = d.DeptID;
```

📘 3. Subqueries & Aggregate Functions

✍️ Explanation

- A **subquery** is a query nested inside another query.
- **Aggregate functions** perform calculations on multiple rows.

Function	Description
COUNT()	Number of rows
SUM()	Total of a column
AVG()	Average value
MIN()	Smallest value
MAX()	Largest value

✅ Code Examples

```
-- Subquery to find employees with salary above average
SELECT name, salary
```

```
FROM employees
WHERE salary > (
    SELECT AVG(salary) FROM employees
);

-- Aggregate salary by department
SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department;
```

📘 4. Creating & Modifying Tables

✍️ Explanation

These commands are used to define and alter table structure.

✅ Creating Tables

```
CREATE TABLE employees (
    EmpID INT PRIMARY KEY,
    Name VARCHAR(100),
    Age INT,
    Salary DECIMAL(10, 2),
    DeptID INT
);
```

✅ Altering Tables

```
-- Add new column
ALTER TABLE employees ADD Email VARCHAR(100);

-- Modify column type
ALTER TABLE employees ALTER COLUMN Age SET DATA TYPE SMALLINT;
```

```
-- Drop column
ALTER TABLE employees DROP COLUMN Email;
```

✅ Inserting Data

```
INSERT INTO employees (EmpID, Name, Age, Salary, DeptID)
VALUES (1, 'Alice', 30, 55000, 101);
```

📘 5. Real-time SQL Practice on Sample Datasets

✍️ Recommended Practice Platforms

- **Mode SQL Editor**
- **SQLZoo**
- **LeetCode SQL**
- **W3Schools SQL TryIt**

✅ Sample Dataset (Employees + Departments)

```
CREATE TABLE departments (
    DeptID INT PRIMARY KEY,
    DeptName VARCHAR(100)
);

CREATE TABLE employees (
    EmpID INT PRIMARY KEY,
    Name VARCHAR(100),
    Age INT,
    Salary DECIMAL(10, 2),
    DeptID INT
);
```

```
INSERT INTO departments VALUES (101, 'HR'), (102, 'IT'), (103, 'Finance');
INSERT INTO employees VALUES
(1, 'Alice', 30, 60000, 101),
(2, 'Bob', 25, 55000, 102),
(3, 'Charlie', 28, 70000, 103),
(4, 'David', 35, 65000, NULL);
```

🎓 **Capstone Project: Company Employee Analytics Dashboard (SQL)**

📁 **Data Tables:**

- employees(EmpID, Name, Age, Salary, DeptID)
- departments(DeptID, DeptName)
- projects(ProjectID, ProjectName, DeptID)
- employee_projects(EmpID, ProjectID)

✅ **Project Tasks:**

- Create the above tables and insert sample data.
- Write queries to:
  - Show each employee's projects.
  - Find departments with the most employees.
  - Calculate average salary per department.
  - Find employees working on more than 2 projects.
  - Create a report of all employees not assigned to any project.

Phase 3: Machine Learning & Predictive Modeling

VII. Introduction to Machine Learning

Objectives:

- Understand the concepts and workflows of machine learning.

📘 1. Supervised vs Unsupervised Learning

✍️ What is Machine Learning (ML)?

Machine Learning is a subset of AI where systems learn patterns from data and make predictions or decisions **without being explicitly programmed**.

🔶 Supervised Learning

Definition:
Supervised learning uses **labeled data** to train models, meaning the input data comes with **known outputs (targets)**.

Examples:

- Predicting house prices (Regression)
- Classifying spam emails (Classification)

Algorithms:

- Linear Regression
- Logistic Regression
- Decision Trees
- Random Forests
- SVM, KNN, etc.

```
# Example: Linear Regression (Supervised)
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=100, n_features=1, noise=10)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

```

```
model = LinearRegression()
model.fit(X_train, y_train)
print("Model Score:", model.score(X_test, y_test))
```

◆ Unsupervised Learning

Definition:

Unsupervised learning works on **unlabeled data**, and the model tries to **discover patterns** (e.g., groups, anomalies).

Examples:

- Customer segmentation (Clustering)
- Dimensionality reduction
- Anomaly detection

Algorithms:

- K-Means Clustering
- PCA (Principal Component Analysis)
- Hierarchical Clustering

```
# Example: KMeans Clustering (Unsupervised)
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

X, _ = make_blobs(n_samples=200, centers=3)
model = KMeans(n_clusters=3)
model.fit(X)

plt.scatter(X[:, 0], X[:, 1], c=model.labels_)
plt.title("KMeans Clustering")
plt.show()
```

🧪 Exercise

- Identify real-world examples of supervised vs unsupervised learning in industries like banking, healthcare, and retail.
- Use sklearn.datasets.load_iris() to apply KMeans and Logistic Regression on the same data (compare results).

📘 2. ML Pipeline & Model Lifecycle

✍️ What is an ML Pipeline?

An ML pipeline is an **end-to-end process** of building, training, and deploying a machine learning model. It standardizes and automates the workflow.

🔁 Stages of the Pipeline

Step	Description
1. Data Collection	Gather raw data
2. Data Cleaning	Handle missing values, outliers
3. Feature Engineering	Create meaningful variables
4. Model Selection	Choose the right algorithm
5. Training	Feed data into the model
6. Evaluation	Use metrics to validate performance
7. Deployment	Use the model in production
8. Monitoring	Track accuracy, retrain as needed

✅ Code Example: Simple Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression())
])

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

📘 3. Train/Test Split & Cross-Validation

✍️ Why Split Data?

We split data to **train** the model on one portion and **test** its performance on unseen data to check for **overfitting**.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.datasets import make_regression

X, y = make_regression(n_samples=200, n_features=1, noise=15)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = LinearRegression()
```

```
model.fit(X_train, y_train)
preds = model.predict(X_test)

print("MSE:", mean_squared_error(y_test, preds))
```

🧠 **Best Practices:**

- Use 70-80% for training, 20-30% for testing
- Always **shuffle** the data if it's ordered

🔶 Cross-Validation (CV)

Definition:

CV splits the dataset into **k folds**, trains the model on (k-1) folds, and tests it on the remaining one — repeating this k times.

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)
model = LogisticRegression()

scores = cross_val_score(model, X, y, cv=5)
print("Cross-Validation Accuracy:", scores)
print("Mean Accuracy:", scores.mean())
```

🧪 Capstone Activity: Model Building Mini-Project

Dataset:

Use a dataset like **Titanic**, **Iris**, or **Loan Prediction**.

Tasks:

1. Load and clean data
2. Split into train/test
3. Build a pipeline (scaler + model)
4. Use cross-validation
5. Report final accuracy and observations

VIII. Supervised Learning Algorithms

Objective:

Learn and apply core machine learning algorithms for regression and classification using Python, including model selection, implementation, and evaluation.

📘 1. Linear Regression & Logistic Regression

🔶 Linear Regression

Used for: Predicting **continuous values**
Example: Predict house prices, salary, etc.

✍️ **Concept:**

Fits a line to data using the equation:

$$y = \beta_0 + \beta_1 x + \epsilon$$

✅ **Code Example**

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error
```

```
X, y = make_regression(n_samples=100, n_features=1, noise=10)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = LinearRegression()
model.fit(X_train, y_train)
preds = model.predict(X_test)

print("MSE:", mean_squared_error(y_test, preds))
```

🔶 **Logistic Regression**

Used for: Binary classification problems
Example: Spam detection, disease prediction

✍️ **Concept:**

Applies a **sigmoid function** to output a probability

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

✅ **Code Example**

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

data = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target)

model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
preds = model.predict(X_test)
```

```
print("Accuracy:", accuracy_score(y_test, preds))
```

📘 2. Decision Tree & Random Forest

🔶 Decision Tree

Used for: Classification or Regression

Intuition: Tree structure where each node represents a decision based on feature value

✅ Code Example (Classifier)

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)

model = DecisionTreeClassifier(max_depth=3)
model.fit(X_train, y_train)

print("Accuracy:", model.score(X_test, y_test))
```

📌 Visualize:

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

plt.figure(figsize=(12, 6))
plot_tree(model, feature_names=load_iris().feature_names,
class_names=load_iris().target_names)
plt.show()
```

🔶 **Random Forest**

Used for: Ensemble model for classification/regression
How it works: Combines **multiple decision trees** to improve performance

✅ **Code Example**

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)

print("Accuracy:", model.score(X_test, y_test))
```

📘 3. K-Nearest Neighbors (KNN)

✍️ What is KNN?

- **Instance-based** learning algorithm.
- Predicts output by finding **K closest data points** in training data.

✅ **Code Example**

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)

model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

print("Accuracy:", model.score(X_test, y_test))
```

📌 You can experiment with:

```
# Try different values of K
for k in range(1, 11):
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_train, y_train)
    print(f"K={k}, Accuracy: {model.score(X_test, y_test):.2f}")
```

📘 4. Support Vector Machine (SVM)

✍️ What is SVM?

SVM finds the **best hyperplane** that separates classes with **maximum margin**.

- Works well for **high-dimensional** and **non-linear** data
- Can use different **kernels**: linear, polynomial, RBF

✅ Code Example

```
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)

model = SVC(kernel='linear')  # Try 'rbf', 'poly', etc.
model.fit(X_train, y_train)

print("Accuracy:", model.score(X_test, y_test))
```

🧪 Capstone Activity: Model Comparison on Real Dataset

Dataset: Iris, Titanic, or Breast Cancer

Tasks:

1. Preprocess the dataset (cleaning, encoding, scaling).
2. Train and evaluate:
    - Logistic Regression
    - Decision Tree
    - Random Forest
    - KNN
    - SVM
3. Compare models using accuracy, precision, recall.
4. Plot confusion matrix and classification report.

IX. Unsupervised Learning Algorithms

Objective :

Learn to identify patterns and structure in unlabeled data using clustering and dimensionality reduction techniques.

1️⃣ K-Means Clustering

✍️ What is K-Means?

K-Means is a **centroid-based clustering algorithm** that groups data into **K clusters** based on feature similarity.

- It's **unsupervised** (no labels)
- Each cluster is defined by its **centroid (mean)**
- Goal: Minimize distance of points to their cluster center

✅ Steps:

1. Choose number of clusters (K)
2. Initialize K centroids randomly
3. Assign points to the nearest centroid
4. Update centroids based on assigned points
5. Repeat until convergence

✅ Code Example

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Create synthetic data
X, _ = make_blobs(n_samples=300, centers=3, random_state=42)

# Apply KMeans
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)

# Plot clusters
plt.scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], color='red',
marker='x')
plt.title("K-Means Clustering")
plt.show()
```

2 Hierarchical Clustering

✍️ What is it?

Hierarchical Clustering builds a hierarchy of clusters by either:

- **Agglomerative** (bottom-up) – start with individual points and merge
- **Divisive** (top-down) – start with all data and split

✅ Key Concept:

The output is a **dendrogram**, which visually represents how clusters are formed.

✅ Code Example

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load data
X = load_iris().data

# Generate linkage matrix
linked = linkage(X, method='ward')

# Plot dendrogram
plt.figure(figsize=(10, 6))
dendrogram(linked, orientation='top', distance_sort='descending', show_leaf_counts=False)
plt.title("Hierarchical Clustering Dendrogram")
plt.show()
```

3️⃣ Dimensionality Reduction (PCA)

✍️ What is PCA?

Principal Component Analysis (PCA) is used to reduce the **number of features** (dimensions) while preserving most of the **variance** in the data.

- Helps in **visualization**, **speeding up training**, and removing noise.

- PCA transforms the original data into **principal components**.

✅ Code Example

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

# Load data
X = load_iris().data
y = load_iris().target

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualize
plt.figure(figsize=(8, 5))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis')
plt.title("PCA - Iris Dataset (2D)")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.grid(True)
plt.show()
```

✅ Use Cases:

- Preprocessing before clustering
- Speeding up model training
- Visualizing high-dimensional data

🎓 Capstone Activity: Customer Segmentation

Dataset: Mall Customers Dataset (Mall_Customers.csv)

Tasks:

1. Perform EDA on age, income, spending score.
2. Use K-Means to segment customers into 3-5 groups.
3. Visualize results using PCA.
4. Interpret segments (e.g., "high income, low spending").

X. Model Evaluation & Tuning

Objective:

Learn to evaluate machine learning models using classification metrics and improve their performance through hyperparameter tuning techniques.

1️⃣ Confusion Matrix, Accuracy, Precision, Recall

✍️ What is a Confusion Matrix?

A confusion matrix shows how many predictions were:

- Correct (True Positives and True Negatives)
- Incorrect (False Positives and False Negatives)

✅ Confusion Matrix Format

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

📊 Common Metrics:

- **Accuracy** = (TP + TN) / Total
- **Precision** = TP / (TP + FP)
- **Recall** = TP / (TP + FN)
- **F1 Score** = 2 × (Precision × Recall) / (Precision + Recall)

✅ Code Example

```
from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import confusion_matrix, classification_report


# Load data

X, y = load_breast_cancer(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(X, y)


# Train model

model = LogisticRegression(max_iter=1000)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)


# Confusion Matrix

print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))


# Full report

print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

2 ROC & AUC Curve

✍ What is ROC?

- **ROC (Receiver Operating Characteristic)** curve plots:

- True Positive Rate (Recall) vs.
- False Positive Rate

✅ **AUC (Area Under Curve):**

- Measures **overall model performance** regardless of classification threshold.
- **AUC = 1** → perfect model
- **AUC = 0.5** → random guessing

✅ **Code Example**

```
from sklearn.metrics import roc_curve, roc_auc_score

import matplotlib.pyplot as plt



# Get probabilities

y_prob = model.predict_proba(X_test)[:, 1]



# ROC Curve

fpr, tpr, _ = roc_curve(y_test, y_prob)



# Plot

plt.plot(fpr, tpr, label="ROC Curve")

plt.plot([0, 1], [0, 1], '--', color='gray')

plt.xlabel("False Positive Rate")

plt.ylabel("True Positive Rate (Recall)")

plt.title("ROC Curve")

plt.legend()

plt.grid()

plt.show()
```

```
# AUC Score

print("AUC Score:", roc_auc_score(y_test, y_prob))
```

3️⃣ Hyperparameter Tuning (Grid Search, Random Search)

✍️ Why Tune Hyperparameters?

- Algorithms have **parameters you set manually** (e.g., C, max_depth, n_neighbors)
- Tuning helps find the best combination for **optimal performance**

🔧 Grid Search

Tries **all possible combinations** of specified hyperparameters.

```
from sklearn.model_selection import GridSearchCV


params = {'C': [0.1, 1, 10], 'solver': ['liblinear', 'lbfgs']}

grid = GridSearchCV(LogisticRegression(max_iter=1000), params, cv=5)

grid.fit(X_train, y_train)


print("Best Params:", grid.best_params_)

print("Best Score:", grid.best_score_)
```

🔍 Random Search

Randomly samples combinations for a **faster search** when grid search is too slow.

```
from sklearn.model_selection import RandomizedSearchCV

from scipy.stats import uniform


params = {'C': uniform(0.1, 10)}
```

```
rand = RandomizedSearchCV(LogisticRegression(max_iter=1000), params, n_iter=10,
cv=5, random_state=42)

rand.fit(X_train, y_train)



print("Best Params:", rand.best_params_)
```

🎓 **Capstone Activity: Model Optimization Challenge**

Dataset: Breast Cancer or Titanic Dataset

Tasks:

1. Train a baseline model (e.g., Logistic Regression or Random Forest)
2. Evaluate performance using:
    - Accuracy, Precision, Recall
    - Confusion Matrix
    - ROC & AUC
3. Tune model using:
    - Grid Search or Random Search
4. Compare before vs. after tuning

Phase 4: Project & Deployment

XI. Real-Time Projects

Objective

These projects help learners apply their skills in real-world business contexts.

📌 **1. Sales Prediction Model**

Objective:
Build a regression model to forecast future sales based on historical data, seasonality, and promotional campaigns.

Key Concepts:

- Linear Regression / Random Forest Regressor
- Feature engineering (date/time, promotions)
- Handling missing data
- Time series or tabular data

Deliverables:

- Train/test split
- Model accuracy (RMSE, MAE)
- Visualize predicted vs. actual sales

📌 2. Customer Segmentation

Objective:
Use clustering techniques to group customers based on spending behavior, demographics, and product preferences.

Key Concepts:

- K-Means, PCA, Hierarchical Clustering
- Data preprocessing (scaling, encoding)
- Elbow method for optimal K
- Visualization using PCA (2D plot)

Deliverables:

- Cluster profiles (e.g., High-spending young adults)
- Segmentation dashboard or charts
- Business insights for targeted marketing

📌 3. Employee Attrition Prediction

Objective:
Build a classification model to predict which employees are at risk of leaving the company.

Key Concepts:

- Logistic Regression, Decision Tree, or Random Forest
- Classification metrics (confusion matrix, ROC-AUC)
- Feature importance (satisfaction, workload, tenure)
- Hyperparameter tuning

Deliverables:

- Accuracy, precision, recall, F1-score
- ROC Curve
- Insights for HR to reduce attrition

📌 4. E-commerce Analytics Dashboard

Objective:
Design a dashboard to analyze e-commerce KPIs like revenue, conversion rate, top products, and customer trends.

Key Concepts:

- Data manipulation using Pandas
- Data visualization using Matplotlib, Seaborn, Plotly
- KPI definitions (AOV, RPV, churn rate)
- Optional: Use Dash or Streamlit for interactive UI

Deliverables:

- Interactive dashboard or static report
- Charts for sales trends, category performance
- Executive summary of insights

XII. Deployment Basics (Optional)

🎯 Objective (One-liner):

Learn how to deploy machine learning models using lightweight web frameworks, host them online, and manage version control using GitHub.

🔶 1. Introduction to Streamlit or Flask

✍️ Concept:

After building an ML model, deployment makes it accessible to users via a **web interface**. Two popular Python tools:

- **Streamlit** → Quick, no-hassle dashboarding (for data apps)
- **Flask** → Lightweight web framework (for flexible APIs)

✅ Example: Streamlit App for Predicting House Prices

```
# Save this as app.py

import streamlit as st
```

```
import pickle


# Load trained model

model = pickle.load(open("model.pkl", "rb"))


st.title("🏠 House Price Predictor")

sqft = st.slider("Enter square footage", 500, 5000)


if st.button("Predict"):

    prediction = model.predict([[sqft]])

    st.success(f"Predicted Price: ${prediction[0]:,.2f}")
```

To run:

```
streamlit run app.py
```

✅ Example: Flask API for Model Deployment

```
# Save this as app.py

from flask import Flask, request, jsonify

import pickle


app = Flask(__name__)

model = pickle.load(open("model.pkl", "rb"))


@app.route("/predict", methods=["POST"])

def predict():

    data = request.get_json()
```

```
    prediction = model.predict([data['features']])

    return jsonify({"prediction": prediction[0]})


if __name__ == "__main__":

    app.run(debug=True)
```

🔶 2. Hosting a Model on the Web

✍️ Concept:

You can **host your Streamlit/Flask app online** to make it publicly accessible.

✅ Common Hosting Platforms:

- 🔷 **Streamlit Cloud** — easiest for Streamlit apps
- 🔷 **Render.com** — free Flask hosting
- 🔷 **Heroku** — supports both Flask and Streamlit
- 🔷 **Replit** — great for quick deployments
- 🔷 **Hugging Face Spaces** — for model demos

✅ Steps to Host (Streamlit on Streamlit Cloud):

1. Push your app to GitHub
2. Go to https://streamlit.io/cloud
3. Connect your GitHub repo
4. Deploy!

Make sure your repo has:

- app.py
- requirements.txt (e.g., streamlit, scikit-learn, etc.)
- model.pkl

🔶 3. Using GitHub for Version Control

✍️ Concept:

GitHub lets you **track code changes**, **collaborate**, and **store projects**.

✅ Basic Git Commands

```
git init                    # Initialize repo

git add .                   # Stage all changes

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/username/repo.git

git push -u origin main    # Push code to GitHub
```