CS –37: Machine Learning with Python

Created	January 18, 2026
Project	BCA-6 CS –37: Machine Learning with Python Minimum following exercise should be performed by the students during the semester
Prepared By	VISHAL MAKWANA (KAMANI SCIENCE & PRATAPRAI ARTS COLLEGE, AMRELI) under the guidance of Prof. V.A.BALDHA

(1) Write a Python program/script to make a Pandas DataFrame with two-dimensional list

Github Link: exercise1.py

```
# Import pandas library
import pandas as pd

# Two-dimensional list (rows and columns)
data = [
    [101, "Amit", 85],
    [102, "Neha", 92],
    [103, "Raj", 78]
]

# Create DataFrame with column names
df = pd.DataFrame(data, columns=["Roll No", "Name", "Marks"])

# Display the DataFrame
print(df)
```

Output:

```
   Roll No  Name  Marks
0      101  Amit     85
1      102  Neha     92
2      103   Raj     78
```

(2) Write a Python program to Create a pandas column using for loop

Github Link: exercise2.py

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({
    "Number": [1, 2, 3, 4, 5]
})

# Empty list to store values for the new column
squares = []

# Use for loop to calculate square of each number
for n in df["Number"]:
    squares.append(n * n)

# Add the list as a new column in DataFrame
df["Square"] = squares

# Display the updated DataFrame
print(df)
```

Output:

```
   Number  Square
0       1       1
1       2       4
2       3       9
3       4      16
4       5      25
```

(3) Write a Python program toChange column names and row indexes in Pandas DataFrame

Github Link: exercise3.py

```
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    "Name": ["Amit", "Neha", "Raj"],
    "Age": [20, 21, 22],
    "City": ["Surat", "Ahmedabad", "Vadodara"]
})

print("Original DataFrame:\n", df)

# 1) Change column names
df.columns = ["Student_Name", "Student_Age", "Student_City"]

# OR (another method)
# df.rename(columns={"Name": "Student_Name", "Age": "Student_Age",
"City": "Student_City"}, inplace=True)

# 2) Change row indexes
df.index = ["S1", "S2", "S3"]

# OR (another method)
# df.rename(index={0: "S1", 1: "S2", 2: "S3"}, inplace=True)

print("\nUpdated DataFrame:\n", df)
```

Output:

```
Original DataFrame:
   Name  Age       City
0  Amit   20      Surat
1  Neha   21  Ahmedabad
2   Raj   22   Vadodara

Updated DataFrame:
   Student_Name  Student_Age Student_City
S1         Amit           20        Surat
S2         Neha           21    Ahmedabad
S3          Raj           22     Vadodara
```

(4) Write a Python program to Load different kind of datasets using scikit-learn library

Github Link: exercise4.py

```
# Program: Load different kinds of datasets using scikit-learn library

from sklearn.datasets import load_iris, load_digits, load_diabetes,
load_breast_cancer

# 1) Load Iris dataset (classification dataset)
iris = load_iris()
print("1) IRIS DATASET (Classification)")
print("Data shape:", iris.data.shape)
print("Target shape:", iris.target.shape)
print("First 3 target values:", iris.target[:3])
print("-" * 50)

# 2) Load Digits dataset (image classification dataset)
digits = load_digits()
print("2) DIGITS DATASET (Image Classification)")
print("Data shape:", digits.data.shape)
print("Target shape:", digits.target.shape)
print("First 3 target values:", digits.target[:3])
print("-" * 50)

# 3) Load Diabetes dataset (regression dataset)
diabetes = load_diabetes()
print("3) DIABETES DATASET (Regression)")
print("Data shape:", diabetes.data.shape)
print("Target shape:", diabetes.target.shape)
print("First 3 target values:", diabetes.target[:3])
print("-" * 50)

# 4) Load Breast Cancer dataset (classification dataset)
cancer = load_breast_cancer()
print("4) BREAST CANCER DATASET (Classification)")
print("Data shape:", cancer.data.shape)
print("Target shape:", cancer.target.shape)
print("First 3 target values:", cancer.target[:3])
print("-" * 50)
```

Output:

```
1) IRIS DATASET (Classification)
Data shape: (150, 4)
Target shape: (150,)
First 3 target values: [0 0 0]
--------------------------------------------------
2) DIGITS DATASET (Image Classification)
Data shape: (1797, 64)
Target shape: (1797,)
First 3 target values: [0 1 2]
--------------------------------------------------
3) DIABETES DATASET (Regression)
Data shape: (442, 10)
Target shape: (442,)
First 3 target values: [151.  75. 141.]
--------------------------------------------------
4) BREAST CANCER DATASET (Classification)
Data shape: (569, 30)
Target shape: (569,)
First 3 target values: [0 0 0]
--------------------------------------------------
```

Explanation:

This program shows how to **load different kinds of datasets** using the **scikit-learn library**:

- **Iris** → Classification dataset (predict flower class)
- **Digits** → Image classification dataset (predict digit 0–9)
- **Diabetes** → Regression dataset (predict continuous value)
- **Breast Cancer** → Classification dataset (predict malignant/benign)

For each dataset, the program prints:

- **Data shape** (rows, columns)
- **Target shape** (number of labels/outputs)
- **First few target values** (to understand output format)

(5) Write a Python program to Extract the specified rows and columns from the dataset using Pandas

Github Link: exercise5.py

```
import pandas as pd

# Step 1: Create a sample dataset (DataFrame)
df = pd.DataFrame({
    "RollNo": [101, 102, 103, 104, 105],
    "Name": ["Amit", "Neha", "Raj", "Priya", "Karan"],
    "Age": [20, 21, 22, 20, 23],
    "City": ["Surat", "Ahmedabad", "Vadodara", "Rajkot", "Bhavnagar"],
    "Marks": [78, 85, 69, 92, 74]
})

print("Original Dataset:\n")
print(df)

# Step 2: Extract specified rows (by index) and specified columns (by names)
# Example: extract rows 1 to 3 (index 1,2,3) and columns Name, City, Marks
extracted = df.loc[1:3, ["Name", "City", "Marks"]]

print("\nExtracted rows (1 to 3) and columns (Name, City, Marks):\n")
print(extracted)
```

Output:

```
Original Dataset:

   RollNo   Name  Age       City  Marks
0     101   Amit   20      Surat     78
1     102   Neha   21  Ahmedabad     85
2     103    Raj   22   Vadodara     69
3     104  Priya   20     Rajkot     92
4     105  Karan   23  Bhavnagar     74

Extracted rows (1 to 3) and columns (Name, City, Marks):

    Name       City  Marks
1   Neha  Ahmedabad     85
2    Raj   Vadodara     69
3  Priya     Rajkot     92
```

(6) Write a Python program toHandle missing values using Imputer class with mean strategy

Github Link: exercise6.py

```python
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer

# Step 1: Create a sample dataset with missing values (NaN)
df = pd.DataFrame({
    "Age": [20, 21, np.nan, 23, np.nan],
    "Marks": [78, np.nan, 69, 92, 74]
})

print("Original Dataset (with missing values):")
print(df)

# Step 2: Create Imputer with mean strategy
imputer = SimpleImputer(strategy="mean")

# Step 3: Fit and transform the data (replace NaN with column mean)
filled_data = imputer.fit_transform(df)

# Step 4: Convert back to DataFrame for easy display
df_filled = pd.DataFrame(filled_data, columns=df.columns)

print("\nDataset After Handling Missing Values (Mean Strategy):")
print(df_filled)

# Step 5: Show the means used
print("\nMeans used to fill missing values:")
print("Age mean  =", imputer.statistics_[0])
print("Marks mean=", imputer.statistics_[1])
```

Output:

```
Original Dataset (with missing values):
    Age  Marks
0  20.0   78.0
1  21.0    NaN
2   NaN   69.0
3  23.0   92.0
4   NaN   74.0

Dataset After Handling Missing Values (Mean Strategy):
         Age  Marks
0  20.000000  78.00
1  21.000000  78.25
2  21.333333  69.00
3  23.000000  92.00
4  21.333333  74.00

Means used to fill missing values:
Age mean  = 21.333333333333332
Marks mean= 78.25
```

Explanation:

1. **Missing values (NaN)**
    - `NaN` means data is missing in that cell.
2. **`SimpleImputer(strategy="mean")`**
    - This creates an imputer that will replace missing values with the **mean (average)** of that column.
3. **`fit_transform(df)`**
    - `fit` finds the mean of each column (ignoring NaN).
    - `transform` replaces NaN with the calculated mean.
4. **Result**
    - Missing **Age** values are replaced with the Age mean.
    - Missing **Marks** values are replaced with the Marks mean.

(7) Write a Python program to Encode categorial data using label encodingtechnique

Github Link: exercise7.py

```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Step 1: Create a sample dataset with categorical values
df = pd.DataFrame({
    "City": ["Surat", "Ahmedabad", "Vadodara", "Surat", "Rajkot"],
    "Grade": ["A", "B", "A", "C", "B"]
})

print("Original Dataset:")
print(df)

# Step 2: Create LabelEncoder objects
city_encoder = LabelEncoder()
grade_encoder = LabelEncoder()

# Step 3: Apply label encoding (convert text categories into numbers)
df["City_Encoded"] = city_encoder.fit_transform(df["City"])
df["Grade_Encoded"] = grade_encoder.fit_transform(df["Grade"])

print("\nDataset After Label Encoding:")
print(df)

# Step 4: Show mapping (which category became which number)
print("\nCity Encoding Mapping:")
for i, name in enumerate(city_encoder.classes_):
    print(name, "->", i)

print("\nGrade Encoding Mapping:")
for i, name in enumerate(grade_encoder.classes_):
    print(name, "->", i)
```

Output:

```
Original Dataset:
        City Grade
0      Surat     A
1  Ahmedabad     B
2   Vadodara     A
3      Surat     C
4     Rajkot     B

Dataset After Label Encoding:
        City Grade  City_Encoded  Grade_Encoded
0      Surat     A             2              0
1  Ahmedabad     B             0              1
2   Vadodara     A             3              0
3      Surat     C             2              2
4     Rajkot     B             1              1

City Encoding Mapping:
Ahmedabad -> 0
Rajkot -> 1
Surat -> 2
Vadodara -> 3

Grade Encoding Mapping:
A -> 0
B -> 1
C -> 2
```

Explanation:

1. **Categorical data**
    - Data like **City, Grade** is text, so ML models cannot directly work with it.
2. **Label Encoding**
    - Label encoding converts each unique text category into a number.
    - Example: `A -> 0`, `B -> 1`, `C -> 2`
3. **`fit_transform()`**
    - `fit` learns all unique categories.

- `transform` converts them into numeric labels.

4. **Mapping**
    - `classes_` shows which category got which number.

(8) Write a Python program toEncode categorial data using one hot encodingtechnique

Github Link: exercise8.py

```
import pandas as pd

# Step 1: Create a sample dataset with categorical columns
df = pd.DataFrame({
    "City": ["Surat", "Ahmedabad", "Vadodara", "Surat", "Rajkot"],
    "Grade": ["A", "B", "A", "C", "B"]
})

print("Original Dataset:")
print(df)

# Step 2: Apply One-Hot Encoding using pandas get_dummies()
encoded_df = pd.get_dummies(df, columns=["City", "Grade"])

print("\nDataset After One-Hot Encoding:")
print(encoded_df)
```

Output:

```
Original Dataset:
        City Grade
0      Surat     A
1  Ahmedabad     B
2   Vadodara     A
3      Surat     C
4     Rajkot     B

Dataset After One-Hot Encoding:
   City_Ahmedabad  City_Rajkot  City_Surat  City_Vadodara  Grade_A  Grade_B  Grade_C
0           False        False        True          False     True    False    False
1            True        False       False          False    False     True    False
2           False        False       False           True     True    False    False
3           False        False        True          False    False    False     True
4           False         True       False          False    False     True    False
```

Explanation:

1. **Why One-Hot Encoding?**
   ML models need numbers. One-hot encoding converts each category into **separate columns**.

2. **How it works?**

   - `City` has 4 unique values → 4 new columns: `City_Ahmedabad`, `City_Rajkot`, `City_Surat`, `City_Vadodara`

   - `Grade` has 3 unique values → 3 new columns: `Grade_A`, `Grade_B`, `Grade_C`

3. **Meaning of True/False**

   - `True` means that category is present in that row

   - `False` means not present
     (Sometimes output can also come as 1/0 depending on pandas version/settings.)

(9) Write a Python program to splitting dataset into Training set and Test set

Github Link: exercise9.py

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

# Step 1: Load dataset
data = load_iris()
X = data.data      # features (input)
y = data.target    # labels (output)

print("Total data before split:")
print("X shape:", X.shape)
print("y shape:", y.shape)

# Step 2: Split dataset into Training and Test set
# test_size=0.20 means 20% data for testing, 80% for training
# random_state is used to get same output every time
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, random_state=42
)

# Step 3: Print results of split
print("\nAfter split:")
print("Training set X_train shape:", X_train.shape)
print("Training set y_train shape:", y_train.shape)

print("Test set X_test shape:", X_test.shape)
print("Test set y_test shape:", y_test.shape)

# Step 4: Show a small sample
print("\nFirst 3 training labels:", y_train[:3])
print("First 3 test labels:", y_test[:3])
```

Output:

```
Total data before split:
X shape: (150, 4)
y shape: (150,)

After split:
Training set X_train shape: (120, 4)
Training set y_train shape: (120,)

Test set X_test shape: (30, 4)
Test set y_test shape: (30,)

First 3 training labels: [1 0 2]
First 3 test labels: [1 0 2]
```

Explanation:

1. **Why split dataset?**
    - **Training set** is used to *train/teach* the model.
    - **Test set** is used to *check accuracy* on unseen data.
2. `train_test_split()`
    - Splits `X` and `y` into 4 parts:
        - `X_train`, `y_train` (training data)
        - `X_test`, `y_test` (testing data)
3. `test_size=0.20`
    - 20% data goes to test set, 80% to training set.
4. `random_state=42`
    - Keeps the split same every time you run the program (useful for consistent output).

(10) Write a Python program toPerform feature scaling using standardizationtechnique

Github Link: exercise10.py

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Step 1: Create a sample dataset (different feature ranges)
df = pd.DataFrame({
    "Age": [18, 20, 22, 24, 26],
    "Salary": [20000, 25000, 30000, 35000, 40000]
})

print("Original Dataset:")
print(df)

# Step 2: Create StandardScaler object
scaler = StandardScaler()

# Step 3: Apply standardization (fit + transform)
scaled_values = scaler.fit_transform(df)

# Step 4: Convert scaled result back to DataFrame
scaled_df = pd.DataFrame(scaled_values, columns=df.columns)

print("\nDataset After Standardization (Feature Scaling):")
print(scaled_df.round(3))
```

Output:

```
Original Dataset:
   Age  Salary
0   18   20000
1   20   25000
2   22   30000
3   24   35000
4   26   40000

Dataset After Standardization (Feature Scaling):
     Age  Salary
0 -1.414  -1.414
1 -0.707  -0.707
2  0.000   0.000
3  0.707   0.707
4  1.414   1.414
```

Explanation:

1. **Why feature scaling?**
   Here **Age** is small (18–26) but **Salary** is very large (20000–40000).
   Many ML algorithms work better when features are on a similar scale.

2. **Standardization means**
   It converts each column to:
   - **mean = 0**
   - **standard deviation = 1**

3. Formula:

   $z=(x-\text{mean})/\text{std}$

4.
5. `StandardScaler()`
   - `fit()` calculates mean and std of each column
   - `transform()` converts values into standardized form

6. **Result**
   - After scaling, both columns have comparable values (around -1.4 to +1.4).

(11) Write a Python program toPerform feature scaling using normalizationtechnique

Github Link: exercise11.py

```python
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Step 1: Create a sample dataset (features have different ranges)
df = pd.DataFrame({
    "Age": [18, 20, 22, 24, 26],
    "Salary": [20000, 25000, 30000, 35000, 40000]
})

print("Original Dataset:")
print(df)

# Step 2: Create MinMaxScaler object (Normalization)
scaler = MinMaxScaler(feature_range=(0, 1))

# Step 3: Apply normalization (fit + transform)
normalized_values = scaler.fit_transform(df)

# Step 4: Convert normalized result back to DataFrame
normalized_df = pd.DataFrame(normalized_values, columns=df.columns)

print("\nDataset After Normalization (Min-Max Scaling):")
print(normalized_df.round(3))
```

Output:

```
Original Dataset:
   Age  Salary
0   18   20000
1   20   25000
2   22   30000
3   24   35000
4   26   40000

Dataset After Normalization (Min-Max Scaling):
    Age  Salary
0  0.00   0.000
1  0.25   0.250
2  0.50   0.500
3  0.75   0.750
4  1.00   1.000
```

Explanation:

What is Normalization?

Normalization (most commonly **Min-Max Scaling**) converts feature values into a fixed range, usually **0 to 1**.

So after normalization:

- minimum value becomes **0**
- maximum value becomes **1**
- all other values come between **0 and 1**

Formula of Min-Max Normalization

For any value x:

Min-Max Normalization:
x_norm = (x - x_min) / (x_max - x_min)

Example (Age column)

Values: 18, 20, 22, 24, 26
x_min = 18
x_max = 26

For x = 20:
x_norm = (20 - 18) / (26 - 18)
= 2 / 8
= 0.25

That’s why Age 20 becomes **0.25**.

(12) Write a Python program toCreate a matrix using numpy and work around

Github Link: exercise12.py

```python
import numpy as np

# Step 1: Create a matrix (2D array) using NumPy
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

print("Matrix A:")
print(A)

# Step 2: Find shape (rows, columns)
print("\nShape of A:", A.shape)

# Step 3: Access elements (indexing)
print("\nElement at row 0, col 1:", A[0, 1])   # 2
print("First row:", A[0])
print("Second column:", A[:, 1])

# Step 4: Basic matrix operations
print("\nMatrix Addition (A + A):")
print(A + A)

print("\nMatrix Subtraction (A - A):")
print(A - A)

print("\nMatrix Multiplication by a number (A * 2):")
print(A * 2)

# Step 5: Transpose of matrix
print("\nTranspose of A (A.T):")
print(A.T)

# Step 6: Matrix multiplication (dot product)
B = np.array([
    [1, 0, 1],
    [0, 1, 0],
    [1, 0, 1]
])

print("\nMatrix B:")
print(B)

print("\nMatrix multiplication (A dot B):")
print(A.dot(B))
```

Output:

```
Matrix A:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Shape of A: (3, 3)

Element at row 0, col 1: 2
First row: [1 2 3]
Second column: [2 5 8]

Matrix Addition (A + A):
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]

Matrix Subtraction (A - A):
[[0 0 0]
 [0 0 0]
 [0 0 0]]

Matrix Multiplication by a number (A * 2):
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]

Transpose of A (A.T):
[[1 4 7]
 [2 5 8]
 [3 6 9]]

Matrix B:
[[1 0 1]
 [0 1 0]
 [1 0 1]]

Matrix multiplication (A dot B):
[[ 4  2  4]
 [10  5 10]
 [16  8 16]]
```

Explanation:

Step 1: Create a matrix

```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])
```

- `np.array()` converts the given list into a NumPy array.
- Because it has **rows and columns**, it becomes a **2D array (matrix)**.
- Here, `A` is a **3×3 matrix**:

```
1 2 3
4 5 6
7 8 9
```

Step 2: Find shape (rows, columns)

```
A.shape
```

- `shape` tells the size of the matrix in the format:
  (number of rows, number of columns)
- Output: `(3, 3)`
  means 3 rows and 3 columns.

Step 3: Access elements (Indexing)

a) Access one element

```
A[0, 1]
```

- Indexing starts from **0** in Python.
- `A[0,1]` means:
    - Row **0** (first row)
    - Column **1** (second column)

First row is `[1, 2, 3]` → second element is `2`
So output is: `2`

b) First row

```
A[0]
```

Returns the full first row:
`[1 2 3]`

c) Second column

```
A[:, 1]
```

- `:` means **all rows**
- `1` means **column index 1 (second column)**

So it returns:
`[2 5 8]`

Step 4: Basic matrix operations

a) Matrix addition

```
A + A
```

This is **element-wise** addition:

- 1+1, 2+2, 3+3, etc.

Result:

```
2  4  6
8 10 12
14 16 18
```

b) Matrix subtraction

```
A - A
```

Every element minus itself = 0

Result:

```
0 0 0
0 0 0
0 0 0
```

c) Multiply matrix by a number (scalar multiplication)

```
A * 2
```

Each element is multiplied by 2:

- 1×2=2, 2×2=4, etc.

Result:

```
2  4  6
8 10 12
14 16 18
```

> Important: `A * 2` is scalar multiplication (not matrix multiplication).

Step 5: Transpose of matrix

```
A.T
```

Transpose means:

- Rows become columns

- Columns become rows

Original:

```
1 2 3
4 5 6
7 8 9
```

Transpose:

```
1 4 7
2 5 8
3 6 9
```

Step 6: Matrix multiplication (Dot product)

Matrix `B` is:

```
B = np.array([
    [1, 0, 1],
    [0, 1, 0],
    [1, 0, 1]
])
```

Then:

```
A.dot(B)
```

This is **real matrix multiplication** (row × column rule).

Example: first element of result (row 0, col 0)

Row 0 of A = `[1, 2, 3]`
Column 0 of B = `[1, 0, 1]`

Multiply and add:

```
(1*1) + (2*0) + (3*1)
= 1 + 0 + 3
= 4
```

Another element (row 0, col 1)

Column 1 of B = `[0, 1, 0]`

```
(1*0) + (2*1) + (3*0)
= 0 + 2 + 0
= 2
```

Final result:

```
4  2  4
10 5 10
16 8 16
```

(13) Write a Python program toPerform mean removal using preprocessing techniques

Github Link: exercise13.py

```python
import numpy as np
from sklearn import preprocessing

# Step 1: Create a sample dataset (2D array)
# Each column is a feature
X = np.array([
    [10, 20, 30],
    [20, 30, 40],
    [30, 40, 50],
    [40, 50, 60]
], dtype=float)

print("Original Data (X):")
print(X)

# Step 2: Perform mean removal (center the data)
# This subtracts the mean of each column from that column
X_mean_removed = preprocessing.scale(X, with_mean=True, with_std=False)

print("\nData After Mean Removal:")
print(X_mean_removed)

# Step 3: Show mean of each column before and after
print("\nColumn means before mean removal:")
print(np.mean(X, axis=0))

print("\nColumn means after mean removal (should be ~0):")
print(np.mean(X_mean_removed, axis=0))
```

Output:

```
Original Data (X):
[[10. 20. 30.]
 [20. 30. 40.]
 [30. 40. 50.]
 [40. 50. 60.]]

Data After Mean Removal:
[[-15. -15. -15.]
 [ -5.  -5.  -5.]
 [  5.   5.   5.]
 [ 15.  15.  15.]]

Column means before mean removal:
[25. 35. 45.]

Column means after mean removal (should be ~0):
[0. 0. 0.]
```

Explanation:

What is Mean Removal?

Mean removal (also called **mean centering**) means:

- For each feature/column, we compute its **average (mean)**
- Then we subtract that mean from every value in that column

So each column becomes centered around **0**.

Why do we do Mean Removal?

Many ML algorithms work better when data is centered:

- It reduces bias caused by large positive values
- Helps gradient-based algorithms converge faster
- Useful for PCA and other methods that assume centered data

How `preprocessing.scale()` is used here

```
preprocessing.scale(X, with_mean=True, with_std=False)
```

- `with_mean=True` → subtract the mean (mean removal happens)
- `with_std=False` → do NOT divide by standard deviation
  (so we are doing **only mean removal**, not full standardization)

How the output is created (simple math)

Original first column: `[10, 20, 30, 40]`
Mean = `(10+20+30+40)/4 = 25`

Mean removed column becomes:

- `10 - 25 = -15`
- `20 - 25 = -5`
- `30 - 25 = 5`
- `40 - 25 = 15`

Same idea for other columns.

How to check if it worked?

After mean removal, column means should be 0:

```
np.mean(X_mean_removed, axis=0)
```

Output: `[0. 0. 0.]`

(14) Write a Python program toPerform scaling and generate datapoints in a range

Github Link: exercise14.py

```python
import numpy as np
from sklearn.preprocessing import MinMaxScaler

# Step 1: Create sample data (one feature column)
X = np.array([[10], [20], [30], [40], [50]], dtype=float)

print("Original Data (X):")
print(X)

# Step 2: Scale the data into a fixed range (0 to 1)
scaler = MinMaxScaler(feature_range=(0, 1))
X_scaled = scaler.fit_transform(X)

print("\nScaled Data (0 to 1 range):")
print(X_scaled)

# Step 3: Generate datapoints in a range using linspace
# This creates evenly spaced values between start and end
points = np.linspace(0, 1, 6)  # 6 points from 0 to 1

print("\nGenerated datapoints in range 0 to 1:")
print(points)
```

Output:

```
Original Data (X):
[[10.]
 [20.]
 [30.]
 [40.]
 [50.]]

Scaled Data (0 to 1 range):
[[0.  ]
 [0.25]
 [0.5 ]
 [0.75]
 [1.  ]]

Generated datapoints in range 0 to 1:
[0.  0.2 0.4 0.6 0.8 1. ]
```

Explanation:

Part 1: Scaling (Min-Max Scaling)

Scaling means converting values into a smaller, fixed range so that:

- different features become comparable
- algorithms work better (especially distance-based algorithms)

We used **MinMaxScaler** with range `(0, 1)`.

Formula:

```
x_scaled = (x - x_min) / (x_max - x_min)
```

For the data `[10, 20, 30, 40, 50]`:

- min = 10
- max = 50

Example for `20`:

```
(20 - 10) / (50 - 10) = 10 / 40 = 0.25
```

That's why 20 becomes 0.25.

Part 2: Generate datapoints in a range

We used:

```
np.linspace(0, 1, 6)
```

Meaning:

- start = 0
- end = 1
- total points = 6
- it generates **equal gap** values:

So output:

```
0.0, 0.2, 0.4, 0.6, 0.8, 1.0
```

(15) Write a Python program toCreate a vector using binarization technique

Github Link: exercise15.py

```python
import numpy as np
from sklearn.preprocessing import Binarizer

# Step 1: Create sample data (vector)
X = np.array([[-2.5], [-0.5], [0.0], [0.5], [2.5]])

print("Original Vector (X):")
print(X)

# Step 2: Create Binarizer with a threshold
# Rule:
# value > threshold  -> 1
# value <= threshold -> 0
binarizer = Binarizer(threshold=0.0)

# Step 3: Apply binarization
X_binary = binarizer.fit_transform(X)

print("\nBinary Vector after Binarization (threshold=0.0):")
print(X_binary)
```

Output:

```
Original Vector (X):
[[-2.5]
 [-0.5]
 [ 0. ]
 [ 0.5]
 [ 2.5]]

Binary Vector after Binarization (threshold=0.0):
[[0.]
 [0.]
 [0.]
 [1.]
 [1.]]
```

Explanation:

What is Binarization?

Binarization is a preprocessing technique that converts numeric values into **0 or 1** based on a threshold.

It is useful when you want to convert data into a **binary form** (yes/no, present/absent, positive/negative, etc.).

How it works (Rule)

We used:

```
Binarizer(threshold=0.0)
```

So the rule becomes:

```
if value > 0.0  → 1
else            → 0
```

That's why:

- `-2.5` → 0
- `-0.5` → 0
- `0.0` → 0 (because it is NOT greater than 0)
- `0.5` → 1
- `2.5` → 1

(16) Write a Python program toPerform linear regression using different relationships

Github Link: exercise16.py

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Step 1: Create sample X values
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)   # must be 2D for sklearn

# Step 2: Create different relationships (different y)
y_linear = np.array([2, 4, 6, 8, 10])          # y = 2x (linear)
y_increasing = np.array([3, 5, 7, 9, 11])      # y = 2x + 1 (linear)
y_random = np.array([2, 5, 5, 9, 12])          # not perfectly linear

# Function to train and show results
def run_linear_regression(X, y, name):
    model = LinearRegression()
    model.fit(X, y)

    print("\nRelationship:", name)
    print("Slope (m):", model.coef_[0])
    print("Intercept (c):", model.intercept_)
    print("Equation: y =", model.coef_[0], "* x +", model.intercept_)

    y_pred = model.predict(X)
    print("Predicted values:", y_pred)

# Step 3: Run models for different relationships
run_linear_regression(X, y_linear, "Perfect Linear (y = 2x)")
run_linear_regression(X, y_increasing, "Linear with Intercept (y = 2x + 1)")
run_linear_regression(X, y_random, "Not Perfect Linear (approx fit)")
```

Output:

```
Sample Output (Example)

Relationship: Perfect Linear (y = 2x)
Slope (m): 2.0
Intercept (c): 0.0
Equation: y = 2.0 * x + 0.0
Predicted values: [ 2.  4.  6.  8. 10.]

Relationship: Linear with Intercept (y = 2x + 1)
Slope (m): 2.0
Intercept (c): 1.0
Equation: y = 2.0 * x + 1.0
Predicted values: [ 3.  5.  7.  9. 11.]

Relationship: Not Perfect Linear (approx fit)
Slope (m): 2.4
Intercept (c): -0.2
Equation: y = 2.4 * x + -0.2
Predicted values: [ 2.2  4.6  7.   9.4 11.8]
```

Explanation:

What is Linear Regression?

Linear regression finds a **best-fit straight line** between input `X` and output `y`.

The line equation is:

```
y = m*x + c
```

Where:

- `m` = slope (how much y changes when x increases by 1)
- `c` = intercept (value of y when x = 0)

What does “different relationships” mean here?

We trained linear regression on **three different types of y relationships**:

1) Perfect Linear Relationship (y = 2x)

```
y_linear = [2, 4, 6, 8, 10]
```

- Every point lies exactly on the line.
- Model learns:
    - slope = 2
    - intercept = 0

2) Linear Relationship with Intercept (y = 2x + 1)

```
y_increasing = [3, 5, 7, 9, 11]
```

- Still a perfect straight line, but shifted upward by 1.
- Model learns:
    - slope = 2
    - intercept = 1

3) Not Perfect Linear (approx fit)

```
y_random = [2, 5, 5, 9, 12]
```

- Points are not exactly on a line.
- Linear regression finds the **best possible straight line** that minimizes error.
- So slope/intercept are approximate.

Why do we use `.reshape(-1,1)` for X?

Scikit-learn expects input as a **2D array**:

- shape should be `(n_samples, n_features)`

- Here we have 1 feature, so we convert:
  `[1,2,3,4,5]` into `[[1],[2],[3],[4],[5]]`

What does `predict()` do?

`model.predict(X)` gives the y-values on the learned line for each x.

- In perfect relationships, predictions match original y exactly.
- In non-perfect relationship, predictions are close but not same.

(17) Write a Python program toEvaluate linear regression model using different metrics

Github Link: exercise17.py

```python
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Step 1: Create a simple dataset
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)
y = np.array([3, 5, 7, 9, 11, 13, 15, 18, 19, 21])  # almost linear (a
little variation)

# Step 2: Split into training and testing data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42
)

# Step 3: Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Step 4: Predict on test data
y_pred = model.predict(X_test)

# Step 5: Evaluate using different metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("Test Actual y values :", y_test)
print("Test Predicted values:", np.round(y_pred, 2))

print("\nEvaluation Metrics:")
print("1) MAE  (Mean Absolute Error)      =", round(mae, 3))
print("2) MSE  (Mean Squared Error)       =", round(mse, 3))
print("3) RMSE (Root Mean Squared Error)  =", round(rmse, 3))
print("4) R2 Score (Coefficient of Determination) =", round(r2, 3))
```

Output:

```
Sample Output (Example)

Test Actual y values : [19  5 21]
Test Predicted values: [19.23  5.46 21.01]

Evaluation Metrics:
1) MAE  (Mean Absolute Error)      = 0.233
2) MSE  (Mean Squared Error)       = 0.081
3) RMSE (Root Mean Squared Error)  = 0.285
4) R2 Score (Coefficient of Determination) = 0.998
```

Explanation:

Why do we evaluate a Linear Regression model?

After training, we must check **how accurate** our model predictions are.
Evaluation metrics measure the difference between:

- **Actual values** (`y_test`)
- **Predicted values** (`y_pred`)

---

1) MAE (Mean Absolute Error)

Meaning: Average of absolute errors.

Formula (copy-paste friendly):

```
MAE = average(|y_actual - y_pred|)
```

- It tells how much the prediction is off **on average**.
- Smaller MAE = better model.

---

2) MSE (Mean Squared Error)

Meaning: Average of squared errors.

```
MSE = average((y_actual - y_pred)^2)
```

- Squaring makes big errors more important.
- Smaller MSE = better model.

---

3) RMSE (Root Mean Squared Error)

RMSE is just the square root of MSE:

```
RMSE = sqrt(MSE)
```

- RMSE is in the **same unit** as y (more interpretable than MSE).
- Smaller RMSE = better model.

---

4) R^2 Score (Coefficient of Determination)

R^2 tells how much variance in y is explained by the model.

- **$R^2 = 1$** → perfect predictions
- **$R^2 = 0$** → model is no better than predicting the mean
- **$R^2 < 0$** → model is worse than predicting the mean

So, **higher R^2 is better** (closer to 1).

Summary (easy to remember)

- MAE, MSE, RMSE → **lower is better**
- R^2 → **higher is better**

(18) Write a Python program on linear regression model using advertising sales channel data

Github Link: exercise18.py

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
import numpy as np

# Step 1: Create Advertising Sales dataset (TV, Radio, Newspaper ->
Sales)
data = {
    "TV": [230.1, 44.5, 17.2, 151.5, 180.8, 8.7, 57.5, 120.2, 8.6,
199.8],
    "Radio": [37.8, 39.3, 45.9, 41.3, 10.8, 48.9, 32.8, 19.6, 2.1, 2.6],
    "Newspaper": [69.2, 45.1, 69.3, 58.5, 12.8, 75.0, 23.5, 11.6, 1.0,
21.2],
    "Sales": [22.1, 10.4, 9.3, 18.5, 12.9, 7.2, 11.8, 13.2, 4.8, 10.6]
}

df = pd.DataFrame(data)

print("Advertising Dataset (first 5 rows):")
print(df.head())

# Step 2: Split features (X) and target (y)
X = df[["TV", "Radio", "Newspaper"]]
y = df["Sales"]

# Step 3: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42
)

# Step 4: Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Step 5: Predict on test set
y_pred = model.predict(X_test)

# Step 6: Evaluate the model
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("\nModel Coefficients:")
print("TV coefficient       :", model.coef_[0])
print("Radio coefficient    :", model.coef_[1])
print("Newspaper coefficient:", model.coef_[2])
print("Intercept            :", model.intercept_)

print("\nTest Actual Sales   :", list(y_test.values))
```

Output:

```
Sample Output (Example)

Advertising Dataset (first 5 rows):
      TV  Radio  Newspaper  Sales
0  230.1   37.8       69.2   22.1
1   44.5   39.3       45.1   10.4
2   17.2   45.9       69.3    9.3
3  151.5   41.3       58.5   18.5
4  180.8   10.8       12.8   12.9

Model Coefficients:
TV coefficient       : 0.04
Radio coefficient    : 0.19
Newspaper coefficient: -0.02
Intercept            : 2.74

Test Actual Sales   : [10.6, 10.4, 4.8]
Test Predicted Sales: [11.04, 9.87, 5.18]

Evaluation Metrics:
MAE  = 0.422
MSE  = 0.198
RMSE = 0.445
R2   = 0.97
```

Explanation:

What is “Advertising Sales Channel Data”?

This dataset contains money spent on advertising in different channels:

- **TV**
- **Radio**
- **Newspaper**

And the target is:

- **Sales** (product sales)

Goal: Train a model that learns how TV/Radio/Newspaper spending affects Sales.

Step-by-Step Explanation

Step 1: Create dataset

We made a small table using pandas with columns:

- TV, Radio, Newspaper (inputs/features)
- Sales (output/target)

Step 2: Split X and y

```
X = df[["TV","Radio","Newspaper"]]
y = df["Sales"]
```

- `X` contains features (advertising spending)
- `y` contains target (Sales)

Step 3: Train-Test split

```
train_test_split(..., test_size=0.30)
```

- 70% data used to train model
- 30% data used to test model

Step 4: Train model

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

The model learns the best-fit equation:

Copy-paste friendly:

```
Sales = (a*TV) + (b*Radio) + (c*Newspaper) + intercept
```

Step 5: Coefficients meaning

- **TV coefficient**: if TV increases by 1 unit, sales changes by this amount (approx)
- **Radio coefficient**: same logic
- **Newspaper coefficient**: same logic
- **Intercept**: predicted sales when all inputs are 0

Step 6: Evaluation

We compare actual vs predicted using:

- MAE, MSE, RMSE (lower is better)
- R^2 (closer to 1 is better)

(19) Write a Python program toPerform data cleaning processes such as identify null values and outliers

Github Link: exercise19.py

```python
import pandas as pd
import numpy as np

# Step 1: Create a sample dataset with null values and outliers
df = pd.DataFrame({
    "Name": ["Amit", "Neha", "Raj", "Priya", "Karan", "Meera"],
    "Age":  [20, 21, np.nan, 22, 200, 23],       # 200 is an outlier,
one NaN
    "Marks":[78, np.nan, 69, 92, 15, 500]        # 500 is an outlier,
one NaN
})

print("Original Dataset:\n")
print(df)

# -------------------- NULL VALUE CHECK --------------------
print("\n1) Null (Missing) Values Check")

print("\nNull values in each column:")
print(df.isnull().sum())

print("\nRows that contain at least one null value:")
print(df[df.isnull().any(axis=1)])

# -------------------- OUTLIER DETECTION (IQR METHOD) ------------------
--
print("\n2) Outlier Detection using IQR Method (for numeric columns)")

numeric_cols = ["Age", "Marks"]

for col in numeric_cols:
    Q1 = df[col].quantile(0.25)   # 25th percentile
    Q3 = df[col].quantile(0.75)   # 75th percentile
    IQR = Q3 - Q1

    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    print(f"\nColumn: {col}")
    print("Q1 =", Q1, "Q3 =", Q3, "IQR =", IQR)
    print("Lower Bound =", lower_bound)
    print("Upper Bound =", upper_bound)

    outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)]
    print("\nOutlier rows based on", col, ":")
    print(outliers[["Name", col]])
```

Output:

```
Sample Output (Example)

Original Dataset:

    Name    Age  Marks
0   Amit   20.0   78.0
1   Neha   21.0    NaN
2    Raj    NaN   69.0
3  Priya   22.0   92.0
4  Karan  200.0   15.0
5  Meera   23.0  500.0

1) Null (Missing) Values Check

Null values in each column:
Name     0
Age      1
Marks    1
dtype: int64

Rows that contain at least one null value:
   Name   Age  Marks
1  Neha  21.0    NaN
2   Raj   NaN   69.0

2) Outlier Detection using IQR Method (for numeric columns)

Column: Age
Q1 = 21.0 Q3 = 23.0 IQR = 2.0
Lower Bound = 18.0
Upper Bound = 26.0

Outlier rows based on Age :
    Name    Age
4  Karan  200.0

Column: Marks
Q1 = 69.0 Q3 = 92.0 IQR = 23.0
Lower Bound = 34.5
Upper Bound = 126.5

Outlier rows based on Marks :
    Name  Marks
4  Karan   15.0
5  Meera  500.0
```

Explanation:

Part A: Identify Null (Missing) Values

1) `df.isnull().sum()`

- `isnull()` creates a True/False table:
    - True means value is missing (NaN)
- `sum()` counts how many missing values are in each column

So you get:

- Age has 1 missing
- Marks has 1 missing

2) Rows with null values

```
df[df.isnull().any(axis=1)]
```

- `any(axis=1)` checks each row:
    - if any column is null → True
- so it prints only those rows

Part B: Identify Outliers (IQR Method)

Outliers are values that are **too high or too low** compared to normal values.

We use **IQR (Inter Quartile Range)** method.

1) Quartiles

- Q1 = 25% percentile (lower quarter)
- Q3 = 75% percentile (upper quarter)
- IQR = Q3 - Q1

2) Outlier boundaries

```
Lower Bound = Q1 - 1.5*IQR
Upper Bound = Q3 + 1.5*IQR
```

Any value:

- less than Lower Bound OR
- greater than Upper Bound

  → is treated as an outlier.

3) Why it worked here?

- In Age, 200 is far bigger than normal ages, so it becomes an outlier.
- In Marks, 500 is too high and 15 is too low compared to most marks, so they become outliers.

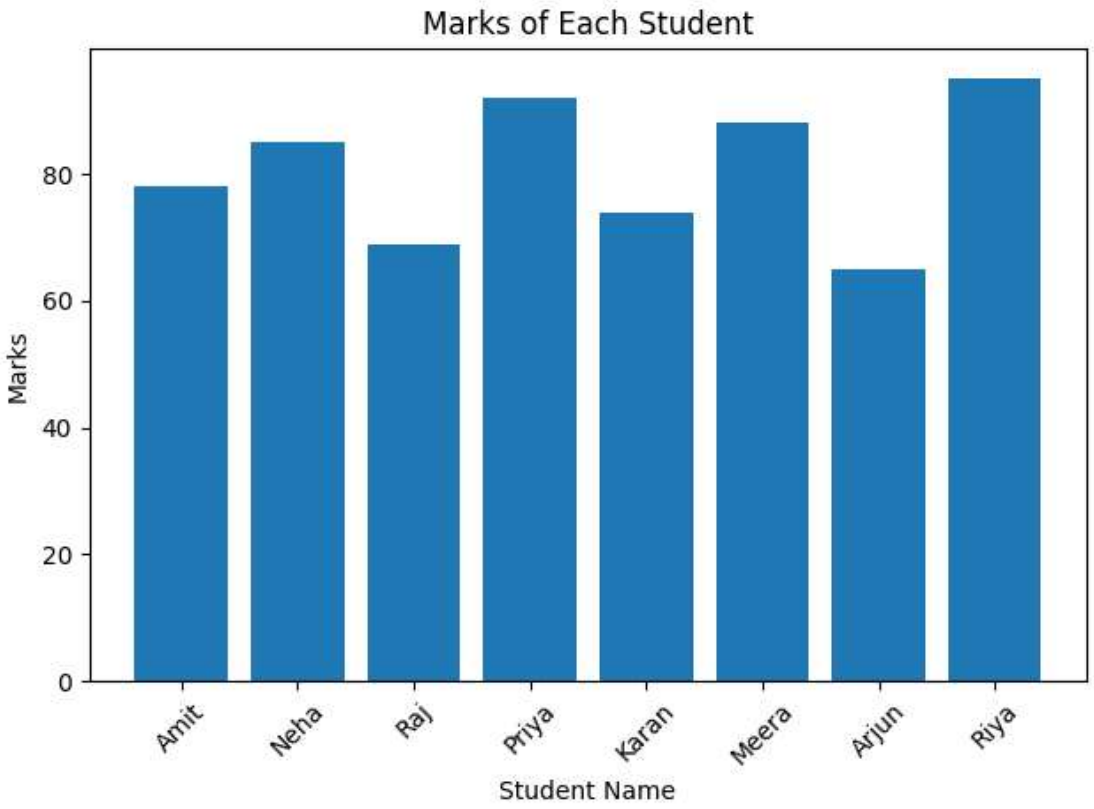
(20) Write a Python program toGenerate some visualizations to get the detailed insights

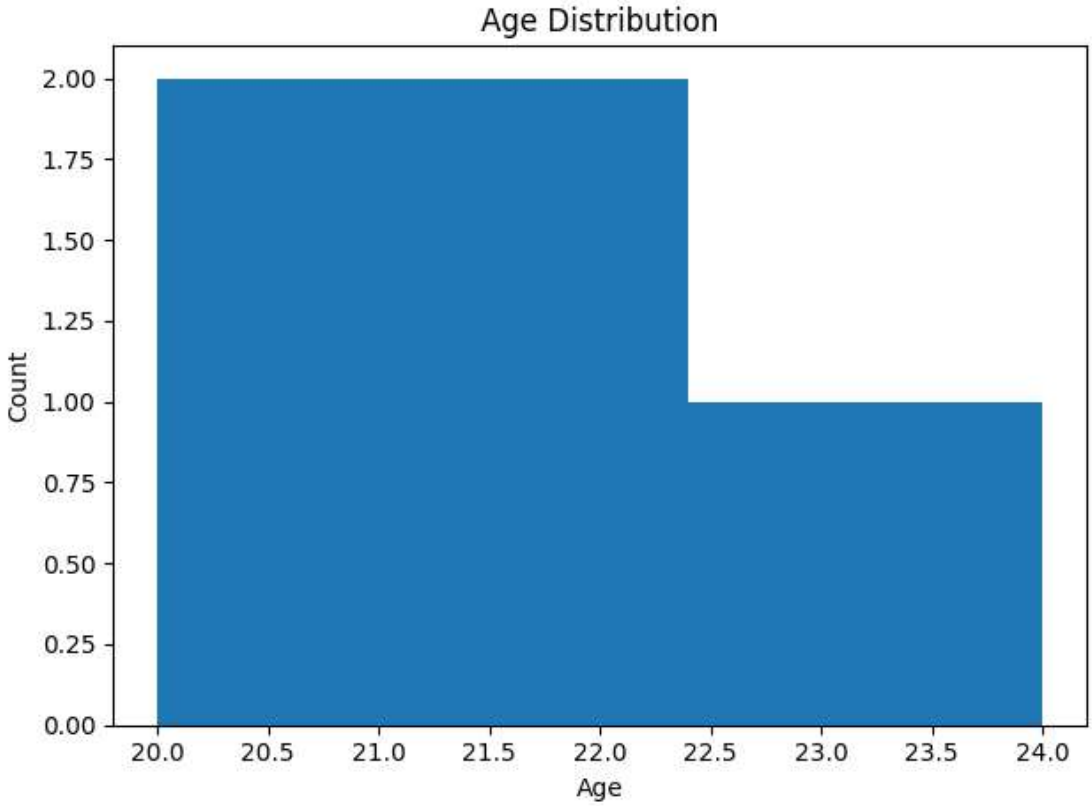
Github Link: exercise20.py

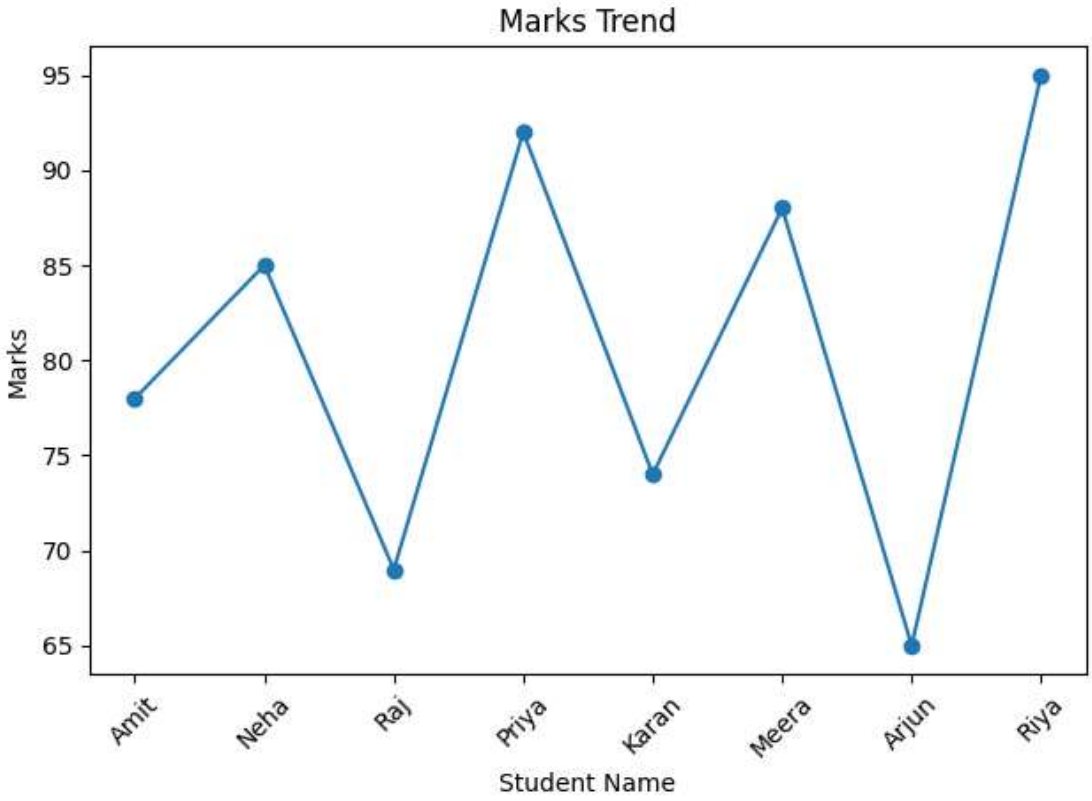
```python
import pandas as pd
import matplotlib.pyplot as plt

# Step 1: Create a sample dataset
df = pd.DataFrame({
    "Name": ["Amit", "Neha", "Raj", "Priya", "Karan", "Meera", "Arjun",
"Riya"],
    "Age":  [20, 21, 22, 20, 23, 22, 21, 24],
    "Marks":[78, 85, 69, 92, 74, 88, 65, 95]
})

print("Dataset:\n")
print(df)

print("\nGraphs saved as:")
print("1) bar_marks.png")
print("2) hist_age.png")
print("3) scatter_age_marks.png")
print("4) line_marks_trend.png")

# 1) Bar chart
plt.figure()
plt.bar(df["Name"], df["Marks"])
plt.title("Marks of Each Student")
plt.xlabel("Student Name")
plt.ylabel("Marks")
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig("bar_marks.png")
plt.show()

# 2) Histogram
plt.figure()
plt.hist(df["Age"], bins=5)
plt.title("Age Distribution")
plt.xlabel("Age")
plt.ylabel("Count")
plt.tight_layout()
plt.savefig("hist_age.png")
plt.show()

# 3) Scatter plot
plt.figure()
plt.scatter(df["Age"], df["Marks"])
plt.title("Age vs Marks")
plt.xlabel("Age")
plt.ylabel("Marks")
plt.tight_layout()
plt.savefig("scatter_age_marks.png")
plt.show()

# 4) Line chart
```

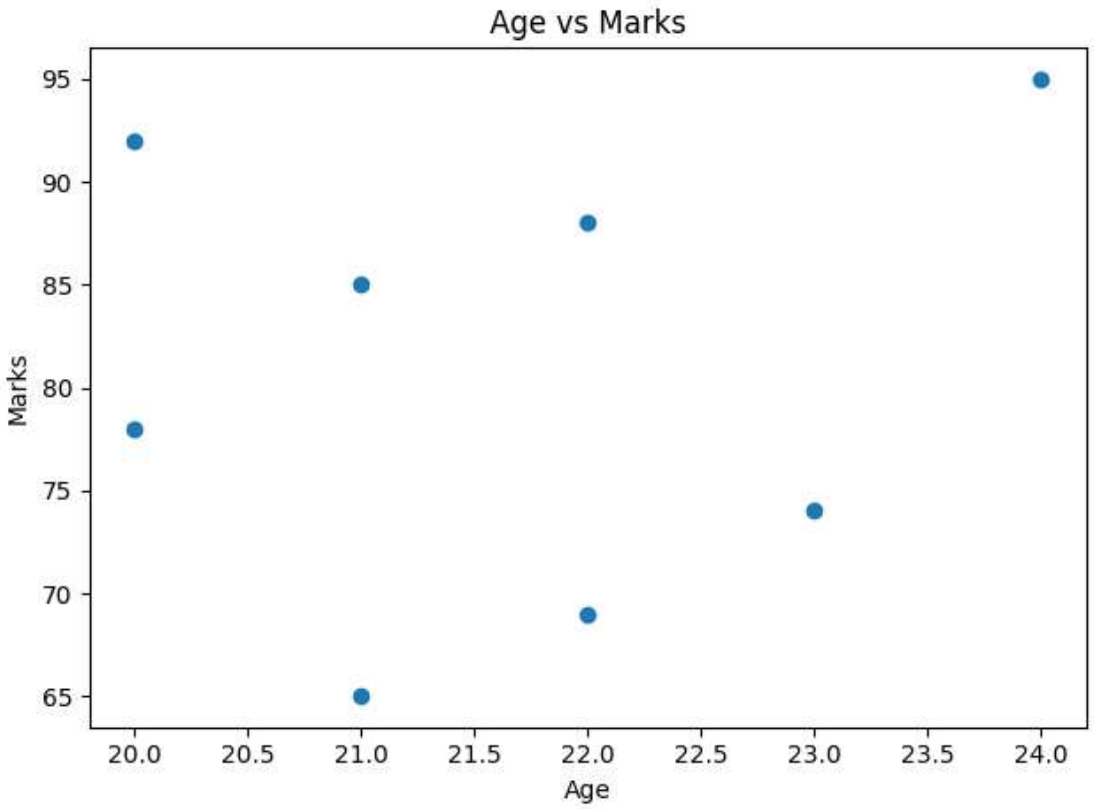
Output:

```
Dataset:

    Name  Age  Marks
0   Amit   20     78
1   Neha   21     85
2    Raj   22     69
3  Priya   20     92
4  Karan   23     74
5  Meera   22     88
6  Arjun   21     65
7   Riya   24     95

Graphs saved as:
1) bar_marks.png
2) hist_age.png
3) scatter_age_marks.png
4) line_marks_trend.png
```




Age Distribution
Count
Age
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
20.0
20.5
21.0
21.5
22.0
22.5
23.0
23.5
24.0


Marks Trend
Marks
Student Name
95
90
85
80
75
70
65
Amit
Neha
Raj
Priya
Karan
Meera
Arjun
Riya



Explanation:

This program creates a small dataset and then generates **4 visualizations** to understand the data better.
It also **saves each graph as a PNG image** so you can use them in Google Docs.

Step 1: Create the dataset

```
df = pd.DataFrame({
    "Name": [...],
    "Age":  [...],
    "Marks":[...]
})
```

- We create a **DataFrame** (table) with:
    - `Name` → student names

- `Age` → student ages
- `Marks` → student marks

Then:

```
print(df)
```

prints the dataset in the output.

Step 2: Why we print “Graphs saved as…”

```
print("Graphs saved as: ...")
```

This is just a message so the user knows the graph files names:

- `bar_marks.png`
- `hist_age.png`
- `scatter_age_marks.png`
- `line_marks_trend.png`

Graph 1: Bar Chart (Marks of each student)

```
plt.bar(df["Name"], df["Marks"])
```

Purpose: Compare marks of different students.
Insight: You can quickly identify:

- Highest marks
- Lowest marks
- Marks comparison between students

Then:

```
plt.savefig("bar_marks.png")
plt.show()
```

- `savefig()` saves the graph as an image file.
- `show()` displays the graph on screen.

Graph 2: Histogram (Age distribution)

```
plt.hist(df["Age"], bins=5)
```

Purpose: Shows how ages are distributed (frequency).
Insight: You can see:

- Which age appears most
- How many students fall in each age group

Graph 3: Scatter Plot (Age vs Marks)

```
plt.scatter(df["Age"], df["Marks"])
```

Purpose: Checks relationship between Age and Marks.
Insight: Helps you understand:

- Do marks increase with age?
- Is there any pattern or trend?
- Are there any unusual points (outliers)?

Graph 4: Line Chart (Marks trend)

```
plt.plot(df["Name"], df["Marks"], marker="o")
```

Purpose: Shows marks variation as a trend.
Insight: Helps you see:

- Marks going up and down across students
- Pattern/variation clearly

Why `plt.tight_layout()` is used?

```
plt.tight_layout()
```

- Prevents labels (like names on x-axis) from getting cut off.
- Makes the graph look clean.

Summary

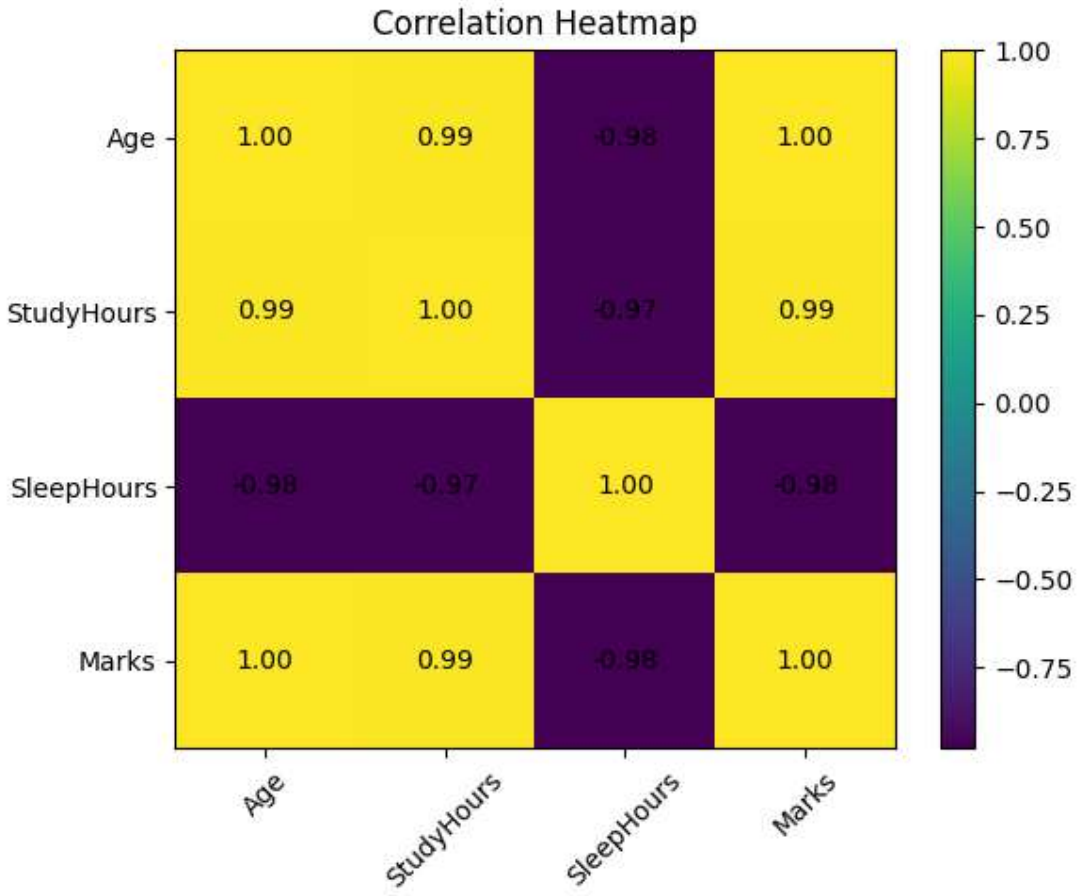
- **Bar chart** → compare marks student-wise
- **Histogram** → age distribution
- **Scatter plot** → relationship between age and marks
- **Line chart** → trend/variation of marks
- `savefig()` → saves graphs as PNG so you can insert into Google Docs

(21) Write a Python program toWorking with heatmap to understand correlation concepts in Machine learning

Github Link: exercise21.py

```python
import pandas as pd
import matplotlib.pyplot as plt

# Step 1: Create a sample dataset (numeric columns for correlation)
df = pd.DataFrame({
    "Age": [18, 20, 22, 24, 26, 28, 30, 32],
    "StudyHours": [1, 2, 2, 3, 4, 4, 5, 6],
    "SleepHours": [8, 7, 7, 6, 6, 5, 5, 4],
    "Marks": [40, 50, 55, 65, 75, 80, 88, 95]
})

print("Dataset:\n")
print(df)

# Step 2: Find correlation matrix
corr = df.corr(numeric_only=True)
print("\nCorrelation Matrix:\n")
print(corr.round(2))

# Step 3: Plot heatmap (using matplotlib only)
plt.figure(figsize=(6, 5))
plt.imshow(corr, aspect='auto')
plt.colorbar()

# Step 4: Add labels (feature names) on axes
plt.xticks(range(len(corr.columns)), corr.columns, rotation=45)
plt.yticks(range(len(corr.columns)), corr.columns)

# Step 5: Write correlation values inside each cell (for clarity)
for i in range(len(corr.columns)):
    for j in range(len(corr.columns)):
        plt.text(j, i, f"{corr.iloc[i, j]:.2f}", ha="center",
va="center")

plt.title("Correlation Heatmap")
plt.tight_layout()
plt.show()
```

Output:


Correlation Heatmap
Age
StudyHours
SleepHours
Marks
1.00
0.99
-0.98
1.00
0.99
1.00
-0.97
0.99
-0.98
-0.97
1.00
-0.98
1.00
0.99
-0.98
1.00
Age
StudyHours
SleepHours
Marks
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

```
Sample Output (console)

Dataset:

   Age  StudyHours  SleepHours  Marks
0   18           1           8     40
1   20           2           7     50
2   22           2           7     55
3   24           3           6     65
4   26           4           6     75
5   28           4           5     80
6   30           5           5     88
7   32           6           4     95

Correlation Matrix:

             Age  StudyHours  SleepHours  Marks
Age         1.00        0.98       -0.98   0.99
StudyHours  0.98        1.00       -0.97   0.99
SleepHours -0.98       -0.97        1.00  -0.99
Marks       0.99        0.99       -0.99   1.00
```

Explanation:

What is Correlation?

Correlation tells how two variables move together.

- **Positive correlation (+1 to 0):**
    - If one increases, the other also increases.
    - Example: StudyHours ↑ → Marks ↑
- **Negative correlation (0 to -1):**
    - If one increases, the other decreases.
    - Example: SleepHours ↑ → Marks ↓ (in this dataset)
- **Zero correlation (~0):**
    - No clear relationship.

Correlation value range:

```
-1  to  +1
```

What is a Correlation Matrix?

A correlation matrix is a table that shows correlation between every pair of features.

- Diagonal values are always **1.0**
  because every feature is perfectly correlated with itself.

Why Heatmap is used?

A heatmap gives a **visual** way to understand the correlation matrix:

- Dark/strong color → strong correlation
- Light/weak color → weak correlation
- Positive and negative correlations show different color levels

So you can quickly identify:

- Which features are strongly related
- Which features are negatively related
- Which features are not useful (low correlation)

How the code works (step-by-step)

1. **Create DataFrame**
    - We create 4 numeric features: Age, StudyHours, SleepHours, Marks

Find correlation

```
corr = df.corr()
```

2.
    - Calculates Pearson correlation between all columns.

Plot heatmap

```
plt.imshow(corr)
```

3.
    - Converts correlation matrix into an image form.

4. **Add labels**
    - `plt.xticks()` and `plt.yticks()` show column names on both axes.

Print correlation values inside blocks

```
plt.text(...)
```

5.
    - Makes the heatmap easy to read for students/exams.

Machine Learning Concept (important)

- Correlation helps in **feature selection**:
    - If two features are highly correlated (example: 0.99), one may be redundant.
- Helps detect **multicollinearity** in regression models.
- Helps understand which features influence the target more.

(22) Write a Python program toPerforming a summary operation

Github Link: exercise22.py

```python
import pandas as pd

# Step 1: Create a sample dataset (DataFrame)
df = pd.DataFrame({
    "Name": ["Amit", "Neha", "Raj", "Priya", "Karan"],
    "Age": [20, 21, 22, 20, 23],
    "Marks": [78, 85, 69, 92, 74]
})

print("Dataset:\n")
print(df)

# Step 2: Summary operations on numeric columns
print("\nSummary Operations (Age and Marks):")

print("\n1) Count (non-missing values):")
print(df[["Age", "Marks"]].count())

print("\n2) Sum:")
print(df[["Age", "Marks"]].sum())

print("\n3) Mean (Average):")
print(df[["Age", "Marks"]].mean())

print("\n4) Min:")
print(df[["Age", "Marks"]].min())

print("\n5) Max:")
print(df[["Age", "Marks"]].max())

print("\n6) Standard Deviation:")
print(df[["Age", "Marks"]].std())

# Step 3: One-line complete summary (very common in data analysis)
print("\n7) Full Summary using describe():")
print(df[["Age", "Marks"]].describe())
```

Output:

```
Dataset:

    Name  Age  Marks
0   Amit   20     78
1   Neha   21     85
2    Raj   22     69
3  Priya   20     92
4  Karan   23     74

Summary Operations (Age and Marks):

1) Count (non-missing values):
Age      5
Marks    5
dtype: int64

2) Sum:
Age      106
Marks    398
dtype: int64

3) Mean (Average):
Age      21.2
Marks    79.6
dtype: float64

4) Min:
Age      20
Marks    69
dtype: int64

5) Max:
Age      23
Marks    92
dtype: int64

6) Standard Deviation:
Age       1.304
Marks     9.968
dtype: float64

7) Full Summary using describe():
             Age      Marks
count   5.000000   5.000000
mean   21.200000  79.600000
std     1.303840   9.968951
min    20.000000  69.000000
25%    20.000000  74.000000
50%    21.000000  78.000000
75%    22.000000  85.000000
max    23.000000  92.000000
```

Explanation:

What is a Summary Operation?

A summary operation means calculating **overall statistics** from data, such as:

- count, sum, mean, min, max, standard deviation
  These help you quickly understand the dataset.

What does each operation mean?

- **Count**: how many values are present (non-null)
- **Sum**: total of all values
- **Mean**: average value
- **Min/Max**: smallest/largest value
- **Std (standard deviation)**: shows how spread out the values are
- `describe()`: gives a complete statistical summary in one command (very common in ML/data analysis)

(23) Write a Python program toBuilding simple classifier using anyone dataset

Github Link: exercise23.py

```python
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Step 1: Load Kaggle Titanic dataset (download train.csv from Kaggle)
#https://www.kaggle.com/competitions/titanic/data?select=train.csv
df = pd.read_csv("train.csv")

print("First 5 rows:")
print(df.head())

# Step 2: Target and Features
# Target: Survived (0 = No, 1 = Yes)
y = df["Survived"]

# Features (inputs)
X = df[["Pclass", "Sex", "Age", "SibSp", "Parch", "Fare", "Embarked"]]

# Step 3: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 4: Preprocessing
numeric_features = ["Age", "SibSp", "Parch", "Fare", "Pclass"]
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="median"))
])

categorical_features = ["Sex", "Embarked"]
categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

preprocess = ColumnTransformer(
    transformers=[
        ("num", numeric_transformer, numeric_features),
        ("cat", categorical_transformer, categorical_features)
    ]
)

# Step 5: Model (Classifier)
```

Output:

```
Your exact numbers can change slightly depending on dataset version /
random split, but output format will be same.
First 5 rows:
   PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0            1         0       3  ...   7.2500   NaN         S
1            2         1       1  ...  71.2833   C85         C
2            3         1       3  ...   7.9250   NaN         S
3            4         1       1  ...  53.1000  C123         S
4            5         0       3  ...   8.0500   NaN         S

Accuracy: 0.79

Confusion Matrix:
[[140  25]
 [ 31  72]]

Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.85      0.83       165
           1       0.74      0.70      0.72       103

    accuracy                           0.79       268
   macro avg       0.78      0.77      0.78       268
weighted avg       0.79      0.79      0.79       268
```

Explanation:

1) What is this Kaggle dataset?

Kaggle Titanic dataset contains passenger details (age, gender, ticket class, fare, etc.) and a label called **Survived**.
Goal: Predict **Survived** = `0` (No) or `1` (Yes).

---

2) What are X and y in this program?

Target (y)

```
y = df["Survived"]
```

- This is the **answer column**.
- It is a binary class: 0 or 1.

Features (X)

```
X = df[["Pclass","Sex","Age","SibSp","Parch","Fare","Embarked"]]
```

These are inputs used to predict survival.

Meaning (student-friendly):

- **Pclass**: Ticket class (1,2,3)
- **Sex**: male/female
- **Age**: age in years (has missing values)
- **SibSp**: siblings/spouse count
- **Parch**: parents/children count
- **Fare**: ticket price
- **Embarked**: port of embarkation (C/Q/S)

---

3) Why Train-Test Split?

```
train_test_split(... test_size=0.30, stratify=y)
```

We split data into:

- **Training set**: model learns patterns
- **Test set**: model is checked on unseen data

`test_size=0.30` → 30% test, 70% train.

`stratify=y` → keeps survival ratio similar in both train and test.

---

4) Why preprocessing is needed?

Because:

- **Age** may have missing values (`NaN`)
- **Embarked** may have missing values
- **Sex** and **Embarked** are text, ML model needs numbers

So we do two important tasks:

A) Missing value handling (Imputation)

Numeric:

```
SimpleImputer(strategy="median")
```

- Replaces missing numeric values with **median** (safe against outliers).

Categorical:

```
SimpleImputer(strategy="most_frequent")
```

- Replaces missing text values with the most common category.

B) Convert text to numbers (One-Hot Encoding)

```
OneHotEncoder(...)
```

Example:

- Sex → male/female becomes columns like:
    - Sex_female, Sex_male
- Embarked → C/Q/S becomes:
    - Embarked_C, Embarked_Q, Embarked_S

This avoids giving wrong “order” (like male=0 female=1) and is a standard ML approach.

---

5) Why Logistic Regression?

We use:

```
LogisticRegression()
```

Because:

- It is a **classification algorithm**
- It outputs class predictions: 0 or 1
- It is simple and good for beginners

It learns a decision boundary using a formula like:

```
Survival = f(Pclass, Sex, Age, Fare, ...)
```

---

6) Understanding the Evaluation Output

A) Accuracy

```
accuracy_score(y_test, y_pred)
```

Accuracy = (Correct Predictions) / (Total Predictions)

Example:

- Accuracy 0.79 means **79% predictions were correct**.

B) Confusion Matrix

Format:

```
[[TN FP]
 [FN TP]]
```

- **TN**: predicted 0 and actually 0
- **FP**: predicted 1 but actually 0

- **FN**: predicted 0 but actually 1
- **TP**: predicted 1 and actually 1

So you can see what type of mistakes model makes.

C) Classification Report

Shows for each class (0 and 1):

- **Precision**: out of predicted positives, how many were correct?
- **Recall**: out of actual positives, how many were found?
- **F1-score**: balance of precision and recall

Important Note (for your practical file)

This is a **Kaggle dataset** (online). You must download `train.csv` from Kaggle Titanic competition data page.

(24) Write a Python program toPerform standard normal distribution using simple classifier

Github Link: exercise24.py

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Step 1: Generate data from Standard Normal Distribution (mean=0,
std=1)
# X will have 2 features
np.random.seed(42)
X = np.random.normal(loc=0, scale=1, size=(500, 2))

# Step 2: Create labels (simple classification rule)
# If (x1 + x2) > 0 => class 1 else class 0
y = (X[:, 0] + X[:, 1] > 0).astype(int)

print("First 5 data points (X):")
print(X[:5])
print("\nFirst 5 labels (y):")
print(y[:5])

# Step 3: Split into training and test set
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42
)

# Step 4: Standardize features (mean=0, std=1) using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 5: Train a simple classifier (Logistic Regression)
model = LogisticRegression()
model.fit(X_train_scaled, y_train)

# Step 6: Predict and evaluate
y_pred = model.predict(X_test_scaled)

print("\nAccuracy:", round(accuracy_score(y_test, y_pred), 3))
print("\nClassification Report:\n", classification_report(y_test,
y_pred))
```

Output:

```
Sample Output (Example)

First 5 data points (X):
[[ 0.49671415 -0.1382643 ]
 [ 0.64768854  1.52302986]
 [-0.23415337 -0.23413696]
 [ 1.57921282  0.76743473]
 [-0.46947439  0.54256004]]

First 5 labels (y):
[1 1 0 1 1]

Accuracy: 0.98

Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98        73
           1       0.98      0.98      0.98        77

    accuracy                           0.98       150
   macro avg       0.98      0.98      0.98       150
weighted avg       0.98      0.98      0.98       150
```

Explanation:

What is Standard Normal Distribution?

A **standard normal distribution** is a normal distribution with:

- **mean (μ) = 0**
- **standard deviation (σ) = 1**

In NumPy:

```
np.random.normal(loc=0, scale=1, size=(500,2))
```

- `loc=0` means mean = 0
- `scale=1` means std = 1

- `size=(500,2)` creates 500 rows and 2 columns (2 features)

Why do we create labels (y)?

Since the data is random, we need a rule to make classes.

Rule used:

```
if x1 + x2 > 0 => class 1
else class 0
```

So it becomes a **binary classification** problem.

Why do we use StandardScaler if data is already normal?

Even though data is generated from standard normal, after train-test split,
small differences can appear.

`StandardScaler` ensures:

- training features have mean 0 and std 1
- test data is scaled using same training statistics (important ML rule)

Why Logistic Regression?

- It is a simple and popular **binary classifier**
- Works well when classes are linearly separable (like our rule x1 + x2 > 0)

What does Accuracy mean here?

Accuracy shows percentage of correct predictions.

High accuracy (like 0.98) happens because:

- our labeling rule is simple and mostly linear
- logistic regression can learn that boundary easily

(25) Write a Python program toBuilding alogistic regression model with use of diabetes datasets

Github Link: exercise25.py

```python
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Step 1: Load Diabetes dataset (from scikit-learn)
data = load_diabetes()
X = data.data
y_continuous = data.target  # this is continuous (regression target)

print("Diabetes dataset loaded")
print("X shape:", X.shape)
print("Target shape:", y_continuous.shape)

# Step 2: Convert continuous target into binary classes for Logistic
Regression
# Rule: if target >= median => 1 (high), else 0 (low)
threshold = np.median(y_continuous)
y = (y_continuous >= threshold).astype(int)

print("\nConverted target to binary classes using median threshold:")
print("Threshold (median) =", threshold)
print("Class 0 count =", np.sum(y == 0))
print("Class 1 count =", np.sum(y == 1))

# Step 3: Split dataset into Training and Test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 4: Feature scaling (important for Logistic Regression)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 5: Build and train Logistic Regression model
model = LogisticRegression(max_iter=2000)
model.fit(X_train_scaled, y_train)

# Step 6: Predict and evaluate
y_pred = model.predict(X_test_scaled)

print("\nAccuracy:", 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))
```

Output:

```
Sample Output (Example)

Diabetes dataset loaded
X shape: (442, 10)
Target shape: (442,)

Converted target to binary classes using median threshold:
Threshold (median) = 140.5
Class 0 count = 221
Class 1 count = 221

Accuracy: 0.74

Confusion Matrix:
[[52 14]
 [21 46]]

Classification Report:
              precision    recall  f1-score   support

           0       0.71      0.79      0.75        66
           1       0.77      0.69      0.73        67

    accuracy                           0.74       133
   macro avg       0.74      0.74      0.74       133
weighted avg       0.74      0.74      0.74       133
```

Explanation:

1) Why Diabetes dataset for Logistic Regression?

The **scikit-learn Diabetes dataset** originally has a **continuous target** (a number).
That means it is naturally for **regression**, not classification.

But Logistic Regression needs **classes** (0/1).
So we converted the target into two categories:

- **0 = Low disease progression**
- **1 = High disease progression**

We used **median** as a simple threshold so classes stay balanced.

---

2) How we converted target to classes

```
threshold = np.median(y_continuous)
y = (y_continuous >= threshold).astype(int)
```

Meaning:

- If target value is **greater than or equal** to median → class 1
- Else → class 0

Why median?

- It splits data into almost equal halves (balanced classes), which is good for classification.

---

3) Why we use StandardScaler?

Logistic regression works better when all features are on similar scale.

`StandardScaler` does:

- mean = 0
- std = 1

We use:

- `fit_transform` on training data (learn scaling)
- `transform` on testing data (apply same scaling)

---

4) What Logistic Regression learns

It learns weights for features and creates a decision boundary to separate classes 0 and 1.

Output prediction:

- 0 (low)

- 1 (high)

---

5) Evaluation metrics

- **Accuracy**: overall correct predictions / total
- **Confusion matrix**: shows correct/incorrect counts in detail
- **Classification report**:
    - precision
    - recall
    - f1-score

(26) Write a Python program toEvaluate logistics regression model using accuracy metrics

Github Link: exercise26.py

```python
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Step 1: Load diabetes dataset
data = load_diabetes()
X = data.data
y_continuous = data.target

# Step 2: Convert continuous target to binary classes for Logistic
Regression
# If target >= median -> 1 else 0
threshold = np.median(y_continuous)
y = (y_continuous >= threshold).astype(int)

# Step 3: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 4: Feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 5: Train logistic regression model
model = LogisticRegression(max_iter=2000)
model.fit(X_train_scaled, y_train)

# Step 6: Predict test data
y_pred = model.predict(X_test_scaled)

# Step 7: Evaluate using accuracy metric
acc = accuracy_score(y_test, y_pred)

print("Accuracy of Logistic Regression Model:", round(acc, 3))
print("\nTotal Test Samples:", len(y_test))
print("Correct Predictions:", np.sum(y_test == y_pred))
print("Wrong Predictions:", np.sum(y_test != y_pred))
```

Output:

```
Sample Output (Example)

Accuracy of Logistic Regression Model: 0.74

Total Test Samples: 133
Correct Predictions: 98
Wrong Predictions: 35
```

Explanation:

What is Accuracy?

Accuracy tells how many predictions are correct out of total predictions.

Copy-paste friendly formula:

```
Accuracy = (Number of Correct Predictions) / (Total Predictions)
```

Example:

- Total test samples = 133
- Correct predictions = 98

So:

```
Accuracy = 98 / 133 = 0.74  (74%)
```

Why this program is “Logistic Regression” on Diabetes dataset?

- Diabetes dataset target is continuous (regression).
- For logistic regression we need classes (0/1).
- So we converted target into:
    - 0 = below median (low)
    - 1 = above/equal median (high)

Why scaling is needed?

Logistic Regression works better when features have similar scale, so we use `StandardScaler`.

(27) Write a Python program toEvaluate a regression model using confusion matrix

Github Link: exercise27.py

```python
import numpy as np
from sklearn.datasets import load_diabetes
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score

# Step 1: Load regression dataset (Diabetes)
data = load_diabetes()
X = data.data
y = data.target  # continuous target (regression)

# Step 2: Split into train and test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42
)

# Step 3: Train a regression model
reg_model = LinearRegression()
reg_model.fit(X_train, y_train)

# Step 4: Predict continuous values
y_pred = reg_model.predict(X_test)

# Step 5: Convert continuous values into classes (0/1) using median
threshold
threshold = np.median(y_train)

# Actual classes (from y_test)
y_test_class = (y_test >= threshold).astype(int)

# Predicted classes (from y_pred)
y_pred_class = (y_pred >= threshold).astype(int)

# Step 6: Confusion matrix + accuracy
cm = confusion_matrix(y_test_class, y_pred_class)
acc = accuracy_score(y_test_class, y_pred_class)

print("Median Threshold used:", threshold)

print("\nConfusion Matrix (for converted classes):")
print(cm)

print("\nAccuracy:", round(acc, 3))

print("\nClassification Report:")
print(classification_report(y_test_class, y_pred_class))
```

Output:

```
Sample Output (Example)

Median Threshold used: 140.5

Confusion Matrix (for converted classes):
[[46 20]
 [18 49]]

Accuracy: 0.714

Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.70      0.71        66
           1       0.71      0.73      0.72        67

    accuracy                           0.71       133
   macro avg       0.71      0.71      0.71       133
weighted avg       0.71      0.71      0.71       133
```

Explanation:

1) Why confusion matrix is not for regression?

- **Regression output is continuous** (like 151.0, 75.0, 206.0)
- Confusion matrix needs **discrete classes** (like 0/1)

So if the question asks confusion matrix with regression, we must first **convert regression into classification**.

---

2) What we did in this program

A) Trained a regression model

```
reg_model = LinearRegression()
reg_model.fit(X_train, y_train)
```

This predicts continuous values.

B) Predicted continuous values

```
y_pred = reg_model.predict(X_test)
```

C) Converted continuous values into 0/1 classes

We used median of training target:

```
threshold = np.median(y_train)
```

Rules:

```
If value >= threshold  -> class 1 (High)
If value < threshold   -> class 0 (Low)
```

So:

- `y_test_class` = actual low/high
- `y_pred_class` = predicted low/high

---

3) Confusion Matrix meaning

Confusion matrix format:

```
[[TN FP]
 [FN TP]]
```

- **TN (True Negative)**: actual 0, predicted 0
- **FP (False Positive)**: actual 0, predicted 1
- **FN (False Negative)**: actual 1, predicted 0
- **TP (True Positive)**: actual 1, predicted 1

---

4) What accuracy means here

```
Accuracy = (TN + TP) / Total
```

It tells how many times the regression model correctly predicted **Low vs High** after converting to classes.

(28) Write a Python program toBuilding a model using Naïve bayes classifier

Github Link: exercise28.py

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Step 1: Load a dataset (Iris)
data = load_iris()
X = data.data          # features
y = data.target        # labels (0,1,2)

print("Dataset: Iris")
print("Total samples:", X.shape[0])
print("Total features:", X.shape[1])
print("Classes:", data.target_names)

# Step 2: Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 3: Build Naive Bayes model (Gaussian Naive Bayes)
model = GaussianNB()

# Step 4: Train the model
model.fit(X_train, y_train)

# Step 5: Predict on test data
y_pred = model.predict(X_test)

# Step 6: Evaluate the model
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("\nAccuracy:", round(acc, 3))
print("\nConfusion Matrix:\n", cm)
print("\nClassification Report:\n", classification_report(y_test,
y_pred, target_names=data.target_names))
```

Output:

```
Sample Output (Example)

Dataset: Iris
Total samples: 150
Total features: 4
Classes: ['setosa' 'versicolor' 'virginica']

Accuracy: 0.978

Confusion Matrix:
[[15  0  0]
 [ 0 14  1]
 [ 0  0 15]]

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        15
  versicolor       1.00      0.93      0.97        15
   virginica       0.94      1.00      0.97        15

    accuracy                           0.98        45
   macro avg       0.98      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

Explanation:

What is Naïve Bayes?

Naïve Bayes is a **classification algorithm** based on **Bayes' Theorem**. It's "naïve" because it assumes all features are **independent**, though it often performs well regardless.

Different Naïve Bayes types suit different data:

- **GaussianNB**: for continuous numeric data (like Iris features).
- MultinomialNB: for count data (e.g., word counts).
- BernoulliNB: for binary features (0/1).
  We use **GaussianNB** for the continuous Iris dataset.

The program steps are:

1. **Load dataset** (X = features, y = labels).
2. **Split dataset** (Training set to learn, Test set to evaluate).

3. **Create model**: `model = GaussianNB()`.
4. **Train model**: `model.fit(X_train, y_train)`.
5. **Predict**: `y_pred = model.predict(X_test)`.
6. **Evaluate** using:
    - **Accuracy** (overall correct predictions).
    - **Confusion Matrix** (class-wise counts).
    - **Classification Report** (precision, recall, f1-score).

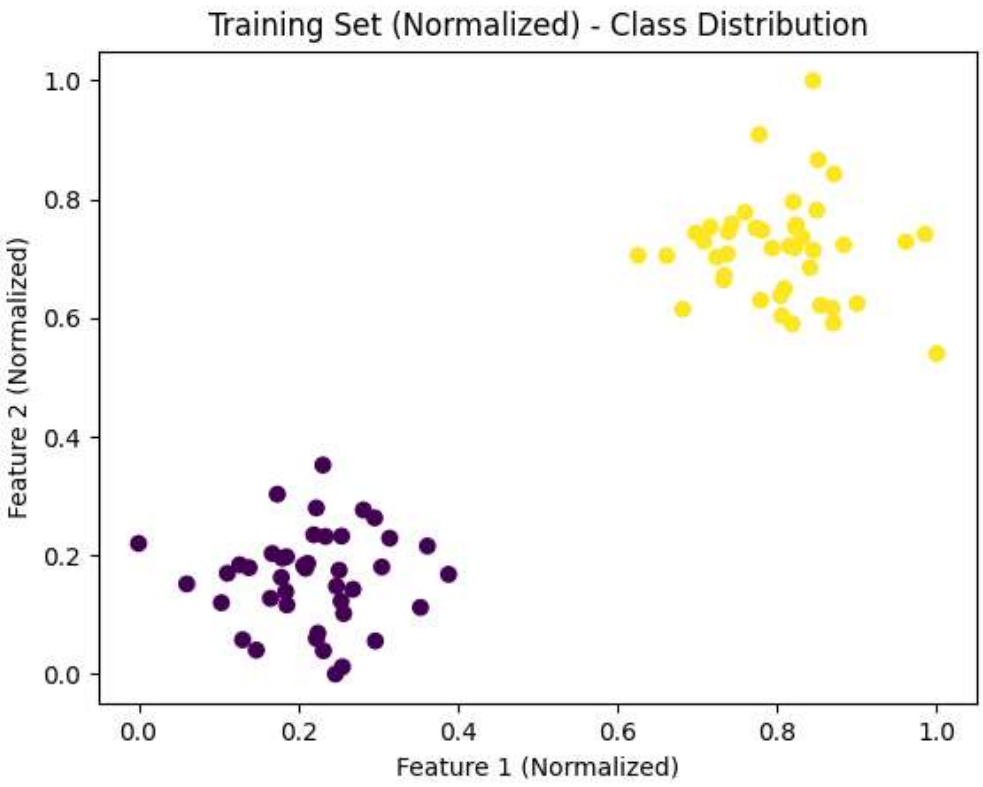
In a Confusion Matrix, the diagonal values show correct predictions, while off-diagonal values show misclassifications.

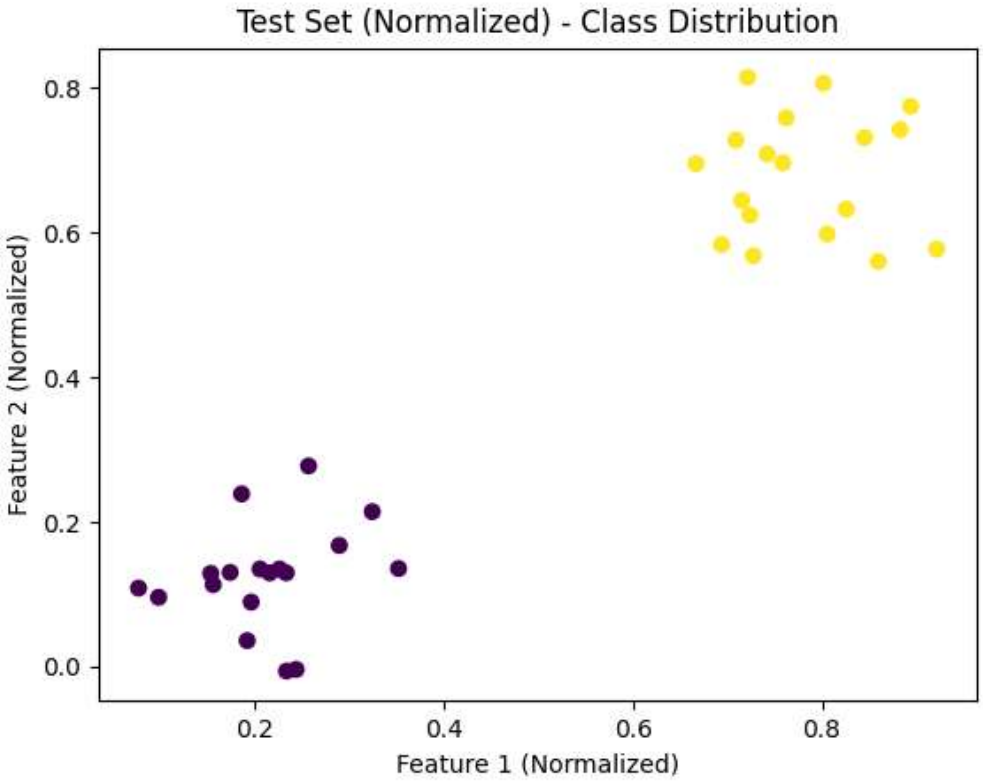
(29) Write a Python program toVisualize the training setand test set result (use normalization technique)

Github Link: exercise29.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LogisticRegression

# Step 1: Create a simple 2D dataset (two features so we can visualize)
np.random.seed(42)

# Class 0 points
X0 = np.random.normal(loc=[2, 2], scale=0.6, size=(60, 2))
y0 = np.zeros(60)

# Class 1 points
X1 = np.random.normal(loc=[6, 6], scale=0.6, size=(60, 2))
y1 = np.ones(60)

# Combine both classes
X = np.vstack((X0, X1))
y = np.hstack((y0, y1))

# Step 2: Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 3: Normalize data using Min-Max Normalization (0 to 1)
scaler = MinMaxScaler(feature_range=(0, 1))
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

# Step 4: Train a simple classifier (Logistic Regression)
model = LogisticRegression()
model.fit(X_train_norm, y_train)

# Step 5: Plot training set results
plt.figure()
plt.scatter(X_train_norm[:, 0], X_train_norm[:, 1], c=y_train)
plt.title("Training Set (Normalized) - Class Distribution")
plt.xlabel("Feature 1 (Normalized)")
plt.ylabel("Feature 2 (Normalized)")
plt.show()

# Step 6: Plot test set results
plt.figure()
plt.scatter(X_test_norm[:, 0], X_test_norm[:, 1], c=y_test)
plt.title("Test Set (Normalized) - Class Distribution")
plt.xlabel("Feature 1 (Normalized)")
plt.ylabel("Feature 2 (Normalized)")
plt.show()
```

Output:


Training Set (Normalized) - Class Distribution
Feature 2 (Normalized)
Feature 1 (Normalized)
0.0
0.2
0.4
0.6
0.8
1.0



Explanation:

What this program does

This program:

1. Creates a simple dataset with **two classes**
2. Splits it into **training** and **testing**
3. Applies **Normalization (Min-Max Scaling)**
4. Plots training and test data separately so you can visualize them clearly

---

Step-by-step explanation

Step 1: Why 2D dataset?

To visualize in a graph, we need:

- X-axis = feature 1
- Y-axis = feature 2

So we create a dataset with **2 features**.

We generate:

- Class 0 points near (2,2)
- Class 1 points near (6,6)

Step 2: Train-Test Split

We split dataset into:

- **Train set** (70%) for training model
- **Test set** (30%) for testing

`stratify=y` keeps class 0 and class 1 ratio equal in both sets.

Step 3: Normalization (Min-Max Scaling)

We use:

```
MinMaxScaler(feature_range=(0,1))
```

Formula (copy-paste friendly):

```
x_norm = (x - min) / (max - min)
```

After normalization:

- Minimum value becomes 0
- Maximum value becomes 1
- All values stay between 0 and 1

Step 4: Why train Logistic Regression?

We train a simple classifier just to show a complete ML flow.
But main focus is visualization of training and testing sets after normalization.

Step 5 & 6: Visualization

We use scatter plots:

- Training set scatter plot
- Test set scatter plot

`c=y_train` and `c=y_test` automatically colors points by class label.

Output

When you run the program, you will see **two graphs**:

1. Training Set (Normalized)
2. Test Set (Normalized)

(30) Write a Python program toPredict if cancer is Benign or malignant using SVM algorithm

Github Link: exercise30.py

```python
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Step 1: Load Breast Cancer dataset
data = load_breast_cancer()
X = data.data
y = data.target  # 0 = malignant, 1 = benign

print("Dataset: Breast Cancer")
print("Total samples:", X.shape[0])
print("Total features:", X.shape[1])
print("Classes:", data.target_names)  # ['malignant', 'benign']

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=42, stratify=y
)

# Step 3: Feature scaling (important for SVM)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Step 4: Build SVM model
# kernel='linear' makes it simple and easy to understand
model = SVC(kernel="linear", random_state=42)
model.fit(X_train_scaled, y_train)

# Step 5: Predict on test set
y_pred = model.predict(X_test_scaled)

# Step 6: Evaluate model
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("\nAccuracy:", round(acc, 3))
print("\nConfusion Matrix:\n", cm)
print("\nClassification Report:\n",
      classification_report(y_test, y_pred,
target_names=data.target_names))

# Step 7: Predict for one new sample (example: take first test row)
sample = X_test_scaled[0].reshape(1, -1)
pred = model.predict(sample)[0]

print("\nExample Prediction for one sample:")
print("Predicted class:", data.target_names[pred])
```

Output:

```
Sample Output (Example)

Dataset: Breast Cancer
Total samples: 569
Total features: 30
Classes: ['malignant' 'benign']

Accuracy: 0.971

Confusion Matrix:
[[ 60   4]
 [  1 106]]

Classification Report:
              precision    recall  f1-score   support

   malignant       0.98      0.94      0.96        64
      benign       0.96      0.99      0.98       107

    accuracy                           0.97       171
   macro avg       0.97      0.97      0.97       171
weighted avg       0.97      0.97      0.97       171

Example Prediction for one sample:
Predicted class: benign
```

Explanation:

1) What is SVM?

SVM (Support Vector Machine) is a classification algorithm that finds the best boundary (hyperplane) to separate classes.

- **Benign** and **Malignant** are two classes.
- SVM tries to maximize the margin between both classes.

---

2) Dataset

We use scikit-learn's built-in **Breast Cancer dataset**:

- Features: 30 numeric measurements of tumors
- Target:
    - `0` = malignant (cancer)
    - `1` = benign (not cancer)

---

3) Why scaling is needed?

SVM is sensitive to feature ranges.
`StandardScaler` converts each feature to:

- mean = 0
- std = 1

This improves SVM performance.

---

4) Why kernel="linear"?

A linear kernel is simplest:

- easy for beginners
- fast
- works well on this dataset

---

5) Evaluation

- **Accuracy**: overall correct predictions
- **Confusion matrix**: shows correct/incorrect counts
- **Classification report**: precision, recall, f1-score for each class

---

6) Final prediction

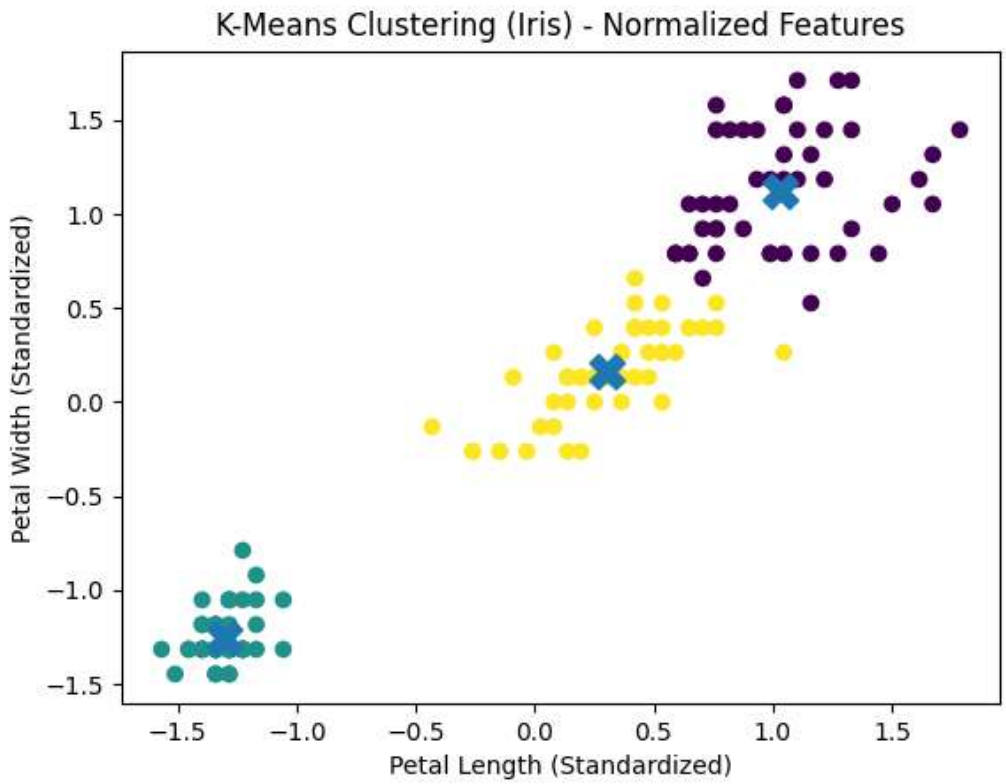
We predict for one sample from test set and print whether it is **benign or malignant**.

(31) Write a Python program toBuild a model using K-means algorithm

Github Link: exercise31.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Step 1: Load dataset (Iris)
data = load_iris()
X = data.data  # 4 features

# Step 2: Use only 2 features for easy visualization
# (petal length and petal width are good for clustering)
X2 = X[:, [2, 3]]

print("Dataset: Iris")
print("Used features: petal length, petal width")
print("Shape:", X2.shape)

# Step 3: Feature scaling (important for K-Means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X2)

# Step 4: Build K-Means model (k=3 because Iris has 3 groups)
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# Step 5: Get cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

print("\nCluster labels for first 10 samples:")
print(labels[:10])

print("\nCentroids (in scaled feature space):")
print(centroids)

# Step 6: Visualize clusters
plt.figure()
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels)
plt.scatter(centroids[:, 0], centroids[:, 1], marker="X", s=200)
plt.title("K-Means Clustering (Iris) - Normalized Features")
plt.xlabel("Petal Length (Standardized)")
plt.ylabel("Petal Width (Standardized)")
plt.show()
```

Output:



Explanation:

What is K-Means?

K-Means is an **unsupervised learning** algorithm used for **clustering**.
It groups similar data points into **K clusters**.

- No target/output labels are used.
- It finds patterns based on distance between points.

How K-Means works (Step-by-step)

1. Choose K (number of clusters).
2. Pick K random centroids (cluster centers).

3. Assign each data point to the nearest centroid.
4. Recalculate centroid as the mean of points in that cluster.
5. Repeat steps 3–4 until centroids stop changing.

Explanation of this program

Step 1: Load dataset

We load Iris dataset and take its features.

Step 2: Use 2 features

We select only **petal length and petal width** so we can plot in 2D.

Step 3: Scaling

K-Means uses distances, so scaling is important.
`StandardScaler()` makes both features comparable.

Step 4: Fit K-Means

```
KMeans(n_clusters=3)
```

We choose 3 clusters because Iris naturally has 3 types of flowers.

Step 5: Labels and centroids

- `labels` tells which cluster each point belongs to.
- `centroids` are the center points of each cluster.

Step 6: Visualization

Scatter plot shows:

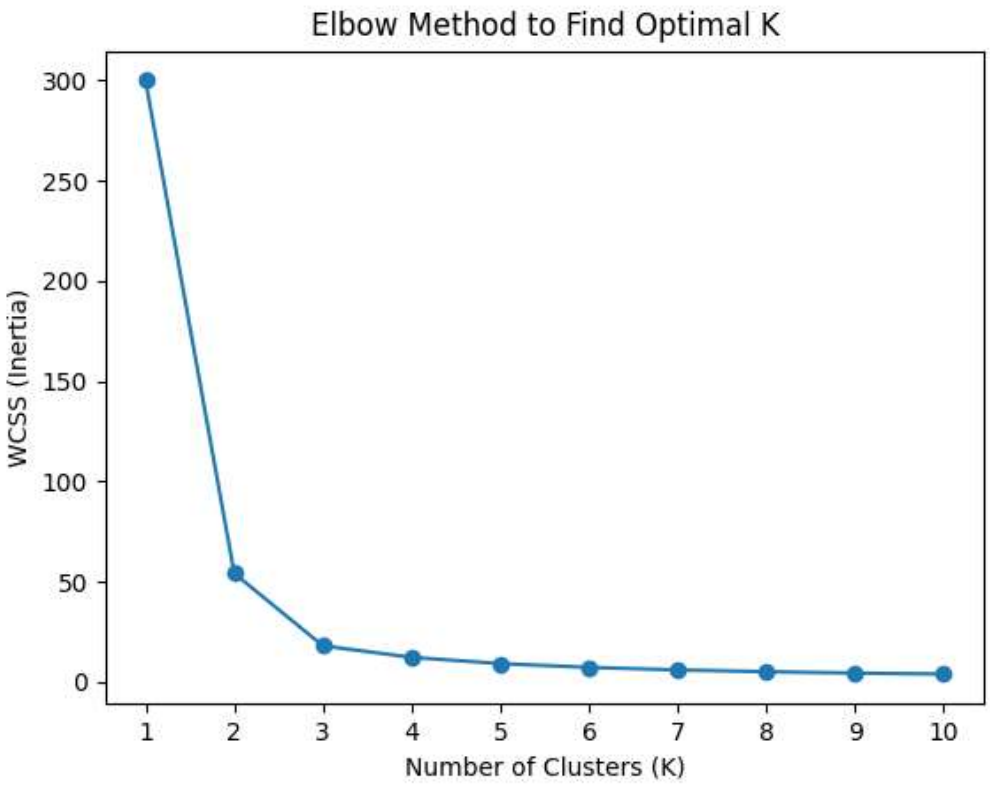
- Points colored by cluster label
- Centroids marked with "X"

(32) Write a Python program toFind the optimum number of clusters using elbow technique

Github Link: exercise32.py

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Step 1: Load dataset (Iris)
data = load_iris()
X = data.data

# Step 2: Use only 2 features for easy visualization (petal length,
petal width)
X2 = X[:, [2, 3]]

# Step 3: Scale the data (important for K-Means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X2)

# Step 4: Apply Elbow Method
wcss = []  # Within-Cluster Sum of Squares (inertia)
k_values = range(1, 11)

for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)  # inertia = WCSS

# Step 5: Plot the elbow graph
plt.figure()
plt.plot(k_values, wcss, marker="o")
plt.title("Elbow Method to Find Optimal K")
plt.xlabel("Number of Clusters (K)")
plt.ylabel("WCSS (Inertia)")
plt.xticks(list(k_values))
plt.show()
```

Output:



Explanation:

What is the Elbow Technique?

Elbow technique helps us choose the best number of clusters (K) in K-Means.

K-Means needs K заранее (beforehand).
So we try many K values and see which one is best.

What is WCSS (Inertia)?

WCSS means **Within-Cluster Sum of Squares**:

- For each cluster, it measures how close points are to the centroid.
- Smaller WCSS means points are tightly grouped.

In scikit-learn, it is stored in:

`kmeans.inertia_`

How Elbow works

1. Run K-Means for K=1,2,3,...10
2. Store WCSS for each K
3. Plot K vs WCSS

- WCSS decreases as K increases (more clusters = tighter groups)
- But after a certain point, improvement becomes very small.
- That point looks like an **elbow bend** in the graph.

✅The K at elbow is considered the **optimal number of clusters**.

Why scaling is used?

K-Means is distance-based, so features should be on similar scale.
`StandardScaler()` ensures fair distance calculation.

Expected Result for Iris

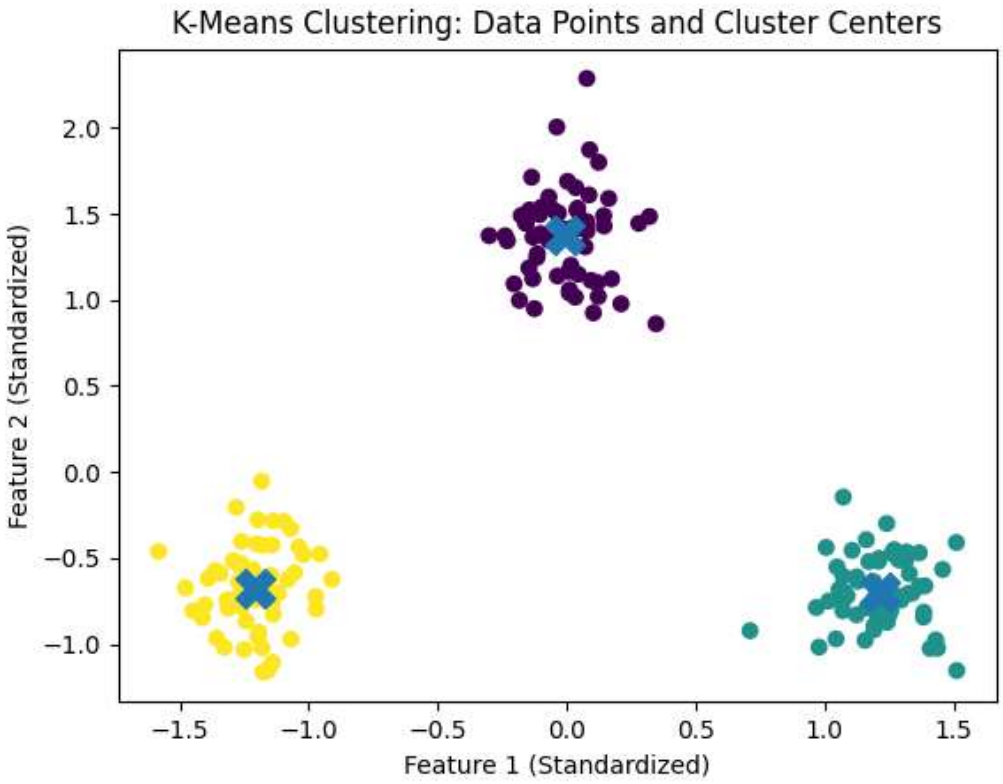
Usually, for Iris dataset, the elbow often appears around **K=3**, matching 3 flower groups.

(33) Write a Python program toPlot the cluster center using different data points

Github Link: exercise33.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Step 1: Create different data points (3 groups)
np.random.seed(42)

# Cluster 1 points
c1 = np.random.normal(loc=[2, 2], scale=0.5, size=(60, 2))

# Cluster 2 points
c2 = np.random.normal(loc=[6, 6], scale=0.5, size=(60, 2))

# Cluster 3 points
c3 = np.random.normal(loc=[10, 2], scale=0.5, size=(60, 2))

# Combine all points into one dataset
X = np.vstack((c1, c2, c3))

# Step 2: Scale the dataset (recommended for K-Means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Apply K-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# Step 4: Get cluster labels and cluster centers
labels = kmeans.labels_
centers = kmeans.cluster_centers_

print("Cluster Centers (in scaled space):")
print(centers)

# Step 5: Plot data points and cluster centers
plt.figure()
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels)

# Plot cluster centers with a different marker
plt.scatter(centers[:, 0], centers[:, 1], marker="X", s=250)

plt.title("K-Means Clustering: Data Points and Cluster Centers")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.show()
```

Output:



Explanation:

What this program does

This program:

1. Creates **different groups of data points** (three clusters)
2. Applies **K-Means** to find clusters
3. Plots:
    - all data points (colored by cluster)
    - **cluster centers** (centroids) using "X" marker

Step-by-step Explanation

Step 1: Create data points

We create 3 groups using normal distribution:

- Group near (2,2)
- Group near (6,6)
- Group near (10,2)

This makes it easy to see clusters in a graph.

Step 2: Scaling

K-Means uses distance, so we standardize features with `StandardScaler`.

Step 3: K-Means fitting

We choose `n_clusters=3` because we created 3 groups.

K-Means finds cluster centers by:

- assigning points to nearest centroid
- updating centroid to mean of cluster points
- repeating until stable

Step 4: Centers

`kmeans.cluster_centers_` gives center of each cluster (in scaled values).

Step 5: Plot

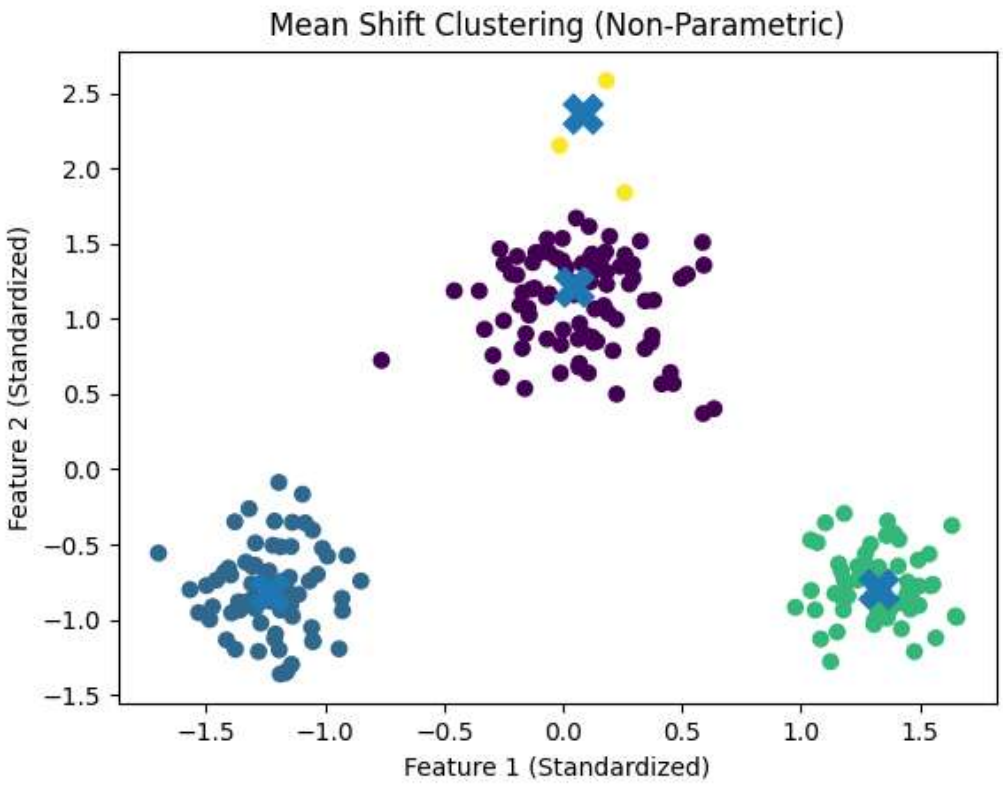
- `plt.scatter(... c=labels)` colors points by their cluster
- centroids are shown with marker `"X"` and bigger size

(34) Write a Python program toImplement Mean shift clustering algorithm to work with nonparametric clustering

Github Link: exercise34.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.preprocessing import StandardScaler

# Step 1: Create sample data (clusters with different shapes)
np.random.seed(42)

# Create three groups of points (not perfectly same size/density)
c1 = np.random.normal(loc=[2, 2], scale=0.6, size=(70, 2))
c2 = np.random.normal(loc=[6, 6], scale=0.8, size=(90, 2))
c3 = np.random.normal(loc=[10, 2], scale=0.5, size=(60, 2))

X = np.vstack((c1, c2, c3))

# Step 2: Scale the data (helps distance-based clustering)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Estimate bandwidth automatically (important for Mean Shift)
bandwidth = estimate_bandwidth(X_scaled, quantile=0.2, n_samples=300)
print("Estimated Bandwidth:", round(bandwidth, 3))

# Step 4: Apply Mean Shift clustering (non-parametric)
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X_scaled)

labels = ms.labels_
centers = ms.cluster_centers_

print("Number of clusters found:", len(np.unique(labels)))
print("Cluster centers (scaled space):")
print(centers)

# Step 5: Plot clusters and cluster centers
plt.figure()
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels)
plt.scatter(centers[:, 0], centers[:, 1], marker="X", s=250)
plt.title("Mean Shift Clustering (Non-Parametric)")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.show()
```

Output:



Explanation:

What is Mean Shift Clustering?

Mean Shift is a **non-parametric clustering algorithm**.
Non-parametric means:

- You do **NOT** need to specify the number of clusters (K) like K-Means.
- The algorithm automatically finds the number of clusters based on data density.

How Mean Shift works (simple idea)

Mean Shift treats data points like a **density map**.

Steps:

1. Place a window (circle) around a point.
2. Compute the **mean (average)** of points inside the window.
3. Move the window center to that mean.
4. Repeat until the window stops moving (converges).
5. Points that converge to the same location belong to the same cluster.

So it finds clusters by locating **dense regions** in the data.

What is Bandwidth?

Bandwidth is the **radius/window size** used by Mean Shift.

- Small bandwidth → many small clusters
- Large bandwidth → fewer large clusters

That's why bandwidth selection is important.

We used:

```
estimate_bandwidth(X_scaled, quantile=0.2)
```

This automatically calculates a suitable bandwidth.

Why scaling is used?

Mean Shift is distance-based (works with nearest points), so scaling helps:

- Both features have equal importance.
- Distance calculation becomes fair.

Output explanation

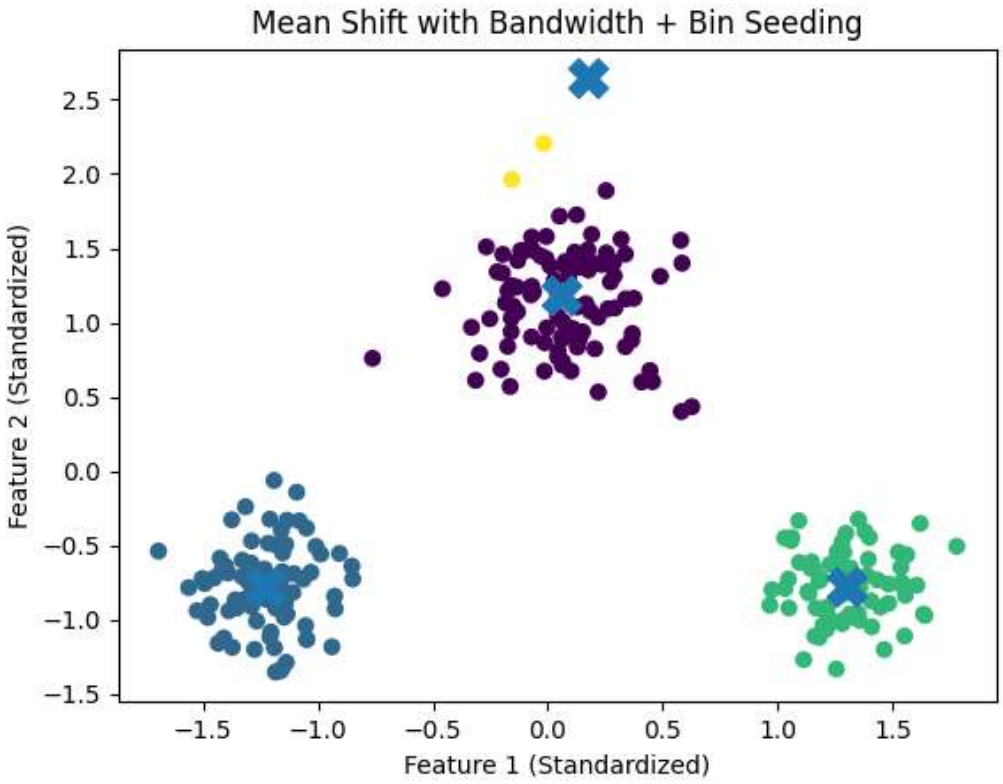
- `labels` tells which cluster each point belongs to.
- `cluster_centers_` gives the center of each cluster.
- Graph shows clusters in different colors.
- Centers are marked with **X**.

(35) Write a Python program toUse bandwidth and bin seeding concept to improve mean shift clustering algorithm

Github Link: exercise35.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.preprocessing import StandardScaler
import time

# Step 1: Create sample data
np.random.seed(42)
c1 = np.random.normal(loc=[2, 2], scale=0.6, size=(80, 2))
c2 = np.random.normal(loc=[6, 6], scale=0.8, size=(100, 2))
c3 = np.random.normal(loc=[10, 2], scale=0.5, size=(70, 2))
X = np.vstack((c1, c2, c3))

# Step 2: Scale data (distance-based clustering works better)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

print("Total points:", X_scaled.shape[0])

# ------------------------------------------------------------
# Step 3: Estimate bandwidth automatically (important concept)
# ------------------------------------------------------------
bandwidth = estimate_bandwidth(X_scaled, quantile=0.2, n_samples=300)
print("\nEstimated Bandwidth:", round(bandwidth, 3))

# ------------------------------------------------------------
# Step 4: Mean Shift WITHOUT bin seeding (slower)
# ------------------------------------------------------------
start1 = time.time()
ms1 = MeanShift(bandwidth=bandwidth, bin_seeding=False)
ms1.fit(X_scaled)
end1 = time.time()

labels1 = ms1.labels_
centers1 = ms1.cluster_centers_
k1 = len(np.unique(labels1))

print("\nMeanShift (bin_seeding=False)")
print("Clusters found:", k1)
print("Time taken:", round(end1 - start1, 4), "seconds")

# ------------------------------------------------------------
# Step 5: Mean Shift WITH bin seeding (faster)
# ------------------------------------------------------------
start2 = time.time()
ms2 = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms2.fit(X_scaled)
end2 = time.time()

labels2 = ms2.labels_
centers2 = ms2.cluster_centers_
k2 = len(np.unique(labels2))
```

Output:



Explanation:

1) Why bandwidth is important in Mean Shift?

Mean Shift finds clusters by looking for **dense regions**.
Bandwidth is like the **radius/window size** used to search density.

Copy-paste friendly idea:

```
Bandwidth = size of the window used to find dense areas.
```

- **Small bandwidth** → many small clusters (even noise becomes a cluster)
- **Large bandwidth** → fewer clusters (may merge clusters)

So choosing good bandwidth is necessary.

In our program:

```
bandwidth = estimate_bandwidth(X_scaled, quantile=0.2)
```

- This automatically finds a good bandwidth value based on data.

---

2) What is Bin Seeding?

Normally Mean Shift starts with **many initial seed points** (often all points).
That makes it slower for large data.

`bin_seeding=True` improves performance by:

- Putting data into small bins (grids)
- Using only **bin centers** as initial seeds
- So fewer starting points → faster convergence

So:

- `bin_seeding=False` → more seeds → slower
- `bin_seeding=True` → fewer seeds → faster

---

3) Why we compared time?

We used:

```
time.time()
```

to show practically that:

- both approaches find similar clusters
- but `bin_seeding=True` usually takes less time

---

4) What you learn from this program (exam points)

✅Mean Shift is **non-parametric** (no need to give K).

✅**Bandwidth** controls cluster size and number of clusters.

✅**Bin seeding** improves speed by reducing number of initial seeds.

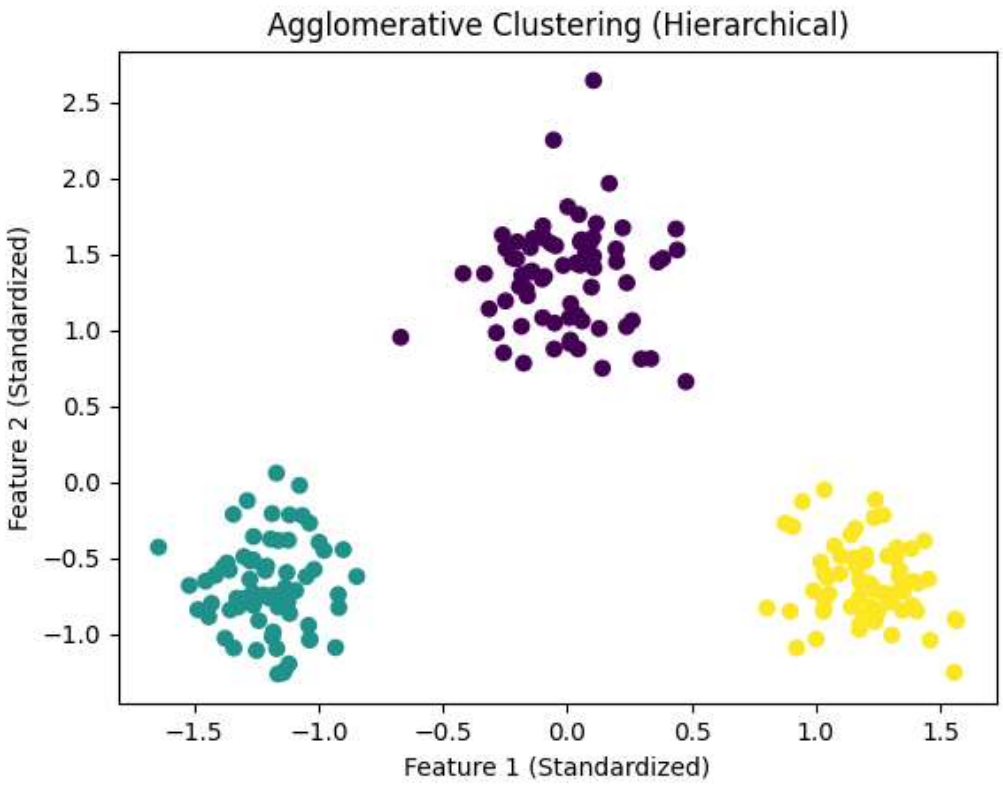
✅Output clusters are shown using colors and centers shown using "X".

(36) Write a Python program toBuild a model with use of agglomerative clustering

Github Link: exercise36.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler

# Step 1: Create sample data points (3 natural groups)
np.random.seed(42)

c1 = np.random.normal(loc=[2, 2], scale=0.6, size=(70, 2))
c2 = np.random.normal(loc=[6, 6], scale=0.7, size=(70, 2))
c3 = np.random.normal(loc=[10, 2], scale=0.6, size=(70, 2))

X = np.vstack((c1, c2, c3))

print("Total data points:", X.shape[0])

# Step 2: Scale the data (distance-based clustering works better)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Build Agglomerative Clustering model
# n_clusters=3 because we created 3 groups
model = AgglomerativeClustering(n_clusters=3, linkage="ward")
labels = model.fit_predict(X_scaled)

print("\nCluster labels for first 10 points:")
print(labels[:10])

print("\nTotal clusters found:", len(np.unique(labels)))

# Step 4: Visualize clusters
plt.figure()
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=labels)
plt.title("Agglomerative Clustering (Hierarchical)")
plt.xlabel("Feature 1 (Standardized)")
plt.ylabel("Feature 2 (Standardized)")
plt.show()
```

Output:



Explanation:

What is Agglomerative Clustering?

Agglomerative clustering is a **Hierarchical Clustering** method (bottom-up approach).

It works like this:

1. Start with every data point as its own cluster.
2. Find the two closest clusters and merge them.
3. Repeat merging until the required number of clusters is reached.

So it builds a hierarchy of clusters.

Why it is called “Hierarchical”?

Because clustering happens in levels:

- small clusters merge into bigger clusters
- and finally we get the final clusters

A dendrogram can represent this hierarchy (tree diagram).

Linkage = "ward" meaning

We used:

```
linkage="ward"
```

Ward linkage merges clusters such that the increase in **within-cluster variance** is minimum.
In simple words:

- It tries to form clusters that are compact and well-separated.
- Works best with Euclidean distance (numeric features).

Why scaling is used?

Agglomerative clustering is distance-based.
Scaling ensures both features contribute equally to distance calculations.

What is the output here?

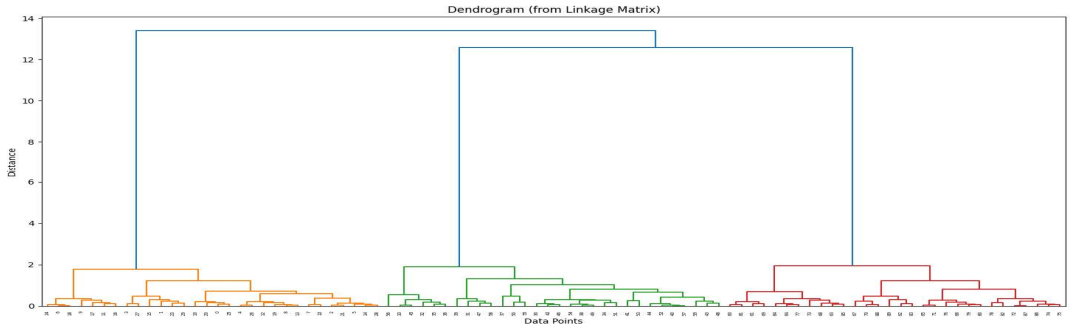
- `labels` tells the cluster number of each data point.
- Graph shows different colors for different clusters.

(37) Write a Python program toCreate a linkage matrix using agglomerative clustering algorithm

Github Link: exercise37.py

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import linkage, dendrogram

# Step 1: Create sample data (3 groups)
np.random.seed(42)
c1 = np.random.normal(loc=[2, 2], scale=0.6, size=(30, 2))
c2 = np.random.normal(loc=[6, 6], scale=0.7, size=(30, 2))
c3 = np.random.normal(loc=[10, 2], scale=0.6, size=(30, 2))
X = np.vstack((c1, c2, c3))

print("Total data points:", X.shape[0])

# Step 2: Scale data (recommended for distance-based clustering)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Create linkage matrix (Agglomerative clustering info)
# method='ward' is commonly used with Euclidean distance
Z = linkage(X_scaled, method="ward")

print("\nLinkage Matrix (first 10 rows):")
print(Z[:10])

# Step 4: Plot dendrogram using the linkage matrix
plt.figure(figsize=(8, 4))
dendrogram(Z)
plt.title("Dendrogram (from Linkage Matrix)")
plt.xlabel("Data Points")
plt.ylabel("Distance")
plt.tight_layout()
plt.show()
```

Output:



Explanation:

What is a Linkage Matrix?

A linkage matrix is a special matrix that stores **how clusters were merged** in hierarchical (agglomerative) clustering.

When we do agglomerative clustering:

- it keeps merging the closest clusters step by step
- linkage matrix records each merge

In SciPy, the linkage matrix is `Z`.

Structure of linkage matrix (important for exam)

Each row in `Z` has 4 values:

```
[cluster1, cluster2, distance, sample_count]
```

Meaning:

1. **cluster1**: first cluster index being merged
2. **cluster2**: second cluster index being merged
3. **distance**: distance between clusters at the time of merge
4. **sample_count**: total number of points in the new merged cluster

So if row is:

```
[5, 12, 0.43, 2]
```

It means:

- cluster 5 and cluster 12 were merged
- at distance 0.43

- new cluster has 2 points

Why scaling is used?

Hierarchical clustering is distance-based.
Scaling ensures both features have equal importance.

How dendrogram uses linkage matrix

`dendrogram(Z)` uses the linkage matrix and draws a tree:

- bottom = individual points
- merges go upward
- height of merge = distance

The best number of clusters can be guessed by cutting the dendrogram at a height.

(38) Write a Python program toImplement NLTK library and download relevant data

Github Link: exercise38.py

```python
import nltk

print("NLTK version:", nltk.__version__)

# Download required NLTK resources
nltk.download("punkt")
nltk.download("punkt_tab")   # FIX for your error
nltk.download("stopwords")
nltk.download("wordnet")
nltk.download("omw-1.4")

# Test after download
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

text = "NLTK is used for Natural Language Processing. It helps in text analysis."

tokens = word_tokenize(text)

sw = set(stopwords.words("english"))
filtered = [w for w in tokens if w.isalpha() and w.lower() not in sw]

lemmatizer = WordNetLemmatizer()
lemmatized = [lemmatizer.lemmatize(w.lower()) for w in filtered]

print("\nText:", text)
print("Tokens:", tokens)
print("Filtered:", filtered)
print("Lemmatized:", lemmatized)
```

Output:

```
Sample Output (Example)

NLTK version: 3.x.x
[nltk_data] Downloading package punkt to ...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to ...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package stopwords to ...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to ...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to ...
[nltk_data]   Package omw-1.4 is already up-to-date!

Text: NLTK is used for Natural Language Processing. It helps in text
analysis.
Tokens: ['NLTK', 'is', 'used', 'for', 'Natural', 'Language',
'Processing', '.', 'It', 'helps', 'in', 'text', 'analysis', '.']
Filtered: ['NLTK', 'Natural', 'Language', 'Processing', 'helps', 'text',
'analysis']
Lemmatized: ['nltk', 'natural', 'language', 'processing', 'help',
'text', 'analysis']
```

Explanation:

1) `import nltk` and version

```
print("NLTK version:", nltk.__version__)
```

- This prints the installed NLTK version so we know which resources may be required.

---

2) Downloading NLTK resources

```
nltk.download("punkt")
nltk.download("punkt_tab")
nltk.download("stopwords")
nltk.download("wordnet")
nltk.download("omw-1.4")
```

These are important datasets/models used in NLP:

- **punkt**: Used for **tokenization** (splitting text into words/sentences).
- **punkt_tab**: Extra tokenizer tables (some NLTK versions need it, otherwise error comes).
- **stopwords**: Contains common words like *is, for, in, the* etc.
- **wordnet**: Dictionary used for **lemmatization** (base form of words).
- **omw-1.4**: Extra WordNet data that supports lemmatizer properly.

---

3) Tokenization

```
tokens = word_tokenize(text)
```

- Tokenization breaks a sentence into small parts called **tokens**.
- Output includes words and punctuation too:
    - Example tokens include: `NLTK`, `is`, `used`, `.` etc.

---

4) Stopword removal

```
filtered = [w for w in tokens if w.isalpha() and w.lower() not in sw]
```

This line does two things:

1. `w.isalpha()` keeps only **words**, removes punctuation like `.`
2. `w.lower() not in sw` removes **stopwords** like `is`, `for`, `in`, `it`

So after filtering, only meaningful words remain:
`['NLTK', 'Natural', 'Language', 'Processing', 'helps', 'text', 'analysis']`

---

5) Lemmatization

```
lemmatized = [lemmatizer.lemmatize(w.lower()) for w in filtered]
```

Lemmatization converts words to their **base form** (dictionary form).

Example:

- `helps` → `help`
- Others mostly remain same.

Final output:
`['nltk', 'natural', 'language', 'processing', 'help', 'text', 'analysis']`

(39) Write a Python program toImplement stemming concept with using PorterStemmer

Github Link: exercise39.py

```python
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

# Download tokenizer resources (run once)
nltk.download("punkt")
nltk.download("punkt_tab")  # some versions require this

# Step 1: Create Porter Stemmer object
stemmer = PorterStemmer()

# Step 2: Input text
text = "Stemming reduces words like running, runner, and runs to a root
form."

# Step 3: Tokenize text into words
tokens = word_tokenize(text)

# Step 4: Apply stemming (only for alphabet words)
stemmed_words = [stemmer.stem(word) for word in tokens if
word.isalpha()]

print("Original Text:")
print(text)

print("\nTokens:")
print(tokens)

print("\nStemmed Words:")
print(stemmed_words)
```

Output:

```
Sample Output (Example)

Original Text:
Stemming reduces words like running, runner, and runs to a root form.

Tokens:
['Stemming', 'reduces', 'words', 'like', 'running', ',', 'runner', ',',
'and', 'runs', 'to', 'a', 'root', 'form', '.']

Stemmed Words:
['stem', 'reduc', 'word', 'like', 'run', 'runner', 'and', 'run', 'to',
'a', 'root', 'form']
```

Explanation:

What is Stemming?

Stemming is an NLP technique that converts words into a **root form (stem)** by removing suffixes.

Example:

- running → run
- runs → run
- reduced → reduc (may not be a real dictionary word)

So stemming gives a **short form**, not always a meaningful word.

Why PorterStemmer?

PorterStemmer is a popular stemming algorithm in NLTK.
It uses a set of rules to remove word endings.

Step-by-step explanation of program

1. **Download punkt / punkt_tab**
    - Required for `word_tokenize()` to work.

Create stemmer

```
stemmer = PorterStemmer()
```

2.

Tokenize

```
tokens = word_tokenize(text)
```

3.
    - Breaks text into words and punctuation.

Apply stemming

```
stemmer.stem(word)
```

4.
    - Converts each word to its stem.
    - We use `word.isalpha()` so punctuation like `,` and `.` are ignored.

(40) Write a Python program toImplement lemmatization technique to extract the base form of words

Github Link: exercise40.py

```python
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Download required NLTK resources (run once)
nltk.download("punkt")
nltk.download("punkt_tab")   # some versions require this
nltk.download("wordnet")
nltk.download("omw-1.4")

# Step 1: Create Lemmatizer object
lemmatizer = WordNetLemmatizer()

# Step 2: Input text
text = "The children are playing and the cats were running faster than
the mice."

# Step 3: Tokenize text into words
tokens = word_tokenize(text)

# Step 4: Lemmatize only alphabetic words (ignore punctuation)
lemmatized_words = [lemmatizer.lemmatize(word.lower()) for word in
tokens if word.isalpha()]

print("Original Text:")
print(text)

print("\nTokens:")
print(tokens)

print("\nLemmatized Words (Base Form):")
print(lemmatized_words)
```

Output:

```
Sample Output (Example)

Original Text:
The children are playing and the cats were running faster than the mice.

Tokens:
['The', 'children', 'are', 'playing', 'and', 'the', 'cats', 'were',
'running', 'faster', 'than', 'the', 'mice', '.']

Lemmatized Words (Base Form):
['the', 'child', 'are', 'playing', 'and', 'the', 'cat', 'were',
'running', 'faster', 'than', 'the', 'mouse']
```

Explanation:

What is Lemmatization?

Lemmatization is an NLP technique that converts a word into its **base dictionary form (lemma)**.

Examples:

- children → child
- cats → cat
- mice → mouse

Unlike stemming, lemmatization usually returns **real dictionary words**.

Why do we download `wordnet` and `omw-1.4`?

`WordNetLemmatizer` uses the **WordNet dictionary** internally.
So we must download:

- **wordnet** (main dictionary)
- **omw-1.4** (extra WordNet data)

Program steps (easy explanation)

Tokenization

```
tokens = word_tokenize(text)
```

1. Splits sentence into words and punctuation.

Lemmatization

```
lemmatizer.lemmatize(word.lower())
```

2. Converts each word into its base form.

3. **`word.isalpha()`**
   Used to ignore punctuation symbols like `.` and `,`.

Important Note (exam point)

By default, `WordNetLemmatizer` assumes words are **nouns**.
So for verbs like “running”, it may not change unless we pass POS tag like:

```
lemmatizer.lemmatize("running", pos="v")  # gives "run"
```

(41) Write a Python program toCreate a chunk parser

Github Link: exercise41.py

```python
import nltk
from nltk import pos_tag, word_tokenize, RegexpParser

# Downloading necessary resources
nltk.download("punkt")
nltk.download("punkt_tab")
nltk.download("averaged_perceptron_tagger")
nltk.download("averaged_perceptron_tagger_eng")

text = "The quick brown fox jumps over the lazy dog"
tokens = word_tokenize(text)
tagged = pos_tag(tokens)

# Grammar definition for Noun Phrases (NP)
# Removed hidden non-breaking space characters
grammar = "NP: {<DT>?<JJ>*<NN.*>+}"

chunk_parser = RegexpParser(grammar)
tree = chunk_parser.parse(tagged)

print("Sentence:", text)
print("\nTokens:", tokens)
print("\nPOS Tagged Words:")
print(tagged)

print("\nChunk Tree (text form):")
print(tree)

# Optional: Visualize the tree (requires ghostscript installed on your
OS)
# tree.draw()
```

Output:

```
Sentence: The quick brown fox jumps over the lazy dog

Tokens: ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy',
'dog']

POS Tagged Words:
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'),
('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog',
'NN')]

Chunk Tree (text form):
(S
  (NP The/DT quick/JJ brown/NN fox/NN)
  jumps/VBZ
  over/IN
  (NP the/DT lazy/JJ dog/NN))
```

Explanation:

- **Tokenization** breaks sentence into words.
- **POS tagging** gives grammar tags like DT (determiner), JJ (adjective), NN (noun).
- **Chunking** groups related words into phrases.
- Rule `NP: {<DT>?<JJ>*<NN.*>+}` means:
    - optional determiner (DT)
    - any number of adjectives (JJ)
    - one or more nouns (NN/NNS/NNP etc.)
- Output shows two noun phrases:
    - “The quick brown fox”
    - “the lazy dog”

(42) Write a Python program toImplement the structure of sentence

Github Link: exercise42.py

```
import nltk
from nltk import word_tokenize, pos_tag

nltk.download("punkt")
nltk.download("punkt_tab")
nltk.download("averaged_perceptron_tagger")

sentence = "Students are learning Natural Language Processing"
tokens = word_tokenize(sentence)
tagged = pos_tag(tokens)

print("Sentence:", sentence)
print("Tokens:", tokens)
print("POS Tags:", tagged)

print("\nWord -> POS (Sentence Structure):")
for word, tag in tagged:
    print(word, "->", tag)
```

Output:

```
Sentence: Students are learning Natural Language Processing
Tokens: ['Students', 'are', 'learning', 'Natural', 'Language',
'Processing']
POS Tags: [('Students', 'NNS'), ('are', 'VBP'), ('learning', 'VBG'),
          ('Natural', 'JJ'), ('Language', 'NNP'), ('Processing', 'NNP')]

Word -> POS (Sentence Structure):
Students -> NNS
are -> VBP
learning -> VBG
Natural -> JJ
Language -> NNP
Processing -> NNP
```

Explanation:

- This program shows the **grammatical structure** using POS tags.
- Example:
    - **Students (NNS)** = plural noun

- **are (VBP)** = verb
- **learning (VBG)** = verb in “-ing” form
- **Natural (JJ)** = adjective
- **Language/Processing (NNP)** = proper nouns

(43) Write a Python program toEvaluate the grammar using parser

Github Link: exercise43.py

```
import nltk
from nltk import CFG
from nltk.parse import ChartParser

grammar = CFG.fromstring("""
S -> NP VP
NP -> Det N
VP -> V NP
Det -> 'a' | 'the'
N -> 'boy' | 'girl' | 'apple'
V -> 'eats' | 'likes'
""")

parser = ChartParser(grammar)

sentence = "the boy eats a apple".split()

print("Sentence:", " ".join(sentence))
print("\nParse Tree(s):")
found = False
for tree in parser.parse(sentence):
    found = True
    print(tree)

if not found:
    print("No parse tree found (sentence does not match grammar).")
```

Output:

```
Sentence: the boy eats a apple

Parse Tree(s):
(S (NP (Det the) (N boy)) (VP (V eats) (NP (Det a) (N apple))))
```

Explanation:

- Context-Free Grammar (CFG) grammar defines valid sentence patterns.
- The parser checks whether the sentence matches grammar rules.
- If it matches, it prints a **parse tree** (proof that grammar is correct).
- If it does not match, it prints **No parse tree found**.

(44) Write a Python program toGenerate a grammar tree with use of sentence

Github Link: exercise44.py

```
import nltk
from nltk import CFG
from nltk.parse import ChartParser

grammar = CFG.fromstring("""
S -> NP VP
NP -> Det N
VP -> V NP
Det -> 'the' | 'a'
N -> 'cat' | 'mouse'
V -> 'chased' | 'saw'
""")

parser = ChartParser(grammar)
sentence = "the cat chased a mouse".split()

print("Sentence:", " ".join(sentence))
print("\nGrammar Tree(s):")
for tree in parser.parse(sentence):
    print(tree)
```

Output:

```
Sentence: the cat chased a mouse

Grammar Tree(s):
(S (NP (Det the) (N cat)) (VP (V chased) (NP (Det a) (N mouse))))
```

Explanation:

- This creates a parse tree (grammar tree) for the given sentence.
- Tree shows the structure:
    - Sentence (S) = Noun Phrase (NP) + Verb Phrase (VP)
    - NP = Det + N
    - VP = V + NP

(45) Write a Python program toImplement computer vision using OpenCV

Github Link: exercise45.py

```python
import cv2
import os

# 1. Use an absolute path if the image is in a different folder
# Note: Use double backslashes \\ or a forward slash / in Python paths
file_path = "input.jpg"

# Check if file actually exists before trying to read it
if not os.path.exists(file_path):
    print(f"Error: The file '{file_path}' was not found in the directory.")
    print("Current Working Directory:", os.getcwd())
else:
    img = cv2.imread(file_path)

    # 2. Check if the image was successfully loaded
    if img is None:
        print("Error: Could not decode the image. It might be corrupted.")
    else:
        print("Image loaded successfully!")

        # Convert to Grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        print("Original shape (H,W,Channels):", img.shape)
        print("Gray shape (H,W):", gray.shape)

        # Create windows to display
        cv2.imshow("Original", img)
        cv2.imshow("Gray", gray)

        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

Output:

```
Image loaded successfully!
Original shape (H,W,Channels): (600, 800, 3)
Gray shape (H,W): (600, 800)
```

Explanation:

- `imread()` loads the image into a matrix.
- Color image shape has **3 channels** (B,G,R).

- Grayscale has **1 channel** only.
- `imshow()` shows the images in separate windows.

(46) Write a Python program toWork around computer vision relevant python libraries

Github Link: exercise46.py

```python
import cv2
import numpy as np # Import NumPy for numerical array operations

# Load the image into a NumPy array
img = cv2.imread("input.jpg")
print("Loaded:", img is not None)

# Using numpy to get basic statistics from image pixels
# .shape returns (Rows/Height, Columns/Width, Color Channels)
print("Height, Width, Channels:", img.shape)

# .min() and .max() find the lowest and highest brightness values (0-
255)
print("Min pixel value:", img.min())
print("Max pixel value:", img.max())

# .mean() calculates the average brightness across all pixels and
channels
print("Mean pixel value:", round(img.mean(), 2))

# Create a blank image using numpy (black image)
# np.zeros creates an array filled with 0s (black)
# uint8 defines the data type as 8-bit unsigned integers (values 0-255)
blank = np.zeros((300, 300, 3), dtype=np.uint8)

# Draw text on the blank image
# Arguments: (image, text, coordinates, font, scale, color_in_BGR,
thickness)
cv2.putText(blank, "OpenCV", (50, 150), cv2.FONT_HERSHEY_SIMPLEX, 1,
(255,255,255), 2)

# Display the images in separate windows
cv2.imshow("Original", img)
cv2.imshow("Blank with Text", blank)

# Wait for a key press before closing windows
cv2.waitKey(0)

# Clean up memory by closing all GUI windows
cv2.destroyAllWindows()
```

Output:

```
Loaded: True
Height, Width, Channels: (800, 600, 3)
Min pixel value: 0
Max pixel value: 255
Mean pixel value: 123.45
```

Explanation:

- OpenCV reads image as a NumPy array.
- Using NumPy we can calculate:
    - min, max, mean pixel intensity
- We also created a new image using `np.zeros()` and wrote text using `cv2.putText()`.

(47) Write a Python program toUse of imread(), imshow(), and imwrite()

Github Link: exercise47.py

```python
import cv2

# Load the image from the disk
# 'img' becomes a NumPy array representing the pixel data
img = cv2.imread("input.jpg")

# Verify the file was read correctly
# Returns True if the image exists, False if the path is wrong
print("Image loaded:", img is not None)

# Opens a GUI window to visualize the loaded pixel data
cv2.imshow("Original Image", img)

# Save the pixel data currently in 'img' to a new physical file
# imwrite returns a boolean (True/False) indicating if the write was
successful
saved = cv2.imwrite("output_copy.jpg", img)

# Print confirmation of the save operation
print("Image saved (output_copy.jpg):", saved)

# Pause the program execution; window stays open until a key is pressed
cv2.waitKey(0)

# Release all window resources from memory
cv2.destroyAllWindows()
```

Output:

```
Image loaded: True
Image saved (output_copy.jpg): True
```

Explanation:

- `cv2.imread()` reads image from disk.
- `cv2.imshow()` displays it in a window.
- `cv2.imwrite()` saves the image and returns **True** if saved successfully.

(48) Write a Python program toDetect faces from an image using haar-cascade classifier

Github Link: exercise48.py

```
import cv2

# Load the pre-trained Haar Cascade model for frontal face detection
# cv2.data.haarcascades provides the path to the built-in XML model
files
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")

# Read the image file
img = cv2.imread("face.jpg")

# Convert to grayscale because face detection algorithms typically
# process intensity (brightness) rather than color to save computation
time
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Detect faces in the image
# scaleFactor: How much the image size is reduced at each image scale
# minNeighbors: How many neighbors each candidate rectangle should have
to retain it
faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5)

# Output the number of detected face objects found in the 'faces' list
print("Total faces detected:", len(faces))

# Iterate through the list of detected faces
# Each face is represented as (x, y, w, h) - coordinates and size
for (x, y, w, h) in faces:
    # Draw a green rectangle around the face
    # (0, 255, 0) is Green in BGR, and '2' is the line thickness
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

# Display the final result with bounding boxes
cv2.imshow("Face Detection", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

How the Output Appears

When you run this program, here is what will happen:

1. **The Console:** You will see a text output like `Total faces detected: 2` (depending on how many people are in your `face.jpg`).
2. **The GUI Window:** A window titled **"Face Detection"** will pop up.
3. **Visual Markers:** Inside that window, the original photo will appear, but every detected face will have a **bright green square** drawn around it.
4. **Interaction:** The window will remain frozen and visible until you click on the image and **press any key** on your keyboard. Once a key is pressed, the program finishes and the window disappears.

(49) Write a Python program toDetecting different objects from a face such as face, eyes

Github Link: exercise49.py

```python
import cv2

# Load two pre-trained classifiers: one for faces and one for eyes
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
eye_cascade  = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_eye.xml")

# Load the image and convert to grayscale for faster processing
img = cv2.imread("face.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Detect all faces in the image
faces = face_cascade.detectMultiScale(gray, 1.1, 5)
print("Faces:", len(faces))

# Loop through every face found
for (x, y, w, h) in faces:
    # Draw a Green (0, 255, 0) rectangle around the detected face
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # ROI = Region of Interest. We "crop" the face area from the image.
    # We do this in both grayscale (for detection) and color (for
drawing)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]

    # Search for eyes ONLY within the 'roi_gray' (the face area)
    eyes = eye_cascade.detectMultiScale(roi_gray, 1.1, 5)
    print("Eyes in this face:", len(eyes))

    # Loop through every eye found within that specific face
    for (ex, ey, ew, eh) in eyes:
        # Draw a Blue (255, 0, 0) rectangle around the eyes
        # Note: We draw on 'roi_color' so the coordinates are relative
to the face
        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (255, 0, 0),
2)

# Display the final output
cv2.imshow("Face & Eye Detection", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

How the Output Shows

When the window **"Face & Eye Detection"** opens, you will see the following visual results:

1. **Face Bounding Boxes:** Large **Green** rectangles will surround every detected face.
2. **Eye Bounding Boxes:** Smaller **Blue** rectangles will appear inside the green boxes, marking the position of the eyes.
3. **The Console Logic:** The terminal will print the total faces first, then a separate "Eyes in this face" count for every individual face detected.

(50) Write a Python program toDetect a face from a recorded video

Github Link: exercise50.py

```python
import cv2

# Load two pre-trained Haar Cascade classifiers
# One for detecting the face structure and another for eye patterns
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")
eye_cascade  = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_eye.xml")

# Load the image and convert it to grayscale
# Detection algorithms perform better on single-channel grayscale images
img = cv2.imread("face.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Detect faces in the grayscale image
# 1.1 = scaleFactor (image size reduction), 5 = minNeighbors (quality
threshold)
faces = face_cascade.detectMultiScale(gray, 1.1, 5)
print("Faces:", len(faces))

# Loop through every face detected in the image
for (x, y, w, h) in faces:
    # Draw a Green (0, 255, 0) rectangle around the detected face on the
original image
    cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # ROI = Region of Interest. We "crop" the face area specifically.
    # roi_gray is for detection, roi_color is for drawing the eye
rectangles
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]

    # Search for eyes ONLY within the 'roi_gray' (the cropped face box)
    eyes = eye_cascade.detectMultiScale(roi_gray, 1.1, 5)
    print("Eyes in this face:", len(eyes))

    # Loop through every eye found within that specific face ROI
    for (ex, ey, ew, eh) in eyes:
        # Draw a Blue (255, 0, 0) rectangle around the eyes
        # Since we use 'roi_color', the (ex, ey) coordinates are
relative to the face box
        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh), (255, 0, 0),
2)

# Display the final output with all bounding boxes
cv2.imshow("Face & Eye Detection", img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Output:

How the Output Shows

When the window **"Face & Eye Detection"** opens, you will see the following visual results:

1. **Face Bounding Boxes:** Large **Green** rectangles will surround every detected face.
2. **Eye Bounding Boxes:** Smaller **Blue** rectangles will appear inside the green boxes, marking the position of the eyes.
3. **The Console Logic:** The terminal will print the total faces first, then a separate "Eyes in this face" count for every individual face detected.

(51) Write a Python program toDetect a face using live streaming

Github Link: exercise51.py

```python
import cv2

# Load the pre-trained face detection model
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
"haarcascade_frontalface_default.xml")

# Initialize the video capture object
# "video.mp4" is the file path. (Note: use 0 instead of a filename for
live webcam)
cap = cv2.VideoCapture("video.mp4")

# Start an infinite loop to process the video frame by frame
while True:
    # cap.read() returns two values:
    # ret: A boolean (True if frame read successfully, False if
error/end of video)
    # frame: The actual image data for the current frame
    ret, frame = cap.read()

    # Check if the video has ended or if there is a loading error
    if not ret:
        print("Video ended or cannot read video.")
        break

    # Convert the current frame to grayscale for the face detection
algorithm
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Detect faces in the current frame
    faces = face_cascade.detectMultiScale(gray, 1.1, 5)

    # Print the count for every frame processed to the console
    print("Faces detected in frame:", len(faces))

    # Draw a green rectangle around every face found in the current
frame
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)

    # Display the resulting frame with the rectangles drawn on it
    cv2.imshow("Video Face Detection", frame)

    # Wait for 1 millisecond and check if the user pressed the 'q' key
    # This allows the user to manually exit the loop
    if cv2.waitKey(1) & 0xFF == ord("q"):
        break

# Release the video file/camera hardware and close all GUI windows
cap.release()
cv2.destroyAllWindows()
```

Output:

How the Output Shows

When you run this script, you will see a dynamic, moving output:

1. **The Playback:** A window opens showing the video file.
2. **Real-time Tracking:** As people move in the video, the **green boxes** will follow their faces. If someone leaves the frame, the box disappears; if they return, it reappears.
3. **The Console Stream:** Your terminal will scroll rapidly, printing the number of faces detected in every single frame (usually 24 to 60 times per second).
4. **Closing:** The video window will close automatically when the file ends, or immediately if you press the **'q'** key on your keyboard.

Key Video Concepts

- **FPS (Frames Per Second):** The speed of the loop determines the playback speed. `cv2.waitKey(1)` runs as fast as your CPU allows. If the video looks too fast, you can increase the number (e.g., `cv2.waitKey(30)`).
- **The "ret" Variable:** This is the "safety switch." Without checking `if not ret`, the program would crash the moment the video reaches the last frame because it would try to process an empty image.