PRACTICAL NO. 1

AIM:

Introduction to JUPYTER IDE and its libraries Pandas and NumPy

THEORY:

Jupyter IDE Overview

Jupyter Notebook is an open-source interactive computing environment that allows users to create and share documents containing live code, equations, visualizations, and narrative text. It supports multiple programming languages, with Python being the most popular. The Jupyter environment is widely used in data science, machine learning, and academic research due to its user-friendly interface and ability to present outputs alongside code.

Key Features of Jupyter Notebook:

- **Interactive Computing:** Supports real-time code execution and visualization.
- **Multi-Language Support:** Compatible with Python, R, Julia, and more.
- **Rich Media Output:** Allows text, images, plots, and HTML content within the notebook.
- **Integration with Libraries:** Seamlessly integrates with data science libraries such as Pandas, NumPy, and Matplotlib.
- **Notebook Format (.ipynb):** Enables easy sharing and collaboration.

Introduction to NumPy

NumPy (Numerical Python) is a fundamental Python library for numerical computing. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to perform complex computations efficiently. NumPy is highly optimised and serves as the backbone for many other data science libraries.

Key Features of NumPy:

- **N-dimensional Array (ndarray):** Provides fast and efficient array manipulation.
- **Mathematical and Statistical Functions**: Includes operations like mean, median, standard deviation, and linear algebra functions.
- **Broadcasting:** Supports operations on arrays of different shapes without explicit looping.
- **Performance Optimization:** Uses low-level C and Fortran libraries for fast execution.
- **Integration:** Works seamlessly with libraries like Pandas, Matplotlib, and SciPy.

Example of NumPy Usage

```
import numpy as np

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

print(arr * 2)
```

Output

[2 4 6 8 10] (Element-Wise Multiplication)

Introduction to Pandas

Pandas is a powerful data analysis and manipulation library built on top of NumPy. It provides data structures and functions for handling structured data, making it an essential tool for data science and machine learning.

Key Features of Pandas:

- **DataFrame and Series:** Two primary data structures for handling tabular and one-dimensional data.
- **Data Manipulation:** Supports filtering, sorting, merging, and grouping of data.
- **Handling Missing Data:** Provides methods to detect, replace, or remove missing values.
- **Data Input/Output:** Supports reading/writing data from multiple formats such as CSV, Excel, and SQL databases.
- **Integration with Other Libraries:** Works well with NumPy, Matplotlib, and Scikit-learn.

Example of Pandas Usage

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}

df = pd.DataFrame(data)

print(df)
```

Output

```
    Name  Age
0   Alice   25
1     Bob   30
2 Charlie   35
```

ALGORITHM:

Algorithm for Running Code in Jupyter Notebook:

1. **Start Jupyter Notebook:**
   - Open the terminal or command prompt.
   - Run the command:
     jupyter notebook
   - This launches Jupyter in a web browser.
2. **Create/Open a Notebook:**
   - Click "New" to create a notebook (select Python as the kernel).
   - Open an existing .ipynb file if required.
3. **Write Code in a Cell:**
   - Enter Python code in a notebook cell.
4. **Run the Cell:**
   - Click the "Run" button.
   - The kernel executes the code and displays the output below the cell.
5. **Save or Export Notebook:**
   - Save the notebook (Ctrl + S).
   - Export as a .py, .html, or .pdf file if needed.
6. **Shutdown the Notebook:**
   - Close the browser tab.

Algorithm for NumPy Operations:

1. **Import the NumPy Library:**
   - import numpy as np
2. **Create a NumPy Array:**
   - arr = np.array([1, 2, 3, 4, 5])
3. **Perform Mathematical Operations:**
   - Scalar multiplication:
     arr = arr * 2 # Multiply each element by 2
4. **Display or Use the Processed Data:**
   - print(arr)

Algorithm for Pandas Operations

1. **Import the Pandas Library:**
   - import pandas as pd
2. **Create a Pandas DataFrame:**
   - data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35]}
   - df = pd.DataFrame(data)
3. **Display the DataFrame:**
   - print(df)
4. **Perform Mathematical Operations on DataFrame:**
   - Calculate mean age:
     mean_age = df['Age'].mean()
5. **Read/Write Data from CSV or Excel Files:**
   - Read from a CSV file:
     df = pd.read_csv('data.csv')
   - Write to a CSV file:
     df.to_csv('output.csv', index=False)
6. **Display Processed Data:**
   - print(df)

PRACTICAL NO. 2

AIM:

Program to demonstrate Simple Linear Regression.

THEORY:

Simple Linear Regression is a fundamental machine learning algorithm used for predicting continuous values based on a single independent variable. It establishes a relationship between two variables by fitting a straight line through the data points.

Equation of Simple Linear Regression

The mathematical representation of Simple Linear Regression is:

$Y=mX+c$
Where:

- Y = Dependent variable (target/predicted value)
- X = Independent variable (feature)
- m = Slope of the line (coefficient)
- c = Intercept (bias term)

ALGORITHM:

1. **Import necessary libraries:**

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

2. **Load dataset:**

```
df = pd.read_csv('data.csv')  # Example dataset
X = df[['Feature']]  # Independent variable
Y = df['Target']  # Dependent variable
```

3. **Split data into training and testing sets:**
   X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
4. **Train the Linear Regression model:**
   model = LinearRegression()

   model.fit(X_train, Y_train)

5. **Make predictions:**
   Y_pred = model.predict(X_test)
6. **Evaluate the model:**
   mse = mean_squared_error(Y_test, Y_pred)

   r2 = r2_score(Y_test, Y_pred)

   print(f'Mean Squared Error: {mse}')

   print(f'R² Score: {r2}')

7. **Visualize the Regression Line:**
   plt.scatter(X_test, Y_test, color='blue', label='Actual Data')

   plt.plot(X_test, Y_pred, color='red', label='Regression Line')

   plt.xlabel('Feature')

   plt.ylabel('Target')

   plt.legend()

   plt.show()

DATASET DESCRIPTION:

A dataset used for Simple Linear Regression consists of two variables:

1. **Independent Variable (Feature/Input) –** Used to predict the target value.
2. **Dependent Variable (Target/Output) –** The value we aim to predict.

Key Components of the Dataset:

1. **Columns (Features & Target)**
   - Feature (X): The independent variable (e.g., Years of Experience, Temperature, Study Hours).
   - Target (Y): The dependent variable (e.g., Salary, Sales, Exam Score).

2. **Rows (Observations)**
    - Each row represents a single data point (e.g., one employee's experience and salary).
3. **Data Types**
    - Numerical Data: Both X and Y are usually continuous values (integers or floats).

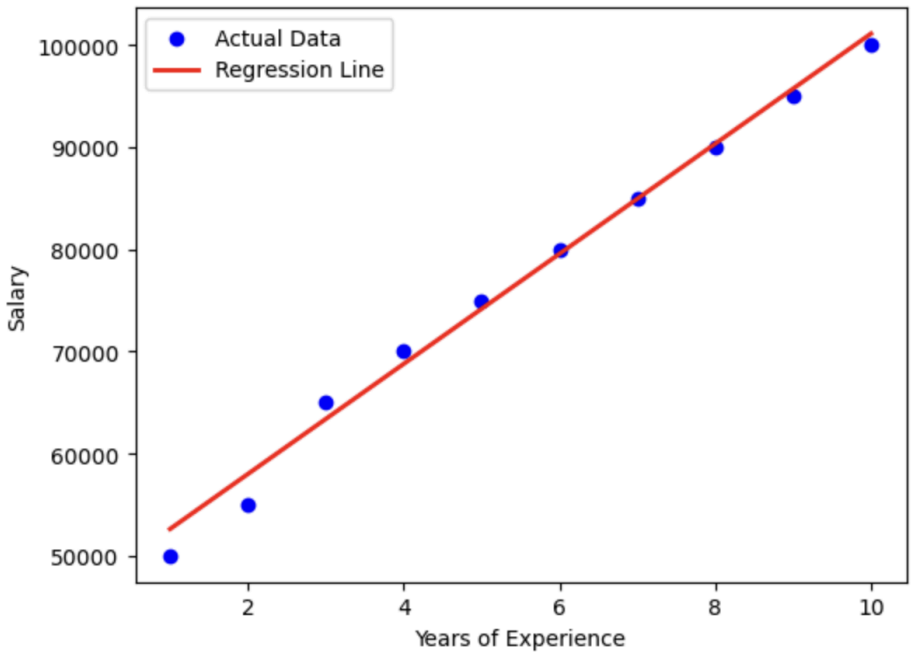
PROGRAMMING CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Prepare Data (Years of Experience vs Salary)
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)  # Feature
y = np.array([50000, 55000, 65000, 70000, 75000, 80000, 85000, 90000, 95000, 100000])  # Target

# Step 2: Split Data into Training and Testing Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

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

# Step 4: Make Predictions
y_pred = model.predict(X_test)

# Step 5: Evaluate the Model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
print(f"R² Score: {r2:.2f}")
print(f"Predicted scores for test data: {y_pred}")

# Step 6: Visualize the Regression Line
plt.scatter(X, y, color='blue', label="Actual Data")
plt.plot(X, model.predict(X), color='red', linewidth=2, label="Regression Line")
plt.xlabel("Years of Experience")
```

```
plt.ylabel("Salary")
plt.legend()
plt.show()

# New test data (unseen values for prediction)
X_new = np.array([[11], [12], [15]])  # Example: Predict salary for 11, 12, and 15 years of experience

# Predict the exam scores for new inputs
y_new_pred = model.predict(X_new)

# Print the predictions
for i, experience in enumerate(X_new.flatten()):
    print(f"Predicted score for {experience} years of salary: {y_new_pred[i]:.3f}")
```

OUTPUT

```
Mean Squared Error: 4820340.37
R² Score: 0.99
Predicted scores for test data: [95732.75862069 58017.24137931]
```




```
Predicted score for 11 years of salary: 106508.621
Predicted score for 12 years of salary: 111896.552
Predicted score for 15 years of salary: 128060.345
```

LEARNING OUTCOMES

PRACTICAL NO. 3

AIM:

Program to demonstrate Logistic Regression

THEORY:

Logistic Regression is a statistical method used for **binary classification problems** where the outcome (dependent variable) is categorical and has two possible values, such as:

- Yes / No
- Pass / Fail
- Spam / Not Spam
- 0 / 1

Unlike **Linear Regression**, which predicts continuous values, Logistic Regression predicts **probabilities** of the output belonging to a particular class.

MATHEMATICAL EQUATION:

The logistic regression model uses the **sigmoid function** (also known as the logistic function) to map predicted values between 0 and 1

$$log\left[\frac{y}{1-y}\right] = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$$

Where:

- b0 is the intercept
- b1,b2,bn are the coefficient or weight
- x1,x2,xn are the independent variable

The output probability is then used to classify data into two categories based on a threshold (usually 0.5).

ALGORITHM:

1. **Initialize Parameters:**
   Initialize the weights w and bias b with random small values.
   w=0,b=0
   These are the parameters the model will learn during training.

2. **Sigmoid Function (Logistic Function):**
   The hypothesis hθ(x) is defined using the sigmoid function. The sigmoid function squashes the linear output between 0 and 1, making it interpretable as a probability:

$$S(x) = \frac{1}{1+e^{-x}}$$

3. **Model Prediction:**
   This output is a probability value between 0 and 1. If the output is greater than 0.5,

predict class 1; otherwise, predict class 0.

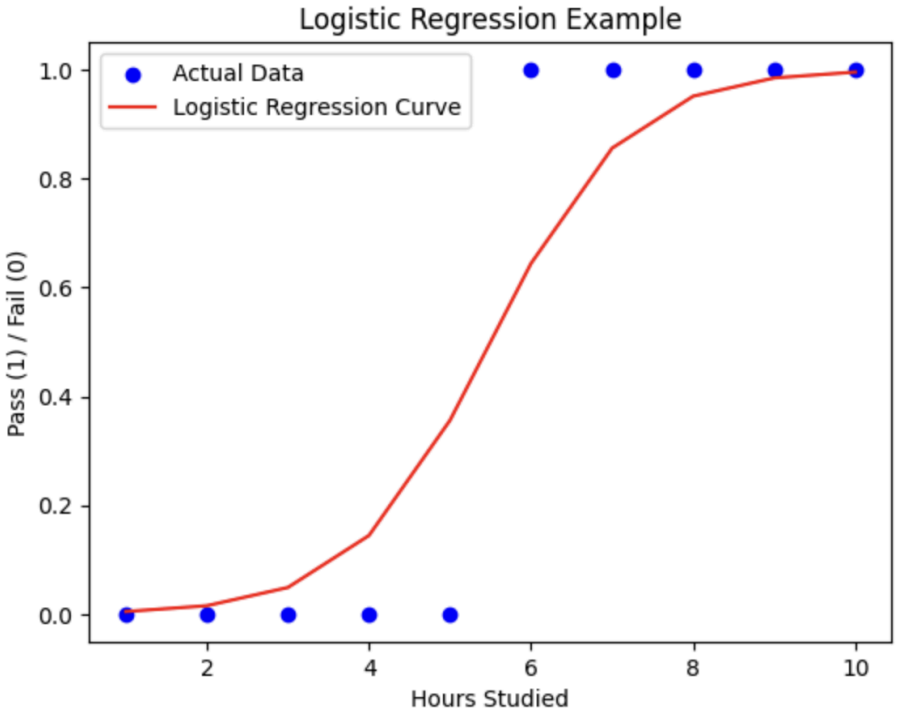
4. **Cost Function (Log-Loss or Binary Cross-Entropy):**
   The cost function measures how well the model's predictions match the actual labels.
5. **Gradient Descent (Optimization):**
   The goal is to minimize the cost function

PROGRAMMING CODE:

```
# Import required libraries
import numpy as np
from sklearn.linear_model import LogisticRegression
import matplotlib.pyplot as plt
# Sample dataset (Independent variable X and Dependent variable Y)
# Example: Hours studied vs Pass (1) or Fail (0)
X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]).reshape(-1, 1)  # Hours studied
Y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])                  # 0 = Fail, 1 = Pass
# Create a Logistic Regression model
model = LogisticRegression()
# Train the model
model.fit(X, Y)
# Predict probabilities
probabilities = model.predict_proba(X)
# Predict output (0 or 1)
predictions = model.predict(X)
# Display the coefficients
print("Coefficient (b1):", model.coef_[0][0])
print("Intercept (b0):", model.intercept_[0])
# Display predictions
print("Predicted classes:", predictions)
# Plotting
plt.scatter(X, Y, color='blue', label='Actual Data')
plt.plot(X, probabilities[:, 1], color='red', label='Logistic Regression Curve')
plt.xlabel('Hours Studied')
plt.ylabel('Pass (1) / Fail (0)')
plt.title('Logistic Regression Example')
plt.legend()
plt.show()
```

OUTPUT

```
Coefficient (b1): 1.1860640568835412
Intercept (b0): -6.52322637938681
Predicted classes: [0 0 0 0 0 1 1 1 1 1]
```



LEARNING OUTCOMES

PRACTICAL NO. 4

AIM:

Program to demonstrate Decision Tree – ID3 Algorithm

THEORY:

ID3 (Iterative Dichotomiser 3) is a classification algorithm used to build Decision Trees. It was introduced by Ross Quinlan and is based on the concept of Information Gain and Entropy from Information Theory.

In Machine Learning, it helps in supervised learning problems, especially for categorical datasets.

ALGORITHM:

1. **Input**:
   - A dataset with features (X) and labels (y)
   - All features must be categorical (if not, discretize them)

2. **Base Case**:
   - If all examples belong to the same class, return a leaf with that class.
   - If no features are left, return a leaf with the **majority class**.

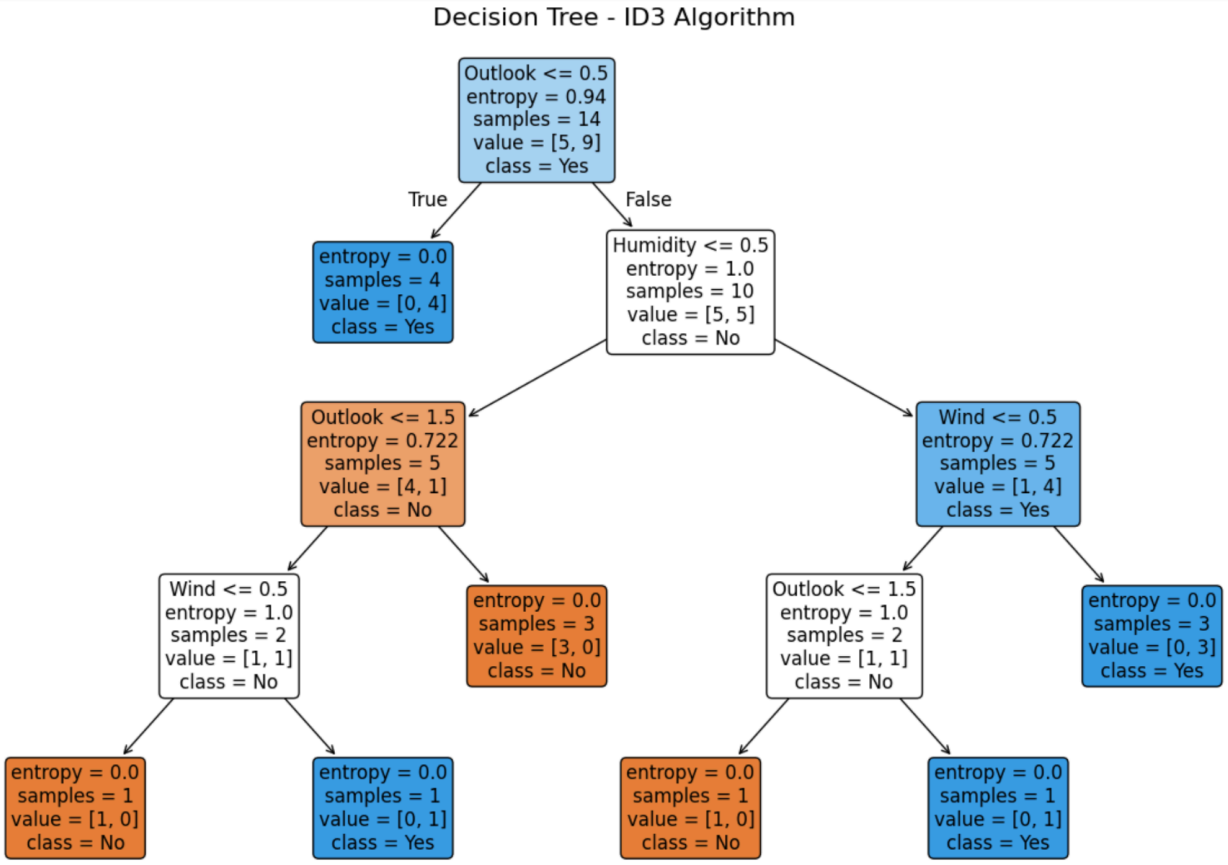
3. **Recursion**:
   - Calculate the **Entropy** of the current dataset.
   - For each feature, calculate **Information Gain**.
   - Choose the feature with the **highest information gain**.
   - Split the dataset on this feature and **recurse** on each subset.

PROGRAMMING CODE:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
import matplotlib.pyplot as plt

# Sample dataset
data = {
    'Outlook': ['Sunny', 'Sunny', 'Overcast', 'Rain', 'Rain', 'Rain', 'Overcast',
                'Sunny', 'Sunny', 'Rain', 'Sunny', 'Overcast', 'Overcast', 'Rain'],
    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool',
                    'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],
    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal',
                 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],
    'Wind': ['Weak', 'Strong', 'Weak', 'Weak', 'Weak', 'Strong', 'Strong',
             'Weak', 'Weak', 'Weak', 'Strong', 'Strong', 'Weak', 'Strong'],
    'PlayTennis': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes',
                   'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']}
```

```
# Create DataFrame
df = pd.DataFrame(data)

# Encode categorical variables
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()

for col in df.columns:
    df[col] = le.fit_transform(df[col])

# Split into features and target
X = df.drop('PlayTennis', axis=1)
y = df['PlayTennis']

# Create and train Decision Tree
clf = DecisionTreeClassifier(criterion="entropy")  # Using ID3 (entropy)
clf.fit(X, y)

# Visualize the Decision Tree
plt.figure(figsize=(16, 10), dpi=100)  # Increase figure size and resolution
tree.plot_tree( clf, feature_names=X.columns,
    class_names=['No', 'Yes'],
    filled=True,
    rounded=True,
    fontsize=12  # Make text smaller to reduce clutter
)
plt.title("Decision Tree - ID3 Algorithm", fontsize=16)
plt.show()
```

OUTPUT



LEARNING OUTCOME

PRACTICAL NO. 5

AIM:

Program to demonstrate k-Nearest Neighbor flowers classification

THEORY:

k-Nearest Neighbors (k-NN) is a supervised classification algorithm that classifies a data point based on how its neighbors are classified. It is instance-based, meaning it doesn't build an explicit model, but instead stores the training instances.

If most of the k nearest neighbors to a point belong to a particular class, the point is classified into that class.

ALGORITHM:

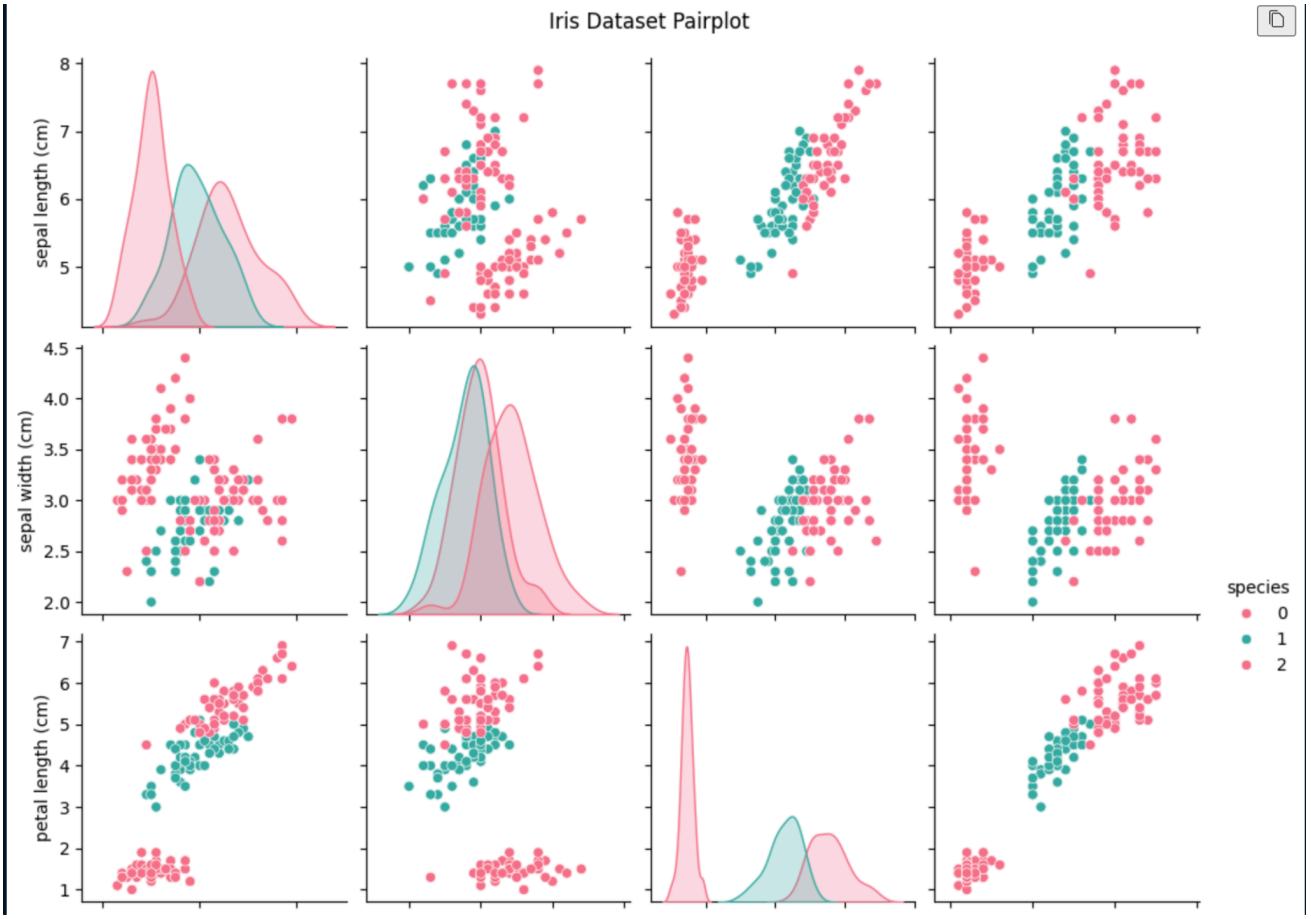
1. Choose the value of k
2. Calculate the distance (e.g., Euclidean) between the test data point and all training data points
3. Sort the distances and pick the top k nearest neighbors
4. Count the number of data points in each class among the k neighbors
5. Assign the class to the test point based on majority vote

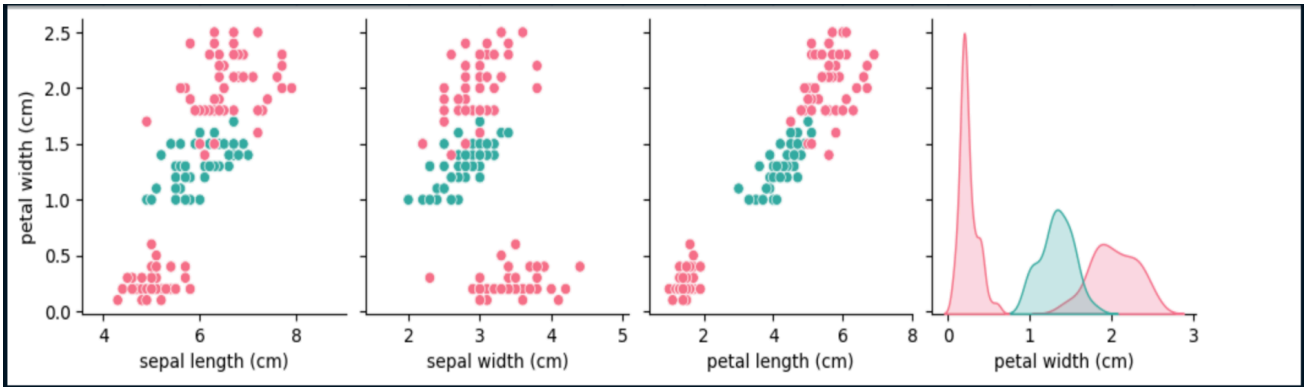
PROGRAMMING CODE:

```
# Step 1: Import required libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Step 2: Load the Iris dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target

# Optional: Visualize with seaborn
sns.pairplot(df, hue='species', palette='husl')
plt.suptitle("Iris Dataset Pairplot", y=1.02)
```

```
plt.show()

# Step 3: Split into features and target
X = df.iloc[:, :-1]   # features
y = df['species']     # target (0=setosa, 1=versicolor, 2=virginica)

# Step 4: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 5: Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 6: Train k-NN model
k = 3
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# Step 7: Make predictions
y_pred = knn.predict(X_test)

# Step 8: Evaluation
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))
print("Accuracy Score:", accuracy_score(y_test, y_pred))

# Step 9: Predict on new flower sample
new_flower = [[5.0, 3.4, 1.5, 0.2]]  # Like a Setosa
new_flower_scaled = scaler.transform(new_flower)
prediction = knn.predict(new_flower_scaled)
print("\nPredicted Species:", iris.target_names[prediction[0]])
```

OUTPUT






```
Confusion Matrix:
 [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30

Accuracy Score: 1.0
```

LEARNING OUTCOME

PRACTICAL NO. 6

AIM:

Program to demonstrate Naïve- Bayes Classifier

THEORY:

What is Naïve Bayes?

- It is a probabilistic classifier based on Bayes' Theorem
- It's called "Naïve" because it assumes that all features are independent of each other.

Why use Naïve Bayes?

- **Fast** and efficient on large datasets
- Works well with **text classification** (spam filtering, sentiment analysis)
- Handles **categorical and numerical** data (with variants like GaussianNB, MultinomialNB, etc.)

ALGORITHM:

Input:

- A training dataset D consisting of feature vectors X = (x_1, x_2, ..., x_n) and their corresponding class labels Y = {C_1, C_2, ..., Ck}
- A new instance X' = (x'_1, x'_2, ..., x'_n) to classify

1. **Calculate Prior Probabilities**
2. **Calculate Likelihood (Conditional Probabilities)**
3. **Calculate Posterior Probabilities**
4. **Select the Class with the Highest Posterior**

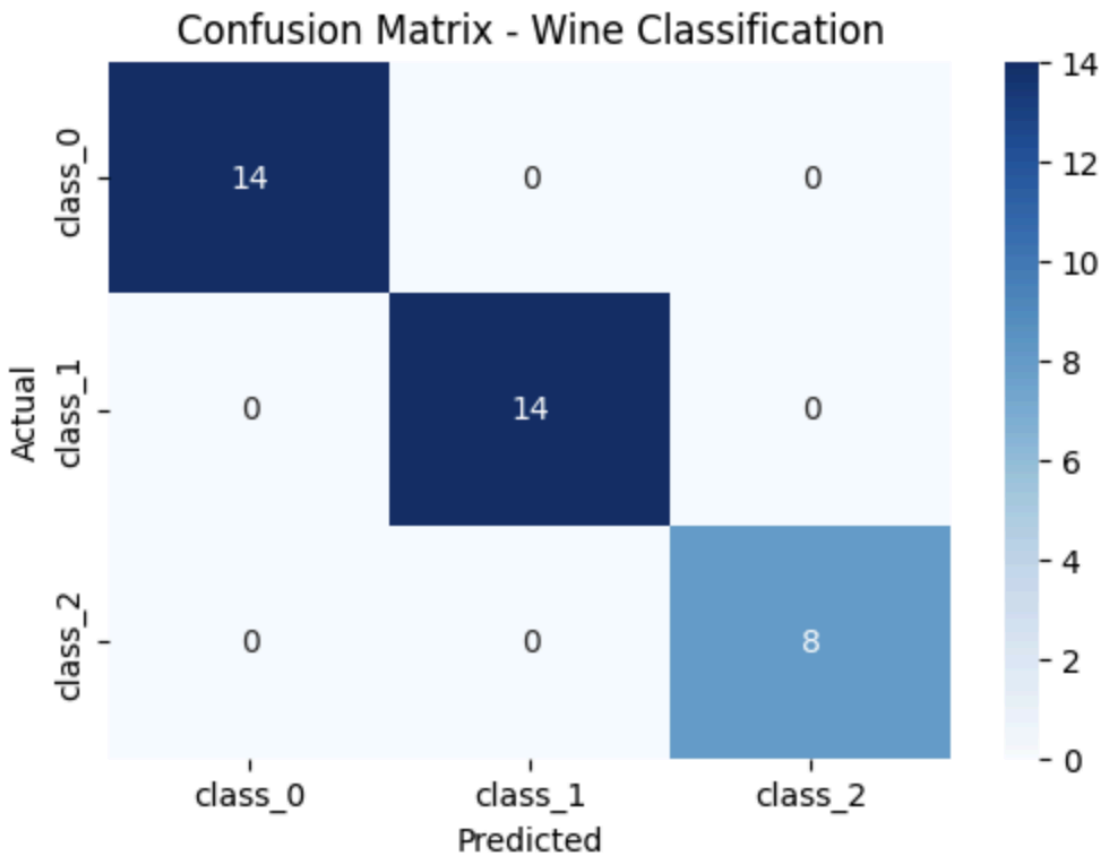
PROGRAMMING CODE:

```
# Step 1: Import Libraries
import pandas as pd
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns

# Step 2: Load Wine Dataset
wine = load_wine()
```

```
X = wine.data
y = wine.target
feature_names = wine.feature_names
target_names = wine.target_names

# Step 3: Create DataFrame for better visualization (optional)
df = pd.DataFrame(X, columns=feature_names)
df['target'] = y

# Step 4: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 5: Create and train Naïve Bayes model
model = GaussianNB()
model.fit(X_train, y_train)

# Step 6: Make predictions
y_pred = model.predict(X_test)

# Step 7: Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=target_names))
print("Accuracy Score:", accuracy_score(y_test, y_pred))

# Step 8: Optional - Visualize Confusion Matrix
plt.figure(figsize=(6,4))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap="Blues",
xticklabels=target_names, yticklabels=target_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - Wine Classification")
plt.show()
```

OUTPUT

```
Confusion Matrix:
 [[14  0  0]
 [ 0 14  0]
 [ 0  0  8]]

Classification Report:
              precision    recall  f1-score   support

     class_0       1.00      1.00      1.00        14
     class_1       1.00      1.00      1.00        14
     class_2       1.00      1.00      1.00         8

    accuracy                           1.00        36
   macro avg       1.00      1.00      1.00        36
weighted avg       1.00      1.00      1.00        36

Accuracy Score: 1.0
```



LEARNING OUTCOME

PRACTICAL NO. 7

AIM:

Program to demonstrate PCA and LDA on Iris Dataset.

THEORY:

Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) are dimensionality reduction techniques used in machine learning and data analysis.

- PCA is an unsupervised technique that transforms data into a lower-dimensional space while preserving as much variance as possible. It helps in feature extraction and visualization.
- LDA is a supervised technique that maximizes class separability, making it useful for classification tasks.

Both techniques reduce the number of features while retaining important information, making models more efficient and reducing computational costs.

ALGORITHM:

PCA Algorithm:

1. Standardize the dataset.
2. Compute the covariance matrix.
3. Compute the eigenvalues and eigenvectors of the covariance matrix.
4. Select the top k eigenvectors corresponding to the k largest eigenvalues.
5. Transform the original dataset into the new k-dimensional space.

LDA Algorithm:

1. Compute the mean vectors for each class.
2. Compute the scatter matrices (within-class and between-class scatter).
3. Compute the eigenvalues and eigenvectors of the scatter matrix.
4. Select the top k eigenvectors corresponding to the largest eigenvalues.
5. Transform the dataset into the new k-dimensional space.

DATASET DESCRIPTION

The Iris dataset is a well-known dataset used for classification tasks. It contains 150 samples of iris flowers from three species:

- Setosa
- Versicolor
- Virginica

Each sample has four features:

1. Sepal Length (cm)
2. Sepal Width (cm)
3. Petal Length (cm)
4. Petal Width (cm)

Target Variable:

- Species (Categorical: Setosa, Versicolor, Virginica)

Dataset Source:

The dataset is originally from Fisher's Iris dataset and is widely used for pattern recognition in Machine Learning.

PROGRAMMING CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
iris = datasets.load_iris()
X = iris.data  # Features
y = iris.target  # Target labels
target_names = iris.target_names
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
pca = PCA(n_components=2)  # Reduce to 2D for visualization
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue']
for i, target_name in enumerate(target_names):
        plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1], label=target_name,   color=colors[i],
alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA on Iris Dataset")
plt.legend()
plt.grid()
plt.show()
lda = LDA(n_components=2)  # Reduce to 2D for visualization
X_lda = lda.fit_transform(X, y)
plt.figure(figsize=(8, 6))
for i, target_name in enumerate(target_names):
        plt.scatter(X_lda[y == i, 0], X_lda[y == i, 1], label=target_name,    color=colors[i],
alpha=0.7)
plt.xlabel("Linear Discriminant 1")
plt.ylabel("Linear Discriminant 2")
plt.title("LDA on Iris Dataset")
plt.legend()
plt.grid()
plt.show()
```

PRACTICAL NO. 8

AIM:

Program to demonstrate DBSCAN algorithm.

THEORY:

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised machine learning algorithm used for clustering data points based on density. Unlike k-Means, which requires a predefined number of clusters, DBSCAN groups data based on high-density regions, making it effective for discovering clusters of arbitrary shape.

Key Characteristics of DBSCAN:

- Can identify clusters of arbitrary shape (unlike k-Means, which assumes spherical clusters).
- Can detect outliers (marked as noise).
- Does not require specifying the number of clusters.

ALGORITHM:

1. Select a point that is not yet classified.
2. Find all neighbors within a distance of ε.
3. If the point has at least MinPts neighbors, mark it as a core point and expand the cluster.
4. If it has fewer than MinPts neighbors, mark it as noise (outlier).
5. Repeat the process for all points until all points are classified.

DATASET DESCRIPTION

The Iris dataset is a well-known dataset used for classification tasks. It contains 150 samples of iris flowers from three species:

- Setosa
- Versicolor
- Virginica

Each sample has four features:

1. Sepal Length (cm)
2. Sepal Width (cm)
3. Petal Length (cm)
4. Petal Width (cm)

Target Variable:

- Species (Categorical: Setosa, Versicolor, Virginica)

Dataset Source:

The dataset is originally from Fisher's Iris dataset and is widely used for pattern recognition in Machine Learning.

PRACTICAL NO. 9

AIM:

Program to demonstrate K-medoid clustering algorithm.

THEORY:

K-Medoid is a clustering algorithm that is similar to K-Means, but instead of using centroids (mean of points), it selects actual data points (medoids) as cluster centers. This makes K-Medoid more robust to outliers compared to K-Means.

Key Characteristics of K-Medoid:

- Uses medoids (actual points) instead of centroids.
- Less sensitive to outliers compared to K-Means.
- Finds clusters by minimizing the sum of distances between points and their medoid.

ALGORITHM:

1. Initialize k medoids randomly from the dataset.
2. Assign each data point to the nearest medoid.
3. Swap medoids with other points in the cluster to find a better set of medoids (i.e., those that minimize total distance).
4. Repeat steps 2-3 until medoids do not change or convergence is reached.

DATASET DESCRIPTION

The Iris dataset is a well-known dataset used for classification tasks. It contains 150 samples of iris flowers from three species:

- Setosa
- Versicolor
- Virginica

Each sample has four features:

1. Sepal Length (cm)
2. Sepal Width (cm)
3. Petal Length (cm)
4. Petal Width (cm)

Target Variable:

• Species (Categorical: Setosa, Versicolor, Virginica)

Dataset Source:

The dataset is originally from Fisher's Iris dataset and is widely used for pattern recognition in Machine Learning.

PROGRAMMING CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from pyclustering.cluster.kmedoids import kmedoids
from scipy.spatial.distance import cdist
iris = datasets.load_iris()
X = iris.data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
np.random.seed(42)
initial_medoids = np.random.choice(len(X_scaled), 3, replace=False)
kmedoids_instance = kmedoids(X_scaled, initial_medoids, data_type='distance_matrix')
kmedoids_instance.process()
clusters = kmedoids_instance.get_clusters()
medoid_indices = kmedoids_instance.get_medoids()
cluster_labels = np.zeros(len(X_scaled))
for i, cluster in enumerate(clusters):
    for index in cluster:
        cluster_labels[index] = i
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue']
for i in range(3):
    plt.scatter(X_pca[np.array(clusters[i]), 0], X_pca[np.array(clusters[i]), 1],
                label=f'Cluster {i}', color=colors[i], alpha=0.7)
plt.scatter(X_pca[medoid_indices, 0], X_pca[medoid_indices, 1],
            color='black', marker='X', s=200, label='Medoids')
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("K-Medoid Clustering on Iris Dataset (pyclustering)")
plt.legend()
plt.grid()
plt.show()
```

PRACTICAL NO. 10

AIM:

Program to demonstrate K-Means Clustering Algorithm on handwritten dataset.

THEORY:

Clustering is an unsupervised learning technique that groups similar data points together based on certain features. Unlike supervised learning, where data is labeled, clustering algorithms find intrinsic patterns in the data without predefined categories.
One of the most widely used clustering techniques is the K-Means Algorithm.
Advantages of K-Means:

- Simple & Efficient – Works well on large datasets.
- Scalable – Can handle high-dimensional data.
- Faster convergence compared to hierarchical clustering.

Disadvantages of K-Means:

- Requires the number of clusters (K) to be specified beforehand.
- Sensitive to initial centroid selection – Poor initialization can lead to bad clustering.
- Works best with spherical clusters – Struggles with complex cluster shapes.

ALGORITHM:

1. Select the number of clusters K.
2. Randomly initialize K cluster centroids.
3. Assign each data point to the nearest centroid (based on Euclidean distance).
4. Compute the new centroid for each cluster as the mean of all assigned points.
5. Repeat steps 3 & 4 until centroids no longer change significantly (convergence).

DATASET DESCRIPTION

We use the Digits Dataset from sklearn.datasets. This dataset contains images of handwritten digits (0-9), each represented as a 64-dimensional vector (8×8 pixel values).

Dataset Details:

Features: 64 (Each 8×8 image is flattened into a 1D vector).
Samples: 1797 handwritten digit images.
Labels: 10 classes (digits 0-9).
Data Type: NumPy array.

Each image is grayscale, and pixel intensity values range from 0 (black) to 16 (white).

PRACTICAL NO. 11

AIM:

To demonstrate Time Series Forecasting using the ARIMA (AutoRegressive Integrated Moving Average) model.

THEORY:

Time Series Forecasting is a statistical technique used to predict future values based on previously observed values over time. It is used in finance, economics, weather prediction, and many other fields.

ARIMA stands for:

AR – AutoRegressive: Model uses the dependency between an observation and a number of lagged observations (previous values).

I – Integrated: Involves differencing of raw observations to make the time series stationary.

MA – Moving Average: Model uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

The parameters of the ARIMA model are:

- p (AR): Number of lag observations included.
- d (I): Number of times the raw observations are differentiated.
- q (MA): Size of the moving average window.

Applications of ARIMA:

- Stock Market Prediction
- Inventory & Demand Forecasting
- Weather Forecasting
- Sales and Revenue Estimation

ALGORITHM:

1. Load the time series dataset.
2. Visualize the time series to understand its trend and seasonality.
3. Apply transformations (if needed) to make the data stationary.
4. Fit an ARIMA(p, d, q) model to the data.
5. Forecast future values.
6. Plot the forecast and evaluate model performance.

DATASET DESCRIPTION

For this practical, we use a synthetic monthly sales dataset or a real-world dataset such as Airline Passenger Data.

Feature: Number of passengers or monthly sales
Time Frame: Monthly data over several years
Frequency: Monthly
Size: 144 data points (12 months × 12 years)

PROGRAMMING CODE:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from sklearn.metrics import mean_squared_error
url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv'
data = pd.read_csv(url, index_col='Month', parse_dates=True)
plt.figure(figsize=(10, 5))
plt.plot(data, label='Original Series')
plt.title('Monthly Airline Passenger Numbers')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.grid()
plt.legend()
plt.show()
data_diff = data.diff().dropna()
result = adfuller(data_diff['Passengers'])
print(f"ADF Statistic: {result[0]}")
print(f"p-value: {result[1]}")
model = ARIMA(data, order=(2,1,2))  # ARIMA(p,d,q)
model_fit = model.fit()
print(model_fit.summary())
forecast = model_fit.forecast(steps=12)
forecast_index = pd.date_range(start=data.index[-1], periods=12, freq='MS')
plt.figure(figsize=(10, 5))
plt.plot(data, label='Training Data')
plt.plot(forecast_index, forecast, label='Forecast', color='red')
plt.title('Forecast of Airline Passengers using ARIMA')
plt.xlabel('Date')
plt.ylabel('Passengers')
plt.grid()plt.legend()
plt.show()
```

PRACTICAL NO. 12

AIM:

Program to demonstrate Multi-class Classification using XGBoost Classifier

THEORY:

Multi-class classification is a type of supervised learning where the target variable has more than two classes (e.g., classifying flowers into Setosa, Versicolor, and Virginica). The goal is to predict which category a given input belongs to.

XGBoost (Extreme Gradient Boosting) is an advanced machine learning algorithm based on gradient boosting decision trees. It is known for its speed, performance, and accuracy in classification and regression tasks.

XGBoost builds an ensemble of weak learners (decision trees) in a stage-wise manner and uses gradient descent to minimize loss.
XGBoost is good for multi-class classification as it:

- Handles multi-class targets natively using the "multi: softprob" or "multi: softmax" objectives.
- Offers regularization to prevent overfitting.
- Handles missing values automatically.
- Supports parallel computation.

ALGORITHM:

1. Load the dataset and preprocess it.
2. Encode the class labels numerically.
3. Split the dataset into training and test sets.
4. Initialize and train the XGBoost Classifier with objective "multi: softmax".
5. Predict labels on the test set.
6. Evaluate the model using accuracy and classification metrics.
7. Visualize results using a confusion matrix.

DATASET DESCRIPTION

The Iris dataset is a well-known dataset used for classification tasks. It contains 150 samples of iris flowers from three species:

- Setosa
- Versicolor
- Virginica

Each sample has four features:

1. Sepal Length (cm)
2. Sepal Width (cm)
3. Petal Length (cm)
4. Petal Width (cm)

Target Variable:

• Species (Categorical: Setosa, Versicolor, Virginica)

Dataset Source:

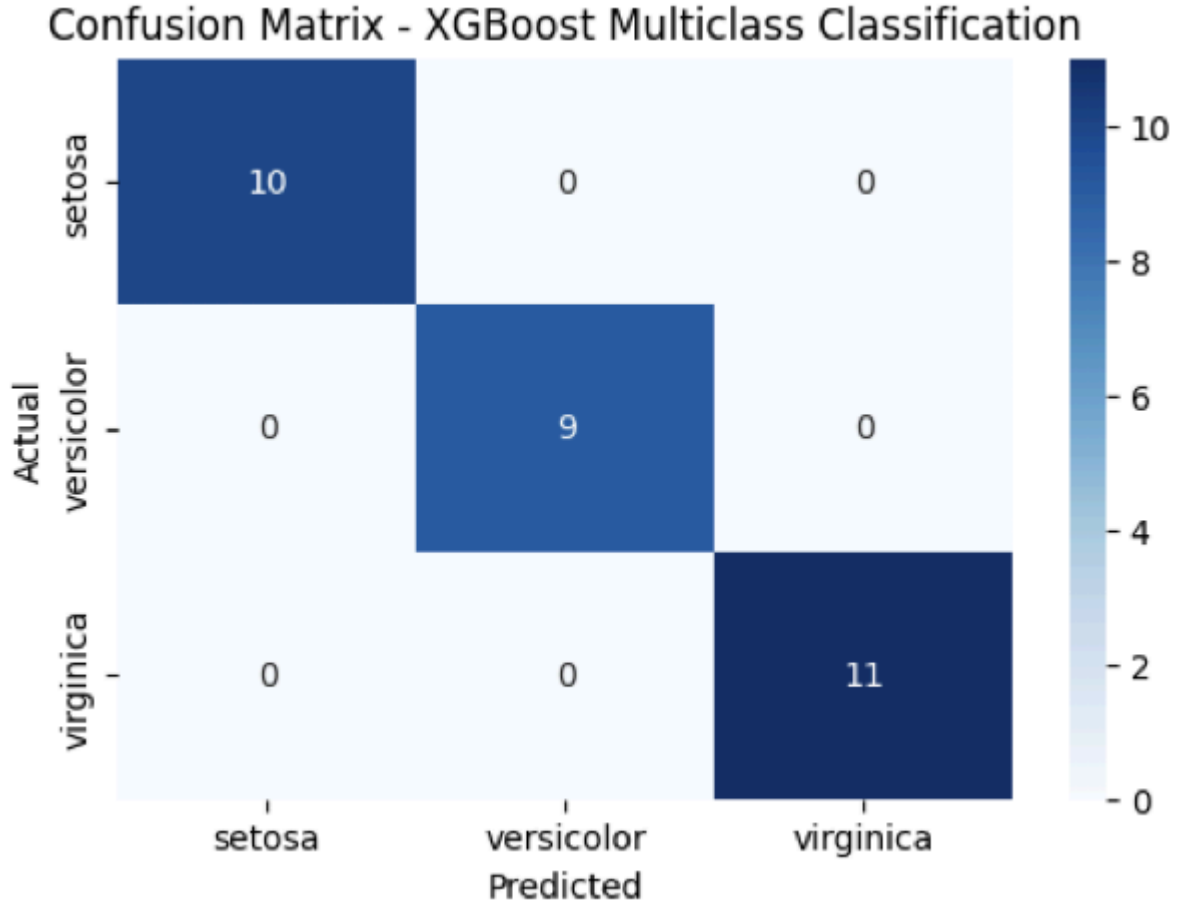
The dataset is originally from Fisher's Iris dataset and is widely used for pattern recognition in Machine Learning.

PROGRAMMING CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import xgboost as xgb
iris = load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = xgb.XGBClassifier(objective='multi:softmax', num_class=3, eval_metric='mlogloss',
use_label_encoder=False)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, cmap='Blues', fmt='d',
        xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix - XGBoost Multiclass Classification")
plt.show()
```

OUTPUT

```
Accuracy: 1.00

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```



LEARNING OUTCOME

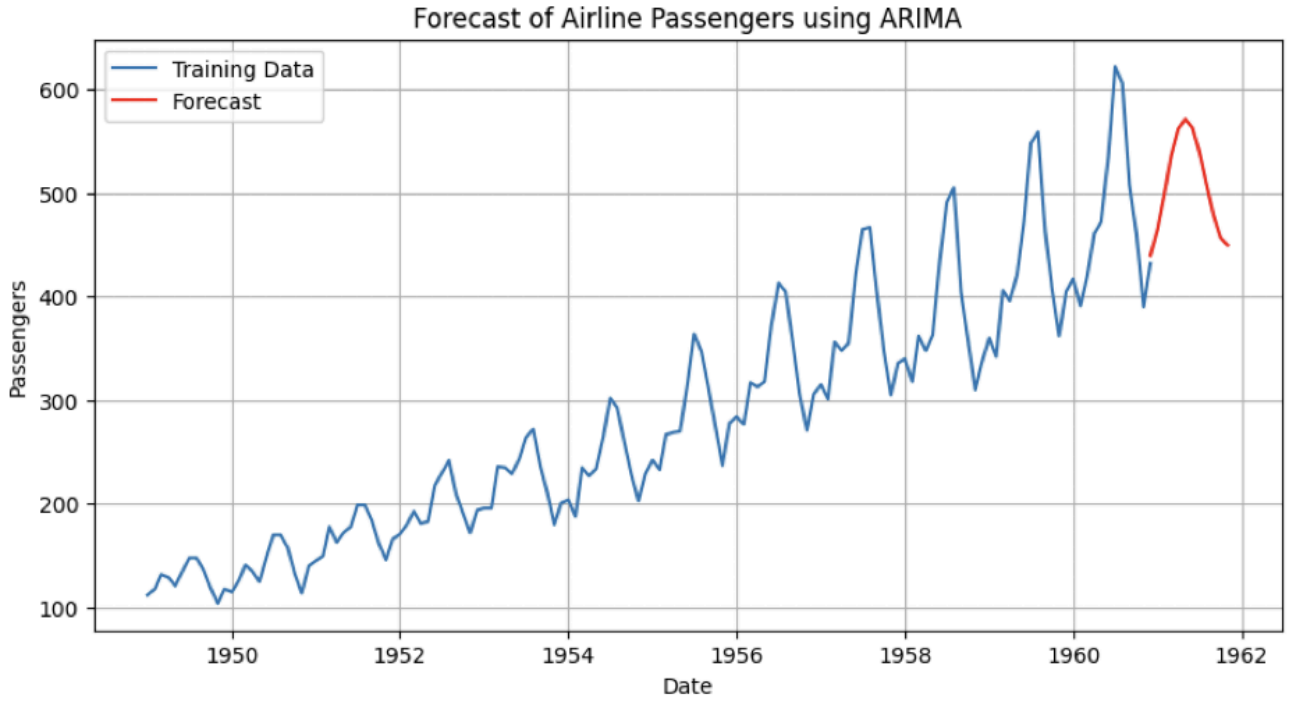
OUTPUT

SARIMAX Results

Dep. Variable:	Passengers	No. Observations:	144
Model:	ARIMA(2, 1, 2)	Log Likelihood	-671.673
Date:	Wed, 16 Apr 2025	AIC	1353.347
Time:	19:35:11	BIC	1368.161
Sample:	01-01-1949	HQIC	1359.366
	- 12-01-1960		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	1.6850	0.020	83.059	0.000	1.645	1.725
ar.L2	-0.9548	0.017	-55.419	0.000	-0.989	-0.921
ma.L1	-1.8432	0.125	-14.798	0.000	-2.087	-1.599
ma.L2	0.9953	0.135	7.374	0.000	0.731	1.260
sigma2	665.9568	114.104	5.836	0.000	442.316	889.597

Ljung-Box (L1) (Q):	0.30	Jarque-Bera (JB):	1.84
Prob(Q):	0.59	Prob(JB):	0.40
Heteroskedasticity (H):	7.38	Skew:	0.27
Prob(H) (two-sided):	0.00	Kurtosis:	3.14

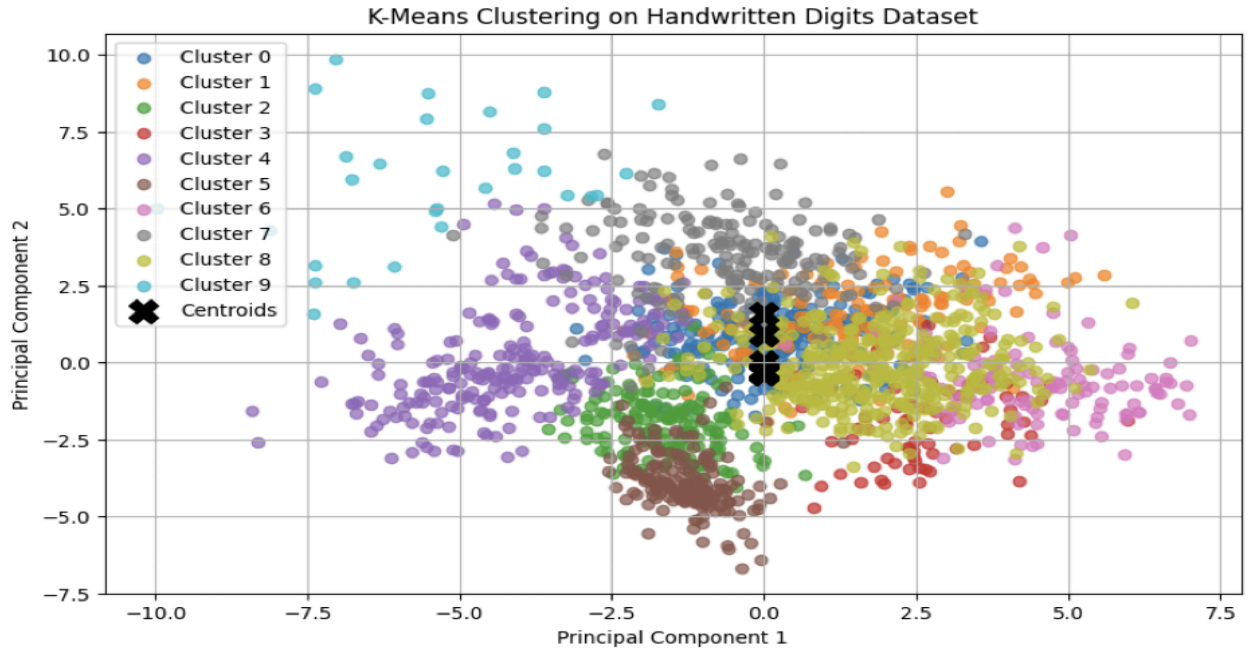


LEARNING OUTCOME

PROGRAMMING CODE:

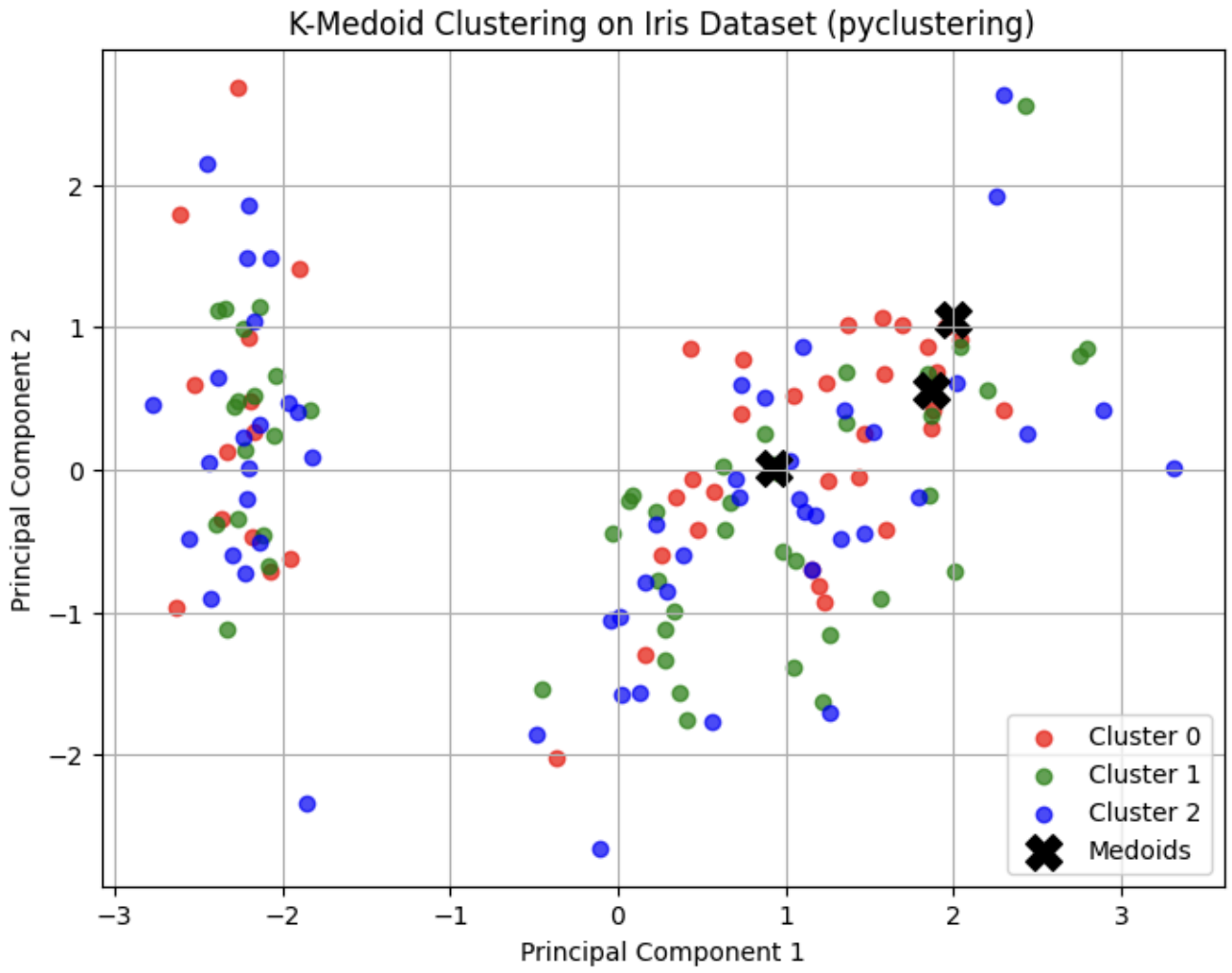
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
digits = datasets.load_digits()
X = digits.data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
k = 10
kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
labels = kmeans.fit_predict(X_scaled)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(10, 6))
for i in range(k):
    plt.scatter(X_pca[labels == i, 0], X_pca[labels == i, 1], label=f'Cluster {i}', alpha=0.7)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            color='black', marker='X', s=200, label='Centroids')
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("K-Means Clustering on Handwritten Digits Dataset")
plt.legend()
plt.grid()
plt.show()
```

OUTPUT



LEARNING OUTCOME

OUTPUT

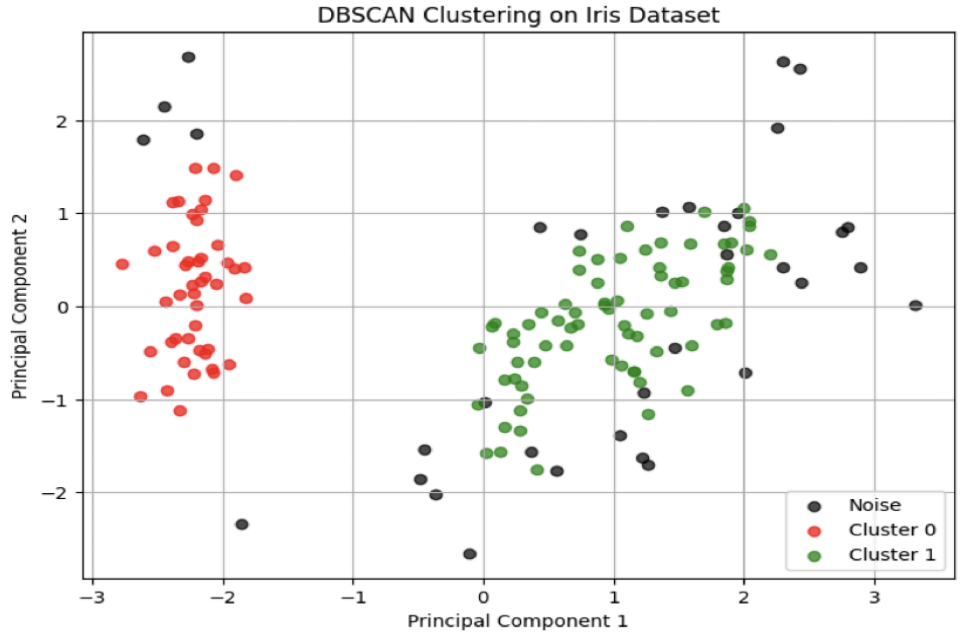


LEARNING OUTCOME

PROGRAMMING CODE:

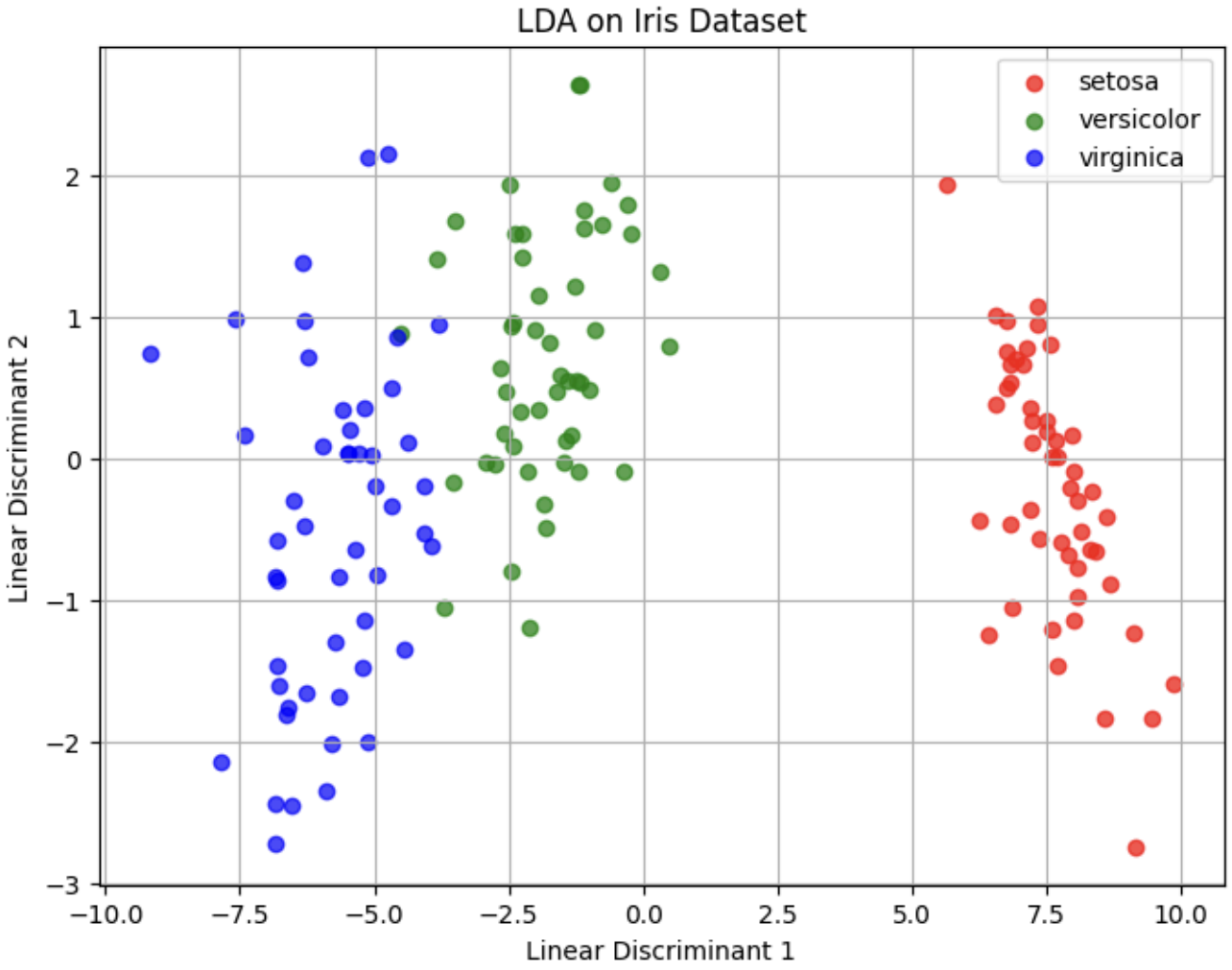
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
iris = datasets.load_iris()
X = iris.data
target_names = iris.target_names
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X_scaled)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(8, 6))
unique_clusters = np.unique(clusters)
colors = ['red', 'green', 'blue', 'black']
for cluster in unique_clusters:
    plt.scatter(X_pca[clusters == cluster, 0], X_pca[clusters == cluster, 1],
                label=f'Cluster {cluster}' if cluster != -1 else 'Noise',
                color=colors[cluster] if cluster != -1 else 'black', alpha=0.7)
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("DBSCAN Clustering on Iris Dataset")
plt.legend()
plt.grid()
plt.show()
```

OUTPUT



LEARNING OUTCOME

OUTPUT



LEARNING OUTCOME