Experiment-1

Aim: Introduction to JUPYTER IDE and it's libraries Pandas and Numpy

Introduction to Jupyter IDE:
Jupyter Notebook is an open-source, web-based interactive computing platform that allows users to create and share documents containing live code, equations, visualizations, and narrative text. It supports a variety of programming languages, including Python, R, and Julia.

Features of Jupyter Notebook:

1) Interactive Interface: Users can write and execute code in cells, making it easy to test small code snippets.
2) Rich Output: Supports inline visualization, making it ideal for data analysis and plotting.
3) Markdown Support: Allows users to write descriptive documentation using markdown alongside code.
4) Sharing: Notebooks can be exported as .ipynb files, PDFs, or HTML for easy sharing.

Jupyter is widely used for data science, machine learning, and academic research due to its flexibility and ease of use.

Introduction to Pandas:
Pandas is a Python library used for data manipulation and analysis. It offers data structures like Series (1D data) and DataFrame (2D data) to handle labeled data efficiently. Pandas is commonly used for tasks such as:

- Data cleaning
- Data transformation
- Aggregating and grouping data
- Merging and joining datasets
- Handling missing data

Key Pandas Features:

1) DataFrame: 2-dimensional labeled data structure with columns of potentially different types.
2) Series: 1-dimensional labeled array capable of holding any data type.
3) Indexing and Slicing: Accessing parts of the data using labels or positions.
4) Group By: Splitting data into groups based on some criteria.
5) Data Cleaning: Handling missing data or transforming columns for analysis.

Example of a Pandas DataFrame:

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
```

```
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)
print(df)
```

Introduction to NumPy:

NumPy (Numerical Python) is a fundamental package for scientific computing in Python. It provides support for arrays (multidimensional data), mathematical functions, and linear algebra operations.

Key NumPy Features:

1) ndarray: A powerful N-dimensional array object for fast and efficient data manipulation.
2) Vectorized Operations: Enables the performance of element-wise operations on arrays, avoiding the need for loops.
3) Mathematical Functions: Provides functions for mathematical operations like trigonometry, statistics, etc.
4) Linear Algebra: Offers support for operations such as matrix multiplication, eigenvalues, and more.

Example of NumPy Arrays:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
matrix = np.array([[1, 2], [3, 4], [5, 6]])
print(matrix)
arr2 = arr + 10
print(arr2)
```

Experiment-2

Aim: Program to implement simple linear regression

Simple Linear Regression is a fundamental type of regression used to predict the value of a **dependent variable (y)** based on the value of an **independent variable (x)**. The relationship between the two variables is assumed to be **linear**, meaning the change in yyy is proportional to the change in xxx.

The goal of simple linear regression is to find the **best-fitting straight line** through the data points, known as the **regression line**, that minimizes the differences (errors) between the observed values and the predicted values.

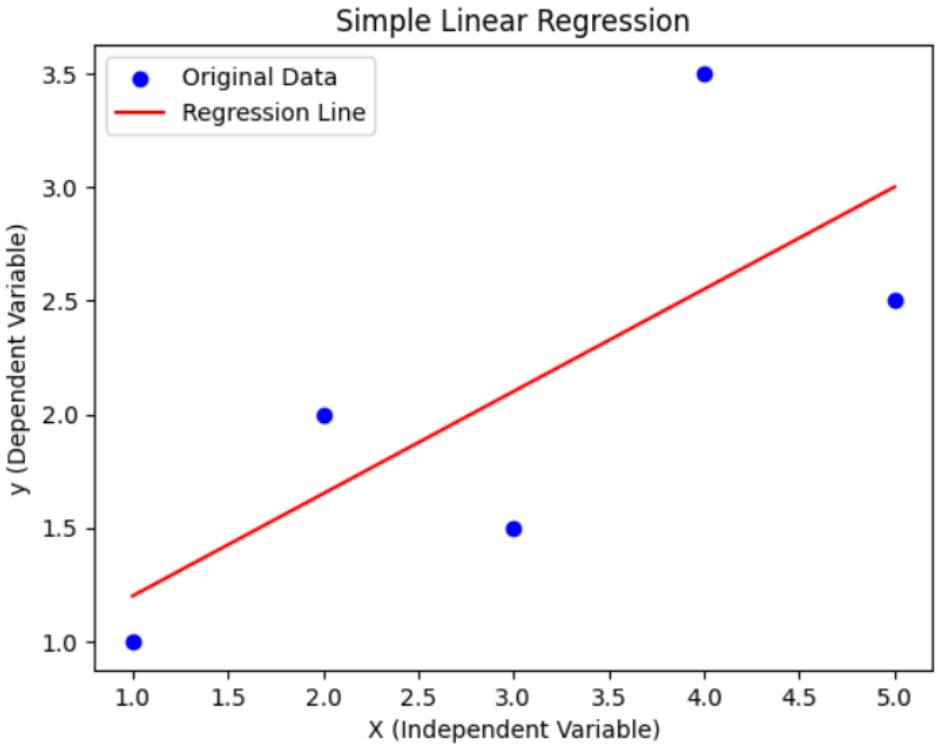
1. **Linear Relationship**:
    - Simple linear regression assumes a **linear relationship** between the dependent and independent variables. The model predicts y based on a straight-line equation: y=mx+cy where:
        - y is the predicted dependent variable,
        - xxx is the independent variable,
        - m (slope) represents how much y changes when xxx increases by 1 unit,
        - c (intercept) is the value of y when x is 0.
2. **Best Fit Line**:
    - The best fit line, or regression line, is the line that minimizes the **sum of squared differences** between the observed data points and the predicted points (errors). This technique is known as **Ordinary Least Squares (OLS)**.
3. **Slope and Intercept**:
    - The **slope (m)** represents the rate of change of the dependent variable with respect to the independent variable. A positive slope means y increases as xxx increases, while a negative slope means yyy decreases as xxx increases.
    - The **intercept (c)** is the point where the regression line crosses the y-axis (i.e., the value of y when x is 0).
4. **Residuals (Errors)**:
    - **Residuals** are the differences between the actual values of y and the predicted values of y by the model. These are the vertical distances from the data points to the regression line.
    - The goal of linear regression is to minimize these residuals to ensure that the predicted values are as close as possible to the actual values.

Code:

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

```
def simple_linear_regression(X, y):
    # Calculate the mean of X and y
    mean_X = np.mean(X)
    mean_y = np.mean(y)

    # Calculate the slope (m) and intercept (b)
    numerator = sum((X - mean_X) * (y - mean_y))
    denominator = sum((X - mean_X) ** 2)
    m = numerator / denominator
    b = mean_y - m * mean_X

    return m, b

# Sample data (X is independent variable, y is dependent variable)
X = np.array([1, 2, 3, 4, 5])
y = np.array([1, 2, 1.5, 3.5, 2.5])

m, b = simple_linear_regression(X, y)
y_pred = m * X + b

plt.scatter(X, y, color='blue', label='Original Data')
plt.plot(X, y_pred, color='red', label='Regression Line')
plt.xlabel('X (Independent Variable)')
plt.ylabel('y (Dependent Variable)')
plt.title('Simple Linear Regression')
plt.legend()
plt.show()

print(f"Slope (m): {m}")
print(f"Intercept (b): {b}")
```

Output:




```
Slope (m): 0.45
Intercept (b): 0.75
```

Experiment-3

Aim: Program to Demonstrate logistic regression

Logistic regression is a **supervised learning algorithm** primarily used for **binary classification** tasks, where the goal is to predict one of two possible outcomes (such as yes/no, 0/1, etc.). It estimates the probability that a given input belongs to a particular class by applying a transformation to a linear combination of input features. Although it's called "regression," it's used for classification tasks.

Important Points of a Program to Demonstrate Logistic Regression:

1. **Data Preprocessing**:
   - Load a dataset that contains the features and target variable (labels) for classification.
   - Handle missing data, if any, and encode categorical variables into numeric form if necessary.
   - Split the data into **training** and **testing** sets to evaluate the model performance.
2. **Model Training**:
   - Import and use the logistic regression algorithm from a machine learning library (like Scikit-learn).
   - Fit the logistic regression model to the training data, where it will learn the relationship between the features and the target variable.
3. **Prediction**:
   - Use the trained model to predict the target values (classes) for the testing data.
   - The model will output probabilities for each data point, and those probabilities will be used to classify the points into one of the two classes.
4. **Model Evaluation**:
   - Evaluate the model's performance on the testing data using appropriate metrics like **accuracy**, **precision**, **recall**, **F1-score**, and **confusion matrix**.
   - Optionally, plot the **ROC curve** and calculate the **AUC** to assess the quality of the probability predictions.
5. **Interpreting Results**:
   - Analyze the performance of the model to determine how well it classifies the data. You can also interpret the learned coefficients to understand the relationship between features and the outcome.

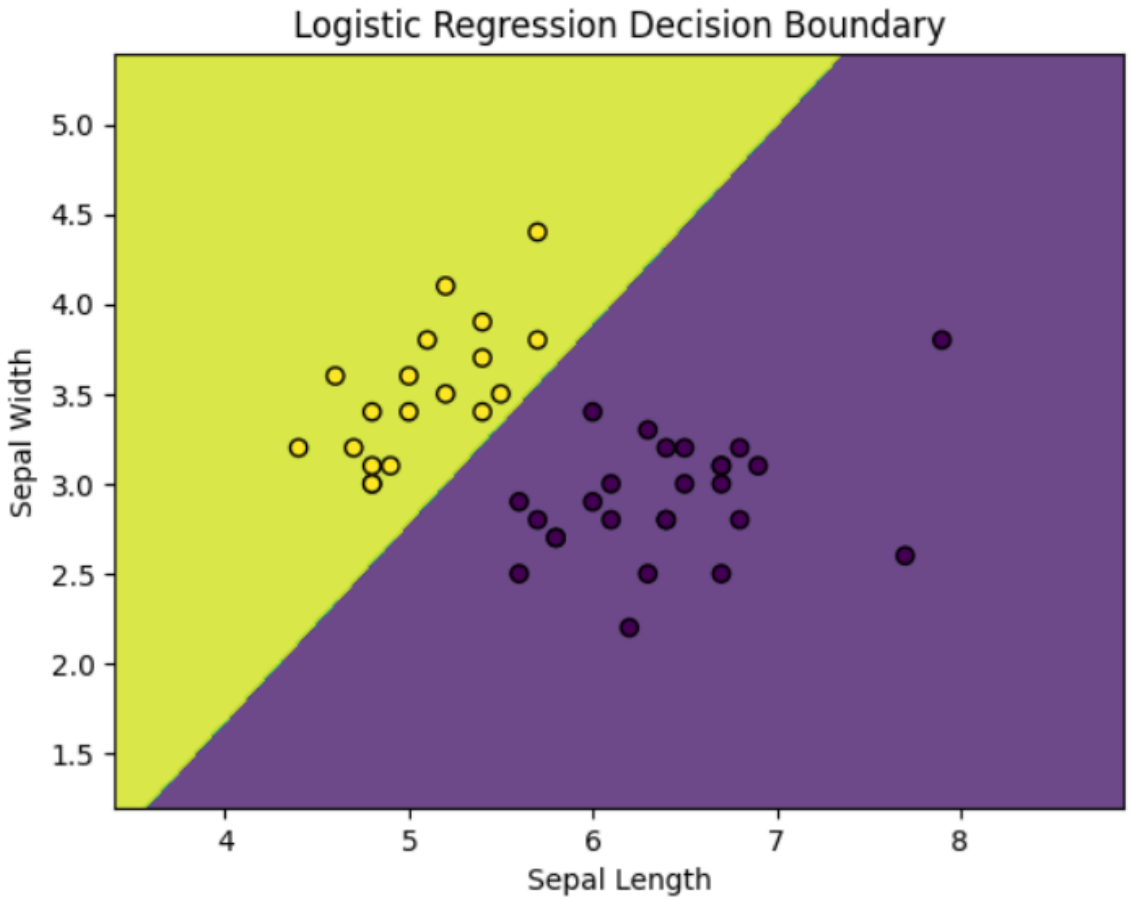
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the Iris dataset from sklearn
# Dataset link: https://archive.ics.uci.edu/ml/datasets/iris
iris = datasets.load_iris()

X = iris.data[:, :2]
y = (iris.target == 0).astype(int)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")

# Plotting the decision boundary and data points
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)


    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
    plt.xlabel('Sepal Length')
    plt.ylabel('Sepal Width')
    plt.title('Logistic Regression Decision Boundary')
    plt.show()

plot_decision_boundary(X_test, y_test, model)
```

Output:

```
Model Accuracy: 100.00%
```



Experiment-4

Aim: Program to Demonstrate Decision tree-ID3 Algorithm

Decision Tree is a supervised learning algorithm used for both classification and regression tasks. The **ID3 (Iterative Dichotomiser 3)** algorithm is a popular decision tree algorithm used for **classification** problems. It builds a decision tree based on the concept of **information gain** and selects the attribute that maximizes the gain to split the data at each step.

1. **Decision Tree Structure**:
    - A decision tree consists of **nodes** and **branches**:
        - **Root Node**: The top-most node that represents the entire dataset.
        - **Internal Nodes**: Represent the features and the decision criteria for splitting the data.
        - **Leaf Nodes**: Represent the final decision or class label.
2. **Recursive Partitioning**:
    - The tree is built recursively by splitting the dataset into smaller subsets based on the feature that provides the **best split**.
    - The process continues until one of the stopping criteria is met (e.g., all instances belong to a single class, or no further improvement can be made).
3. **Attribute Selection with ID3**:
    - ID3 uses the **information gain** to decide which feature to split the data on at each step. Information gain measures the reduction in entropy (uncertainty) after a dataset is split on a feature.
4. **Entropy**:
    - **Entropy** measures the impurity or uncertainty in the dataset. A higher entropy value means the dataset is more mixed (uncertain), and a lower entropy means the dataset is more homogenous.
    - The goal is to minimize entropy after each split.
5. **Information Gain**:
    - Information gain is calculated for each feature as the difference between the entropy of the dataset before and after the split. The feature with the highest information gain is selected for the split.
6. **Stopping Criteria**:
    - The algorithm stops when:
        - All data points belong to the same class.
        - There are no more features to split.
        - A predefined maximum depth is reached.
7. **Handling Overfitting**:
    - **Pruning** techniques can be used to prevent overfitting, where the tree becomes too complex and fits noise in the training data.
    - Pruning removes branches that do not add much value to the prediction process.

Code:

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

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

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)

# Create Decision Tree classifier using ID3 algorithm (criterion='entropy')
clf = DecisionTreeClassifier(criterion='entropy')

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Predict the target for the testing data
y_pred = clf.predict(X_test)

# Evaluate the accuracy of the model
accuracy = clf.score(X_test, y_test)

# Filter predictions where class is 0 or 1
filtered_samples = [(i, pred) for i, pred in enumerate(y_pred) if pred in [0, 1]]

# Print accuracy
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print predictions for samples where class is 0 or 1
print("Predictions for samples in class 0 and 1:")
for index, prediction in filtered_samples:
    print(f"Sample {index}: Predicted class {prediction}")
```

Output:

```
Accuracy: 95.56%
Predictions for samples in class 0 and 1:
Sample 0: Predicted class 0
Sample 1: Predicted class 1
Sample 2: Predicted class 1
Sample 3: Predicted class 0
Sample 5: Predicted class 1
Sample 7: Predicted class 0
Sample 8: Predicted class 0
Sample 10: Predicted class 1
Sample 11: Predicted class 0
Sample 13: Predicted class 1
Sample 14: Predicted class 1
Sample 15: Predicted class 0
Sample 16: Predicted class 1
Sample 17: Predicted class 1
Sample 18: Predicted class 0
Sample 19: Predicted class 0
Sample 20: Predicted class 1
Sample 21: Predicted class 1
Sample 23: Predicted class 0
Sample 25: Predicted class 1
Sample 26: Predicted class 0
Sample 27: Predicted class 0
Sample 28: Predicted class 1
Sample 30: Predicted class 1
Sample 32: Predicted class 1
Sample 35: Predicted class 0
Sample 36: Predicted class 1
Sample 37: Predicted class 0
Sample 38: Predicted class 1
Sample 41: Predicted class 0
Sample 42: Predicted class 1
Sample 44: Predicted class 1
```

Experiment-5

Aim: Program to Demonstrate K-Nearest Neighbor flowers classification

The **K-Nearest Neighbors (KNN)** algorithm is a simple and effective classification method that can be used to classify flowers (or any data) based on their features. For example, we can classify flowers into species based on features like petal length, petal width, sepal length, and sepal width.

Algorithm:

1. **Start with the Dataset**:
   - Collect a labeled dataset of flowers with features like petal length, petal width, sepal length, and sepal width.
   - Each flower has a corresponding label (e.g., species of the flower).
2. **Choose the Value of kkk**:
   - Decide the number of nearest neighbors (kkk) to use in the algorithm. Common choices are 3, 5, or 7, but this can be tuned based on the problem.
3. **Compute Distance**:
   - For a new flower (test instance) whose species needs to be predicted, calculate the distance between this flower and all other flowers in the dataset.
4. **Find the kkk Nearest Neighbors**:
   - Identify the kkk flowers in the dataset that have the smallest distance to the new flower.
5. **Assign a Class Label (Majority Voting)**:
   - Among the kkk nearest neighbors, count how many belong to each flower species (class).
   - The species (class) with the most votes among the kkk neighbors is assigned as the predicted class of the new flower.
6. **Repeat for All Test Flowers**:
   - Repeat steps 3-5 for every test flower (flower without a known label).

Example:

1. **Training Data**:
   - Features: Sepal Length, Sepal Width, Petal Length, Petal Width.
   - Labels: Flower species like Iris Setosa, Iris Versicolor, and Iris Virginica.
2. **New Flower Data**:
   - Given features: Sepal Length = 5.1, Sepal Width = 3.5, Petal Length = 1.4, Petal Width = 0.2.
   - Predict its species using KNN.
3. **Steps**:
   - Compute the Euclidean distance between the new flower and all flowers in the training dataset.
   - Choose k=3k = 3k=3 (for example) and find the 3 nearest neighbors.
   - Perform majority voting on the species of the 3 nearest neighbors.
   - Assign the species with the majority vote to the new flower.

Code:

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

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

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)

# Create K-Nearest Neighbors classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)

# Train the classifier on the training data
knn.fit(X_train, y_train)

# Predict the target for the testing data
y_pred = knn.predict(X_test)

# Evaluate the accuracy of the model
accuracy = knn.score(X_test, y_test)

# Filter predictions where class is 0 or 1
filtered_samples = [(i, pred) for i, pred in enumerate(y_pred) if pred in [0, 1]]

# Print accuracy
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print predictions for samples where class is 0 or 1
print("Predictions for samples in class 0 and 1:")
for index, prediction in filtered_samples:
    print(f"Sample {index}: Predicted class {prediction}")
```

Output:

```
Accuracy: 97.78%
Predictions for samples in class 0 and 1:
Sample 0: Predicted class 0
Sample 1: Predicted class 1
Sample 2: Predicted class 1
Sample 3: Predicted class 0
Sample 5: Predicted class 1
Sample 7: Predicted class 0
Sample 8: Predicted class 0
Sample 10: Predicted class 1
Sample 11: Predicted class 0
Sample 13: Predicted class 1
Sample 14: Predicted class 1
Sample 15: Predicted class 0
Sample 16: Predicted class 1
Sample 17: Predicted class 1
Sample 18: Predicted class 0
Sample 19: Predicted class 0
Sample 20: Predicted class 1
Sample 21: Predicted class 1
Sample 22: Predicted class 1
Sample 23: Predicted class 0
Sample 25: Predicted class 1
Sample 26: Predicted class 0
Sample 27: Predicted class 0
Sample 28: Predicted class 1
Sample 30: Predicted class 1
Sample 32: Predicted class 1
Sample 35: Predicted class 0
Sample 36: Predicted class 1
Sample 37: Predicted class 0
Sample 38: Predicted class 1
Sample 41: Predicted class 0
Sample 42: Predicted class 1
Sample 44: Predicted class 1
```

Experiment-6

Aim: Program to Demonstrate Naïve-Bayes Classifier

Naive Bayes is a **probabilistic classifier** based on **Bayes' Theorem**, which assumes that the features (input variables) are **independent** of each other given the class label. This assumption of independence is what makes the classifier "naive." Despite this simplification, Naive Bayes often performs very well in various applications, especially for text classification and spam detection.

 Naive Assumption:

- Naive Bayes assumes that all features are **independent** of each other. This means that the presence of one feature does not influence the presence of any other feature, which simplifies the calculation of the likelihood P(X|C)P(X|C)P(X|C).
- Despite this strong assumption, Naive Bayes performs well in practice, especially for high-dimensional datasets (e.g., text data).

 Different Types of Naive Bayes:

- There are several variations of the Naive Bayes classifier depending on the type of data:
  - **Gaussian Naive Bayes**: Assumes that the continuous features follow a normal (Gaussian) distribution.
  - **Multinomial Naive Bayes**: Used for discrete features, typically for text data where the features represent word counts or term frequencies.
  - **Bernoulli Naive Bayes**: Used for binary/boolean data, where features represent binary outcomes (e.g., presence or absence of a word in a document).

Applications of Naive Bayes:

1. **Spam Filtering**:
   - Naive Bayes is commonly used in email spam filters to classify emails as spam or not spam based on the frequency of words in the email body.
2. **Text Classification**:
   - Used in sentiment analysis, document classification, and news article categorization by analyzing the frequency of words in a document.
3. **Medical Diagnosis**:
   - Applied in medical diagnosis to predict diseases based on symptoms, using the probabilities of various conditions given the symptoms.
4. **Sentiment Analysis**:
   - In social media analysis, Naive Bayes is used to classify reviews, tweets, or comments as positive, negative, or neutral based on word occurrences.

Code:

```
# Import required libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
import pandas as pd
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)
nb_classifier = GaussianNB()
nb_classifier.fit(X_train, y_train)
y_pred = nb_classifier.predict(X_test)
accuracy = nb_classifier.score(X_test, y_test)
summary_df = pd.DataFrame({
    'Sample Index': range(len(y_test)),
    'Actual Class': y_test,
    'Predicted Class': y_pred
})
# Filter predictions for classes 0 and 1
filtered_summary = summary_df[summary_df['Predicted Class'].isin([0, 1])]
# Print accuracy
print(f"Accuracy: {accuracy * 100:.2f}%")
# Display the summary DataFrame with predictions
print("\nSummary of Predictions (Actual vs Predicted):")
print(filtered_summary)
```

Output:

```
Accuracy: 93.33%

Summary of Predictions (Actual vs Predicted):
    Sample Index  Actual Class  Predicted Class
0              0             0                0
1              1             1                1
2              2             1                1
3              3             0                0
7              7             0                0
8              8             0                0
10            10             1                1
11            11             0                0
13            13             1                1
14            14             1                1
15            15             0                0
16            16             1                1
17            17             1                1
18            18             0                0
19            19             0                0
20            20             1                1
21            21             1                1
23            23             0                0
25            25             1                1
26            26             0                0
27            27             0                0
28            28             1                1
30            30             1                1
32            32             1                1
35            35             0                0
36            36             1                1
37            37             0                0
38            38             1                1
41            41             0                0
42            42             2                1
44            44             1                1
```

Experiment-7

Aim: Program to demonstrate PCA and LDA on Iris dataset

Both **PCA** and **LDA** are dimensionality reduction techniques but are used for different purposes. Let's go through their concepts briefly and then see how to apply both on the Iris dataset.

1. **Principal Component Analysis (PCA)**

- **PCA** is an **unsupervised** technique used to reduce the dimensionality of the dataset by projecting the data onto directions (principal components) that maximize the variance.
- It doesn't consider class labels and focuses solely on capturing the most variance in the data.

Steps for PCA:

1. Standardize the data (mean = 0, variance = 1).
2. Compute the covariance matrix.
3. Calculate the eigenvalues and eigenvectors of the covariance matrix.
4. Choose the top kkk eigenvectors (principal components).
5. Project the data onto these principal components to get a reduced dimensionality.

2. **Linear Discriminant Analysis (LDA)**

- **LDA** is a **supervised** technique that finds a projection maximizing the separation between multiple classes.
- It focuses on maximizing the difference between class means while minimizing the variance within each class.

Steps for LDA:

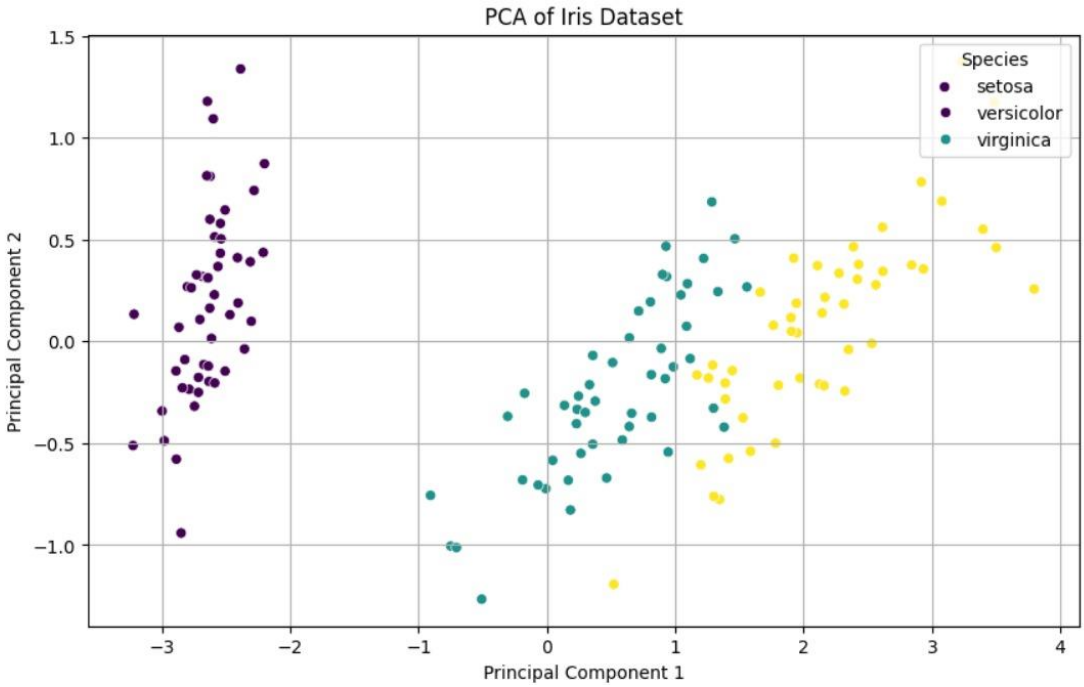
1. Compute the mean vectors for each class.
2. Compute the within-class scatter matrix and the between-class scatter matrix.
3. Calculate the eigenvalues and eigenvectors of the matrix Sw−1SbS_w^{-1} S_bSw−1Sb (where SwS_wSw is the within-class scatter matrix and SbS_bSb is the between-class scatter matrix).
4. Select the top eigenvectors that correspond to the largest eigenvalues and use them to project the data onto a lower-dimensional space.

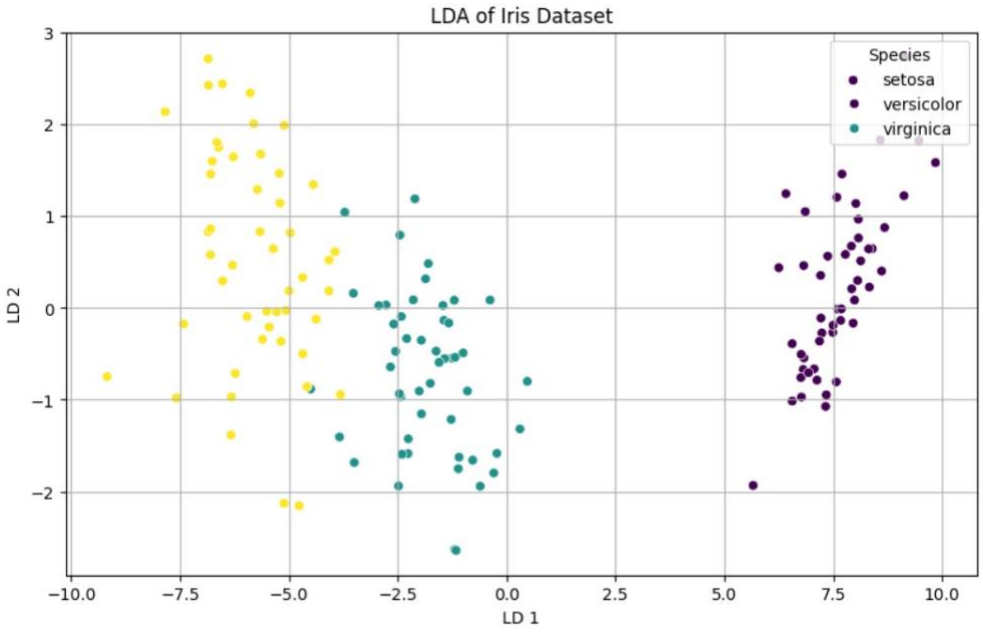
Code:

```
# Import required libraries
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.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split

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

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)

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

# Create a DataFrame for PCA results
pca_df = pd.DataFrame(data=X_pca, columns=['Principal Component 1', 'Principal Component 2'])
pca_df['Target'] = y
# Plotting PCA results
plt.figure(figsize=(10, 6))
```

```
sns.scatterplot(x='Principal Component 1', y='Principal Component 2', hue='Target', data=pca_df, palette='viridis')

plt.title('PCA of Iris Dataset')

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.legend(title='Species', loc='upper right', labels=target_names)

plt.grid()

plt.show()

# --- LDA ---

# Perform LDA

lda = LinearDiscriminantAnalysis(n_components=2)

X_lda = lda.fit_transform(X, y)

# Create a DataFrame for LDA results

lda_df = pd.DataFrame(data=X_lda, columns=['LD 1', 'LD 2'])

lda_df['Target'] = y

# Plotting LDA results

plt.figure(figsize=(10, 6))

sns.scatterplot(x='LD 1', y='LD 2', hue='Target', data=lda_df, palette='viridis')

plt.title('LDA of Iris Dataset')

plt.xlabel('LD 1')

plt.ylabel('LD 2')

plt.legend(title='Species', loc='upper right', labels=target_names)

plt.grid()

plt.show()
```

Output:


PCA of Iris Dataset
Species
setosa
versicolor
virginica
Principal Component 2
Principal Component 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4


LDA of Iris Dataset
Species
setosa
versicolor
virginica
LD 2
LD 1
3
2
1
0
−1
−2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Experiment-8

Aim: Program to demonstrate DBSCAN clustering algorithm

DBSCAN is a popular **density-based clustering algorithm**. It groups together points that are close to each other (dense regions) and marks outliers (points in sparse regions) that don't belong to any cluster. It is particularly effective when clusters have irregular shapes or when noise is present.

Algorithm:

1. **Mark all points as unvisited**.
2. For each unvisited point:
    - **Check if it's a core point**: Count how many points (including itself) are within eps distance of it.
        - If the number of points is greater than or equal to minPts, it's a **core point**, and a new cluster is started.
        - Otherwise, the point is marked as **noise** (it may be reclassified later if it becomes part of another cluster).
3. **Expand the cluster**:
    - For each core point in the cluster, visit its neighbors. If they are core points, they are added to the cluster, and their neighbors are visited as well. This process continues until no new points can be added to the cluster.
4. **Repeat** until all points have been visited.

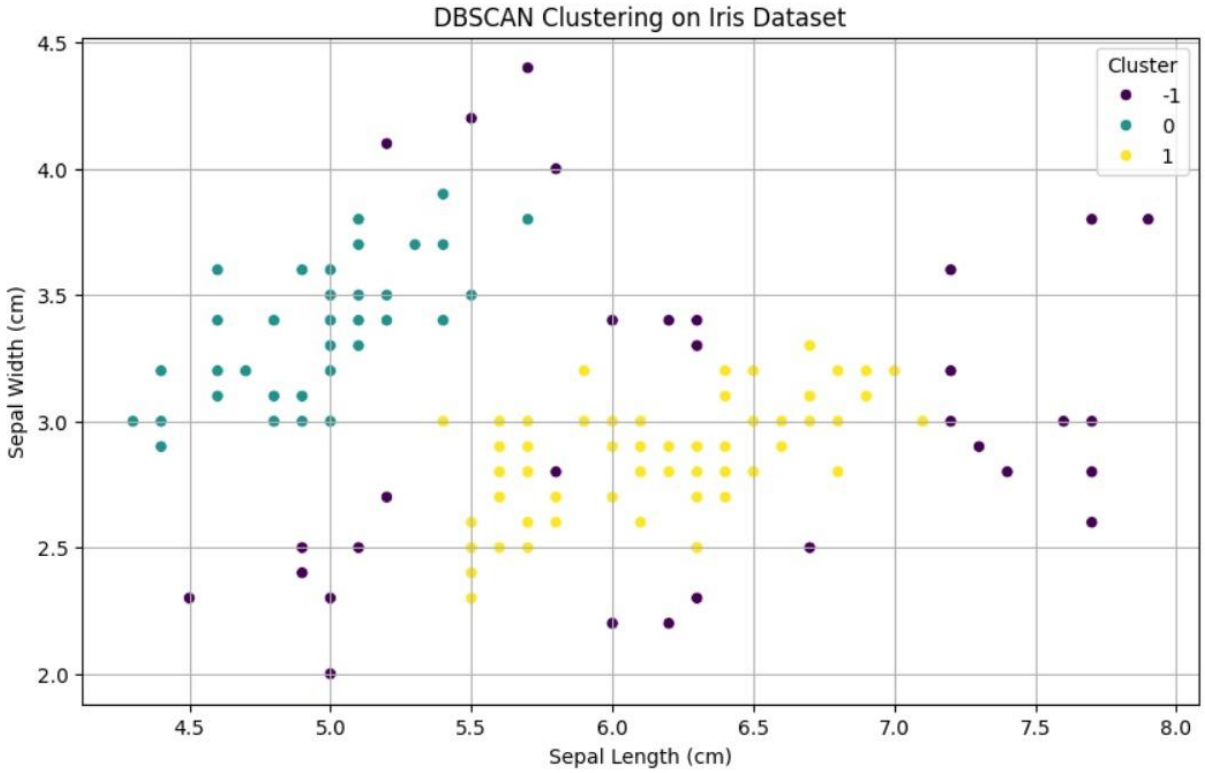
Points to Remember:

1. **Core Point**: A point is a core point if it has at least a minimum number (minPts) of neighboring points within a given radius (epsilon, or eps).
2. **Border Point**: A point that is not a core point but is within the epsilon distance of a core point. It lies on the boundary of a cluster.
3. **Noise Point**: A point that is neither a core point nor a border point, i.e., it doesn't have enough neighbors within epsilon and is not reachable from any core point.

Code:

```
# Import required libraries
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.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

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

# Standardizing the features
X_scaled = StandardScaler().fit_transform(X)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_labels = dbscan.fit_predict(X_scaled)

# Create a DataFrame to hold the results
dbscan_df = pd.DataFrame(X, columns=iris.feature_names)
dbscan_df['Cluster'] = dbscan_labels

# Plotting the results
plt.figure(figsize=(10, 6))
sns.scatterplot(x='sepal length (cm)', y='sepal width (cm)', hue='Cluster', data=dbscan_df, palette='viridis',
legend='full')
plt.title('DBSCAN Clustering on Iris Dataset')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.grid()
plt.show()
```

Output:


DBSCAN Clustering on Iris Dataset
Cluster
-1
0
1
Sepal Width (cm)
Sepal Length (cm)
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

Experiment-9

Aim: Program to demonstrate K-Medoid clustering algorithm

K-Medoids is a **partitioning-based clustering algorithm** similar to k-means, but instead of using the mean of the points to define the center of the cluster, it uses **medoids** (actual data points) as cluster centers. The objective of K-Medoids is to minimize the sum of dissimilarities between points in a cluster and the medoid of that cluster.

K-Medoids is more **robust to outliers** and noise because it uses actual data points as centers, unlike k-means which can be skewed by extreme values.

Algorithm Steps:

1. **Initialization**:
    - Randomly select **k** data points from the dataset as the initial **medoids** (representative points of clusters).
2. **Assign each point to the nearest medoid**:
    - Calculate the dissimilarity (distance, typically Manhattan or Euclidean) of each point to every medoid.
    - Assign each point to the cluster with the nearest medoid.
3. **Update Medoids**:
    - For each cluster, calculate the total dissimilarity of all points in the cluster to each point within the cluster.
    - Select the point that minimizes the total dissimilarity as the new medoid for that cluster.
4. **Repeat**:
    - Repeat the assignment and update steps until the medoids no longer change or the total cost (sum of dissimilarities) stabilizes.
5. **Termination**:
    - The algorithm converges when no further changes occur in the medoids or the cluster assignments.

Applications:

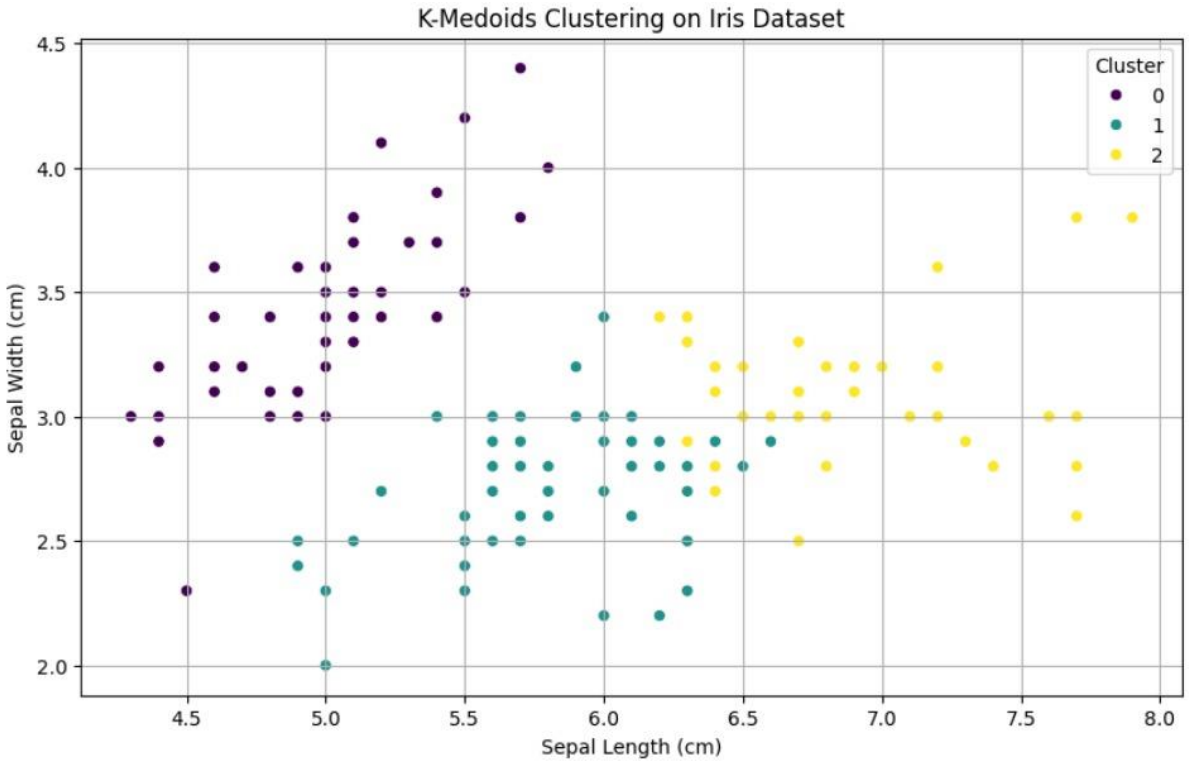
- Customer segmentation**.**
- Gene expression clustering**.**
- Image segmentation**.**

Code:

```
# Import required libraries
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_extra.cluster import KMedoids
from sklearn.preprocessing import StandardScaler
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# Standardizing the features
X_scaled = StandardScaler().fit_transform(X)
# Apply K-Medoids
kmedoids = KMedoids(n_clusters=3, random_state=1)
kmedoids_labels = kmedoids.fit_predict(X_scaled)
# Create a DataFrame to hold the results
kmedoids_df = pd.DataFrame(X, columns=iris.feature_names)
kmedoids_df['Cluster'] = kmedoids_labels
# Plotting the results
plt.figure(figsize=(10, 6))
sns.scatterplot(x='sepal length (cm)', y='sepal width (cm)', hue='Cluster', data=kmedoids_df,
palette='viridis', legend='full')
plt.title('K-Medoids Clustering on Iris Dataset')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.grid()
plt.show()
```

```
# Display the medoids
medoids = kmedoids.cluster_centers_
print("Medoids of each cluster:")
print(medoids)
```

Output:




```
Medoids of each cluster:
[[-1.02184904  0.78880759 -1.2833891  -1.3154443 ]
 [-0.17367395 -0.59237301  0.42173371  0.13250973]
 [ 1.2803405   0.09821729  0.93327055  1.18556721]]
```

Experiment-10

Aim: Program to demonstrate K-Means Clustering Algorithm on Handwritten Dataset

K-Means is a popular clustering algorithm that aims to partition data into **k clusters**, where each data point belongs to the cluster with the nearest mean (centroid). For a **handwritten digits dataset**, like the MNIST dataset, K-Means can be used to group similar digits based on their pixel values. Although K-Means is an unsupervised algorithm (meaning it doesn't rely on labels), we can still apply it to see if it clusters similar digits together.

Algorithm Steps:

1. **Initialization**:
   - Choose **k** random points from the dataset as initial centroids (representative of clusters).
2. **Assignment Step**:
   - Assign each data point (handwritten digit) to the nearest centroid based on a distance metric (usually Euclidean distance).
3. **Update Step**:
   - After all points are assigned to clusters, compute the new centroids by taking the mean of the points in each cluster.
4. **Repeat**:
   - Repeat the assignment and update steps until the centroids no longer change (convergence).
5. **Termination**:
   - The algorithm stops when the centroids stabilize or the maximum number of iterations is reached.

Steps for Applying K-Means to Handwritten Digits Dataset:

1. **Dataset**:
   - You can use a dataset like **MNIST** (28x28 pixel images of handwritten digits) or any similar dataset of handwritten digits.
2. **Preprocessing**:
   - Flatten each image into a 1D vector. For example, a 28x28 pixel image becomes a 784-dimensional vector.
   - Normalize pixel values to range between 0 and 1 (optional but recommended for better convergence).
3. **Apply K-Means**:
   - Choose **k=10** (since there are 10 digit classes: 0-9).
   - Run the K-Means algorithm to cluster the digits.
4. **Evaluate**:

- Since K-Means is unsupervised, there is no direct label information. However, you can evaluate the clusters by seeing how well they align with the actual digit labels (using metrics like **cluster purity** or **adjusted Rand index**).

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import fetch_openml
# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X = mnist.data
y = mnist.target.astype(int)
# Reduce dataset size for faster processing (optional)
X_sample = X.sample(n=1000, random_state=1)
# Apply K-Means
n_clusters = 10  # Number of digits
kmeans = KMeans(n_clusters=n_clusters, random_state=1)
kmeans.fit(X_sample)
# Get cluster labels
labels = kmeans.labels_
# Visualize the clusters
plt.figure(figsize=(10, 8))
for i in range(n_clusters):
    plt.subplot(2, 5, i + 1)
    # Get the cluster center (medoid)
    cluster_center = kmeans.cluster_centers_[i].reshape(28, 28)
    plt.imshow(cluster_center, cmap='gray')
 plt.title(f'Cluster {i}')
  plt.axis('off')
```

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

Output: