1. Simple python program using conditional statements, looping , performing operations such as Insert , Update, Delete, Display, Sorting and searching on data types like List, Tuple, Set, Dictionary.

```python
def list_operations():
    my_list = []
    print("\n--- List Operations ---")
    print("1. Insert")
    print("2. Update")
    print("3. Delete")
    print("4. Display")
    print("5. Sort")
    print("6. Search")
    print("7. Exit")

    while True:
        choice = input("Enter choice: ")

        if choice == '1':
            value = input("Enter value to insert: ")
            my_list.append(value)
        elif choice == '2':
            old = input("Enter value to update: ")
            if old in my_list:
                new = input("Enter new value: ")
                my_list[my_list.index(old)] = new
            else:
                print("Value not found.")
        elif choice == '3':
            value = input("Enter value to delete: ")
            if value in my_list:
                my_list.remove(value)
            else:
                print("Value not found.")
        elif choice == '4':
            print("List contents:", my_list)
        elif choice == '5':
            my_list.sort()
            print("Sorted List:", my_list)
        elif choice == '6':
            search = input("Enter value to search: ")
            print("Found!" if search in my_list else "Not found.")
        elif choice == '7':
            break
        else:
            print("Invalid choice.")

def tuple_operations():
    my_tuple = ("apple", "banana", "cherry")
    print("\n--- Tuple Operations (Immutable) ---")
    print("Original Tuple:", my_tuple)
    print("1. Display")
    print("2. Search")
    print("3. Convert to List and Add Item")
    choice = input("Enter choice: ")
    if choice == '1':
        print("Tuple Contents:", my_tuple)
    elif choice == '2':
        item = input("Enter item to search: ")
        print("Found!" if item in my_tuple else "Not found.")
    elif choice == '3':
        item = input("Enter item to add: ")
        temp = list(my_tuple)
        temp.append(item)
        my_tuple = tuple(temp)
        print("Updated Tuple:", my_tuple)
    else:
        print("Invalid choice.")

def set_operations():
    my_set = set()
    print("\n--- Set Operations ---")
    print("1. Insert")
    print("2. Delete")
    print("3. Display")
    print("4. Search")
    print("5. Exit")
    while True:
        choice = input("Enter choice: ")

        if choice == '1':
            value = input("Enter value to insert: ")
            my_set.add(value)
        elif choice == '2':
            value = input("Enter value to delete: ")
            my_set.discard(value)
        elif choice == '3':
            print("Set contents:", my_set)
        elif choice == '4':
            value = input("Enter value to search: ")
            print("Found!" if value in my_set else "Not found.")
        elif choice == '5':
            break
        else:
            print("Invalid choice.")

def dict_operations():
    my_dict = {}
    print("\n--- Dictionary Operations ---")
    print("1. Insert")
    print("2. Update")
    print("3. Delete")
    print("4. Display")
    print("5. Search")
    print("6. Exit")
    while True:
        choice = input("Enter choice: ")
        if choice == '1':
            key = input("Enter key: ")
            value = input("Enter value: ")
            my_dict[key] = value
        elif choice == '2':
            key = input("Enter key to update: ")
            if key in my_dict:
                value = input("Enter new value: ")
                my_dict[key] = value
            else:
                print("Key not found.")
        elif choice == '3':
            key = input("Enter key to delete: ")
            if key in my_dict:
                del my_dict[key]
            else:
                print("Key not found.")
        elif choice == '4':
            print("Dictionary contents:", my_dict)
        elif choice == '5':
            key = input("Enter key to search: ")
            print("Found!" if key in my_dict else "Not found.")
        elif choice == '6':
            break
        else:
            print("Invalid choice.")

# Main Program Loop
print("\n===== Main Menu =====")
print("1. List")
print("2. Tuple")
print("3. Set")
print("4. Dictionary")
print("5. Exit")

while True:
    main_choice = input("Enter Choice: ")

    if main_choice == '1':
        list_operations()
    elif main_choice == '2':
        tuple_operations()
    elif main_choice == '3':
        set_operations()
    elif main_choice == '4':
        dict_operations()
    elif main_choice == '5':
        print("Exiting Program.")
        break
    else:
        print("Invalid choice. Try again.")
```

Sample output:

```
===== Main Menu =====
1. List
2. Tuple
3. Set
4. Dictionary
5. Exit
Enter Choice: 1

--- List Operations ---
1. Insert
2. Update
3. Delete
4. Display
5. Sort
6. Search
7. Exit
Enter choice: 1
Enter value to insert: mango
Enter choice: 1
Enter value to insert: apple
Enter choice: 4
List contents: ['mango', 'apple']
Enter choice: 5
Sorted List: ['apple', 'mango']
Enter choice: 6
Enter value to search: mango
Found!
Enter choice: 7


Enter Choice: 2

--- Tuple Operations (Immutable) ---
Original Tuple: ('apple', 'banana', 'cherry')
1. Display
2. Search
3. Convert to List and Add Item
Enter choice: 2
Enter item to search: banana
Found!



Enter Choice: 3

--- Set Operations ---
1. Insert
2. Delete
3. Display
4. Search
5. Exit
Enter choice: 1
Enter value to insert: orange
Enter choice: 3
Set contents: {'orange'}
Enter choice: 4
Enter value to search: apple
Not found.
Enter choice: 5
```

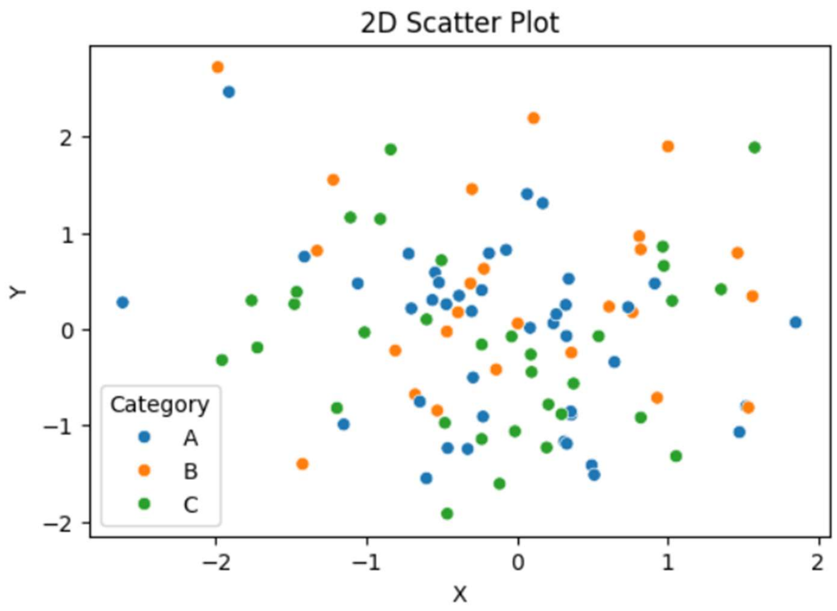
```
Enter Choice: 4

--- Dictionary Operations ---
1. Insert
2. Update
3. Delete
4. Display
5. Search
6. Exit
Enter choice: 1
Enter key: name
Enter value: Alice
Enter choice: 1
Enter key: age
Enter value: 25
Enter choice: 4
Dictionary contents: {'name': 'Alice', 'age': '25'}
Enter choice: 2
Enter key to update: age
Enter new value: 26
Enter choice: 5
Enter key to search: name
Found!
Enter choice: 6
```

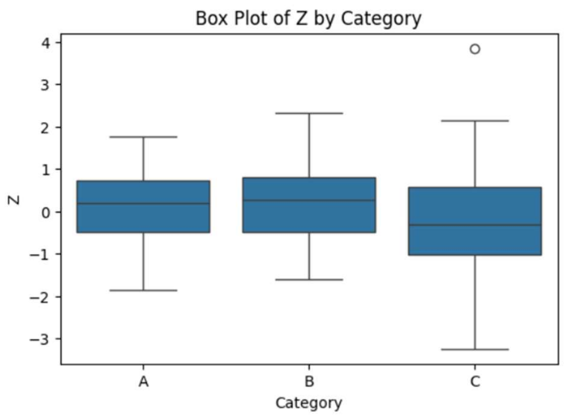
```
Enter Choice: 5
Exiting Program.
```

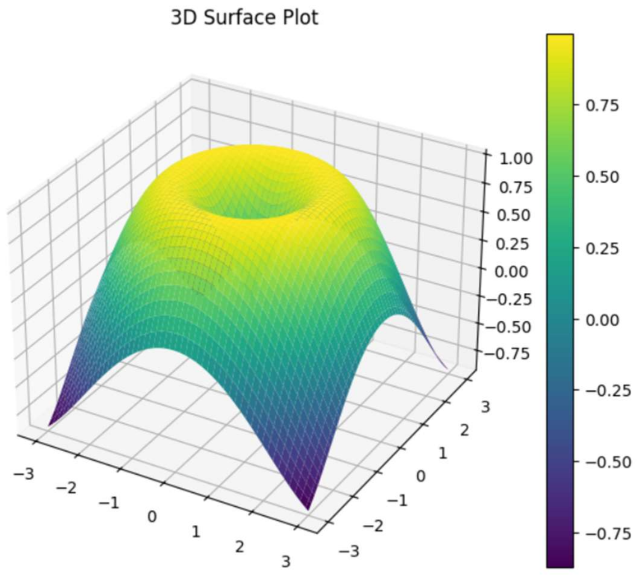
2. Visualize the n-dimensional data using Scatter plots, box plot, heat maps, contour plots, 3D surface plots using python packages.

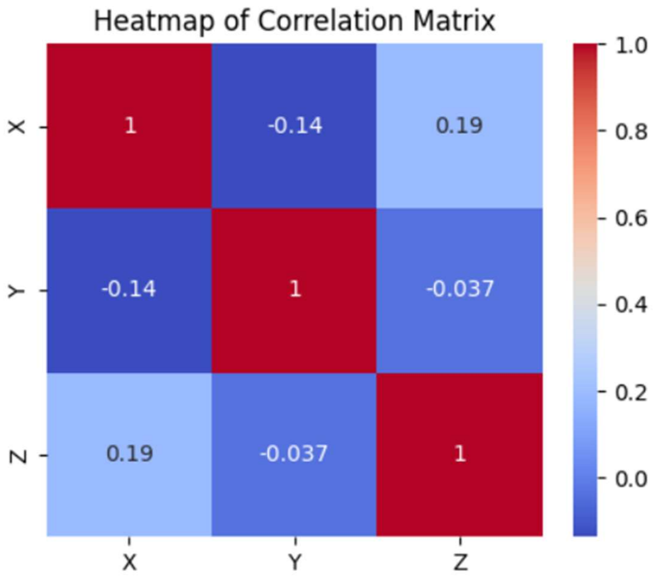
   pip install numpy pandas seaborn matplotlib

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate some random n-dimensional data
np.random.seed(42)
n = 100
data = pd.DataFrame({
    'X': np.random.normal(0, 1, n),
    'Y': np.random.normal(0, 1, n),
    'Z': np.random.normal(0, 1, n),
    'Category': np.random.choice(['A', 'B', 'C'], n)
})

# -------- Scatter Plot (2D) --------
def scatter_plot():
    plt.figure(figsize=(6, 4))
    sns.scatterplot(data=data, x='X', y='Y', hue='Category')
    plt.title("2D Scatter Plot")
    plt.show()

# -------- Box Plot --------
def box_plot():
    plt.figure(figsize=(6, 4))
    sns.boxplot(data=data, x='Category', y='Z')
    plt.title("Box Plot of Z by Category")
    plt.show()

# -------- Heatmap --------
def heatmap():
    correlation = data[['X', 'Y', 'Z']].corr()
    plt.figure(figsize=(5, 4))
    sns.heatmap(correlation, annot=True, cmap='coolwarm')
    plt.title("Heatmap of Correlation Matrix")
    plt.show()

# -------- Contour Plot --------
def contour_plot():
    x = np.linspace(-3, 3, 100)
    y = np.linspace(-3, 3, 100)
    X, Y = np.meshgrid(x, y)
    Z = np.sin(X**2 + Y**2)

    plt.figure(figsize=(6, 5))
    cp = plt.contourf(X, Y, Z, cmap='viridis')
    plt.colorbar(cp)
    plt.title("Contour Plot of sin(X² + Y²)")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

# -------- 3D Surface Plot --------
def surface_plot():
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')

    x = np.linspace(-3, 3, 100)
    y = np.linspace(-3, 3, 100)
    X, Y = np.meshgrid(x, y)
    Z = np.sin(np.sqrt(X**2 + Y**2))

    surf = ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
    fig.colorbar(surf)
    ax.set_title("3D Surface Plot")
    plt.show()

# -------- Call All Plots --------
scatter_plot()
box_plot()
heatmap()
contour_plot()
surface_plot()
```

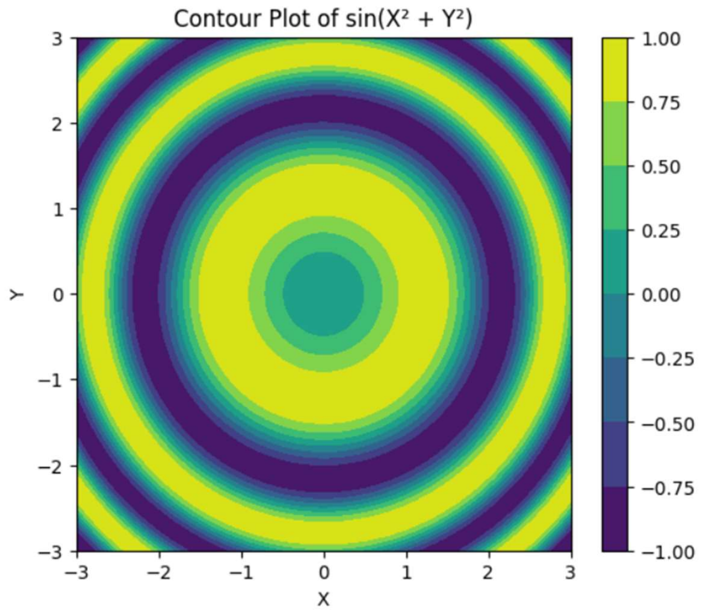








	X	Y	Z
X	1	-0.14	0.19
Y	-0.14	1	-0.037
Z	0.19	-0.037	1



3. Write a program to implement Hill Climbing Algorithm.

```python
import random

# Objective function (maximize this)
def objective_function(x):
    return -x**2 + 5

# Hill Climbing algorithm
def hill_climbing(start_x, step_size, max_iterations):
    current_x = start_x
    current_score = objective_function(current_x)

    for i in range(max_iterations):
        # Try a new solution nearby
        new_x = current_x + random.uniform(-step_size, step_size)
        new_score = objective_function(new_x)

        print(f"Iteration {i+1}: x = {current_x:.4f}, f(x) = {current_score:.4f}")

        # If the new solution is better, move to it
        if new_score > current_score:
            current_x = new_x
            current_score = new_score
        else:
            # No improvement; stop if you want a simple version
            pass

    print("\nFinal Solution:")
    print(f"x = {current_x:.4f}, f(x) = {current_score:.4f}")
    return current_x, current_score

# Run the algorithm
# best_x, best_score = hill_climbing(start_x=random.uniform(-5, 5), step_size=0.1,
max_iterations=100)
best_x, best_score = hill_climbing(start_x=0.1, step_size=0.05, max_iterations=5)
```

Sample output:

```
Iteration 1: x = 0.1000, f(x) = 4.9900

Iteration 2: x = 0.0587, f(x) = 4.9966

Iteration 3: x = 0.0235, f(x) = 4.9994

Iteration 4: x = -0.0112, f(x) = 4.9999

Iteration 5: x = -0.0421, f(x) = 4.9982


Final Solution:

x = -0.0112, f(x) = 4.9999
```

4. a) Write a program to implement the Best First Search (BFS) algorithm.

```
import heapq

class Node:
    def __init__(self, name, heuristic, parent=None):
        self.name = name
        self.heuristic = heuristic
        self.parent = parent

    def __lt__(self, other):
        return self.heuristic < other.heuristic

def best_first_search(graph, start, goal, heuristic_values):
    open_list = []
    closed_list = set()

    heapq.heappush(open_list, Node(start, heuristic_values[start]))

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.name == goal:
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]

        if current_node.name in closed_list:
            continue

        closed_list.add(current_node.name)

        for neighbor in graph.get(current_node.name, []):
            if neighbor not in closed_list:
                heapq.heappush(open_list, Node(neighbor, heuristic_values[neighbor],
current_node))

    return None

# Take custom input for the graph and heuristic values
def get_input():
    graph = {}
    heuristic_values = {}

    print("Enter the graph structure:")
    n = int(input("Enter number of nodes: "))

    for _ in range(n):
        node = input("Enter node name: ")
        neighbors = input(f"Enter neighbors for {node} (comma separated): ").split(",")
        graph[node] = [neighbor.strip() for neighbor in neighbors]
```

```python
    print("\nEnter heuristic values:")
    for _ in range(n):
        node = input("Enter node name for heuristic: ")
        heuristic = int(input(f"Enter heuristic value for {node}: "))
        heuristic_values[node] = heuristic

    start = input("\nEnter the start node: ")
    goal = input("Enter the goal node: ")

    return graph, heuristic_values, start, goal

# Main execution
if __name__ == "__main__":
    graph, heuristic_values, start_node, goal_node = get_input()
    path = best_first_search(graph, start_node, goal_node, heuristic_values)

    if path:
        print(f"\nPath from {start_node} to {goal_node}: {path}")
    else:
        print(f"\nNo path found from {start_node} to {goal_node}.")
```

Sample output:

```
Enter the graph structure:

Enter number of nodes: 5

Enter node name: A

Enter neighbors for A (comma separated): B, C

Enter node name: B

Enter neighbors for B (comma separated): A, D, E

Enter node name: C

Enter neighbors for C (comma separated): A, F

Enter node name: D

Enter neighbors for D (comma separated): B

Enter node name: E

Enter neighbors for E (comma separated): B, G


Enter heuristic values:

Enter node name for heuristic: A

Enter heuristic value for A: 6

Enter node name for heuristic: B

Enter heuristic value for B: 4

Enter node name for heuristic: C
Enter heuristic value for C: 3

Enter node name for heuristic: D

Enter heuristic value for D: 7

Enter node name for heuristic: E

Enter heuristic value for E: 2


Enter the start node: A

Enter the goal node: G


Path from A to G: ['A', 'C', 'F', 'E', 'G']
```

4b) Write a program to implement the A* algorithm.

```
import heapq

class Node:
    def __init__(self, name, heuristic, parent=None):
        self.name = name
        self.heuristic = heuristic
        self.parent = parent

    def __lt__(self, other):
        return self.heuristic < other.heuristic

def best_first_search(graph, start, goal, heuristic_values):
    open_list = []
    closed_list = set()

    heapq.heappush(open_list, Node(start, heuristic_values[start]))

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.name == goal:
            path = []
            while current_node:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]

        if current_node.name in closed_list:
            continue

        closed_list.add(current_node.name)

        for neighbor in graph.get(current_node.name, []):
            if neighbor not in closed_list:
                heapq.heappush(open_list, Node(neighbor, heuristic_values[neighbor], 
current_node))

    return None

# Take custom input for the graph and heuristic values
def get_input():
    graph = {}
    heuristic_values = {}

    print("Enter the graph structure:")
    n = int(input("Enter number of nodes: "))

    for _ in range(n):
        node = input("Enter node name: ")
        neighbors = input(f"Enter neighbors for {node} (comma separated): ").split(",")
```

```python
        graph[node] = [neighbor.strip() for neighbor in neighbors]

    print("\nEnter heuristic values:")
    for _ in range(n):
        node = input("Enter node name for heuristic: ")
        heuristic = int(input(f"Enter heuristic value for {node}: "))
        heuristic_values[node] = heuristic

    start = input("\nEnter the start node: ")
    goal = input("Enter the goal node: ")

    return graph, heuristic_values, start, goal

# Main execution
if __name__ == "__main__":
    graph, heuristic_values, start_node, goal_node = get_input()
    path = best_first_search(graph, start_node, goal_node, heuristic_values)

    if path:
        print(f"\nPath from {start_node} to {goal_node}: {path}")
    else:
        print(f"\nNo path found from {start_node} to {goal_node}.")
```

Sample Output:

```
Enter the graph structure:

Enter number of nodes: 5

Enter node name: A

Enter neighbors for A (comma separated): B, C

Enter node name: B

Enter neighbors for B (comma separated): A, D, E

Enter node name: C

Enter neighbors for C (comma separated): A, F

Enter node name: D

Enter neighbors for D (comma separated): B

Enter node name: E

Enter neighbors for E (comma separated): B, G


Enter heuristic values:

Enter node name for heuristic: A

Enter heuristic value for A: 6

Enter node name for heuristic: B

Enter heuristic value for B: 4
Enter node name for heuristic: C

Enter heuristic value for C: 3

Enter node name for heuristic: D

Enter heuristic value for D: 7

Enter node name for heuristic: E

Enter heuristic value for E: 2


Enter edge costs (cost between nodes):

Enter number of edges: 6

Enter edge (u, v, cost): A, B, 1

Enter edge (u, v, cost): A, C, 2

Enter edge (u, v, cost): B, D, 2

Enter edge (u, v, cost): B, E, 1

Enter edge (u, v, cost): C, F, 3

Enter edge (u, v, cost): E, G, 4


Enter the start node: A

Enter the goal node: G


Path from A to G: ['A', 'B', 'E', 'G']
```

5) Write a program to implement Min-Max algorithm and Alpha-beta pruning algorithm.

```python
def minimax(depth, node_index, is_maximizing_player, scores, target_depth):
    if depth == target_depth:
        return scores[node_index]

    if is_maximizing_player:
        return max(
            minimax(depth + 1, node_index * 2, False, scores, target_depth),
            minimax(depth + 1, node_index * 2 + 1, False, scores, target_depth)
        )
    else:
        return min(
            minimax(depth + 1, node_index * 2, True, scores, target_depth),
            minimax(depth + 1, node_index * 2 + 1, True, scores, target_depth)
        )

def alphabeta(depth, node_index, is_maximizing_player, scores, target_depth, alpha, beta):
    if depth == target_depth:
        return scores[node_index]

    if is_maximizing_player:
        max_eval = float('-inf')
        for i in range(2):
            eval = alphabeta(depth + 1, node_index * 2 + i, False, scores, target_depth,
alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):
            eval = alphabeta(depth + 1, node_index * 2 + i, True, scores, target_depth,
alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval


# ---------------- Input Section ----------------

print("Enter the depth of the game tree (e.g., 3 for 8 leaf nodes):")
tree_depth = int(input("Depth: "))
num_leaves = 2 ** tree_depth

print(f"Enter {num_leaves} leaf node scores separated by space:")
scores_input = input("Scores: ")
scores = list(map(int, scores_input.strip().split()))

if len(scores) != num_leaves:
    print(f"Error: Expected {num_leaves} scores, but got {len(scores)}.")
else:
    optimal_value = minimax(0, 0, True, scores, tree_depth)
    print(f"\nOptimal value using Minimax: {optimal_value}")

    optimal_value_ab = alphabeta(0, 0, True, scores, tree_depth, float('-inf'),
float('inf'))
    print(f"Optimal value using Alpha-Beta Pruning: {optimal_value_ab}")
```

Sample output:

```
Enter the depth of the game tree (e.g., 3 for 8 leaf nodes):

Depth: 3

Enter 8 leaf node scores separated by space:

Scores: 3 5 6 9 1 2 0 -1


Optimal value using Minimax: 5

Optimal value using Alpha-Beta Pruning: 5
```

6) Write a program to develop the Naive Bayes classifier based on split up of training and testing dataset as 90-10, 70-30. a) Iris dataset b) Titanic dataset

```
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split


# -----------------------------------------Build the classifier-------------------------------

class NaiveBayesClassifier:
    def __init__(self):
        self.prior = {}
        self.conditional = {}

    def fit(self, X, y):
        self.classes = np.unique(y)
        for c in self.classes:
            self.prior[c] = np.mean(y == c)

        for feature in X.columns:
            self.conditional[feature] = {}
            for c in self.classes:
                feature_values = X[feature][y == c]
                self.conditional[feature][c] = {
                    'mean': np.mean(feature_values),
                    'std': np.std(feature_values)
                }

    def predict(self, X):
        y_pred = []
        for _, sample in X.iterrows():
            probabilities = {}
            for c in self.classes:
                probabilities[c] = self.prior[c]
                for feature in X.columns:
                    mean = self.conditional[feature][c]['mean']
                    std = self.conditional[feature][c]['std']
                    x = sample[feature]
                    probabilities[c] *= self._gaussian_pdf(x, mean, std)
            y_pred.append(max(probabilities, key=probabilities.get))
        return y_pred

    def _gaussian_pdf(self, x, mean, std):
        if std == 0:
            return 1.0 if x == mean else 0.0
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent
```

```python
# ------------------------------------------------a)Iris Dataset-------------------------
---------------

from sklearn.datasets import load_iris

# Load the iris dataset
iris = load_iris()
X_df = pd.DataFrame(iris.data, columns=iris.feature_names)
y_df = pd.Series(iris.target, name="species")

# Function to run experiment for Iris dataset
def run_iris_experiment(test_size, label):
    print(f"\n=== Iris Dataset - {label} Split (Test size = {test_size}) ===")
    X_train, X_test, y_train, y_test = train_test_split(X_df, y_df, test_size=test_size)

    classifier = NaiveBayesClassifier()
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:\n", cm)
    accuracy = np.mean(y_pred == y_test)
    print("Accuracy:", accuracy)

# Run with 90-10 split
run_iris_experiment(test_size=0.1, label="90-10")

# Run with 70-30 split
run_iris_experiment(test_size=0.3, label="70-30")




# ---------------------------------------------------b)Titanic Dataset-------------------
-------------------------


# Load and preprocess Titanic dataset
df = pd.read_csv('titanic.csv')
df = df[['Survived', 'Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']]
df['Age'].fillna(df['Age'].median(), inplace=True)
df['Fare'].fillna(df['Fare'].median(), inplace=True)
df['Embarked'].fillna(df['Embarked'].mode()[0], inplace=True)
df['Embarked'] = df['Embarked'].map({'C': 0, 'Q': 1, 'S': 2})

# Function to run experiment with different splits
def run_titanic_experiment(test_size, label):
    print(f"\n=== {label} Split (Test size = {test_size}) ===")
    train, test = train_test_split(df, test_size=test_size)

    X_train = train.drop('Survived', axis=1)
    y_train = train['Survived']
    X_test = test.drop('Survived', axis=1)
    y_test = test['Survived']

    classifier = NaiveBayesClassifier()
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)

    cm = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:\n", cm)
    accuracy = np.mean(y_pred == y_test)
    print("Accuracy:", accuracy)

# Run with 90-10 split
run_titanic_experiment(test_size=0.1, label="90-10")

# Run with 70-30 split
run_titanic_experiment(test_size=0.3, label="70-30")
```

Sample Output:

```
=== Iris Dataset - 90-10 Split (Test size = 0.1) ===
Confusion Matrix:
 [[5 0 0]
  [0 3 0]
  [0 0 7]]
Accuracy: 1.0

=== Iris Dataset - 70-30 Split (Test size = 0.3) ===
Confusion Matrix:
 [[15  0  0]
  [ 0 15  0]
  [ 0  2 13]]
Accuracy: 0.9555555555555556

=== 90-10 Split (Test size = 0.1) ===
Confusion Matrix:
 [[48  6]
  [ 8 17]]
Accuracy: 0.725

=== 70-30 Split (Test size = 0.3) ===
Confusion Matrix:
 [[104  19]
  [ 25  31]]
Accuracy: 0.7528089887640449
```

7) Write a program to develop the KNN classifier for the k values as 3,5,7 based on split up of training and testing dataset as 90-10, 70-30,

a) Glass dataset

b) Fruit dataset

using the different distance metrics like Euclidean and Manhattan distance.

```
import numpy as np
import pandas as pd
from collections import Counter
from sklearn.model_selection import train_test_split

# Distance metrics
def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

def manhattan_distance(x1, x2):
    return np.sum(np.abs(x1 - x2))

# KNN Classifier
class KNN:
    def __init__(self, k, distance_metric):
        self.k = k
        self.distance_metric = distance_metric

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        return [self._predict(x) for x in X]

    def _predict(self, x):
        distances = [self.distance_metric(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]




# Experiment function
def run_knn_experiment(X, y, dataset_name, test_size, distance_metric, metric_name):
    print(f"\n--- {dataset_name} Dataset | Split: {int((1-test_size)*100)}-
{int(test_size*100)} | Distance: {metric_name} ---")
    for k in [3, 5, 7]:
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
random_state=42)
        model = KNN(k=k, distance_metric=distance_metric)
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)
        accuracy = np.sum(predictions == y_test) / len(y_test)
        print(f"k={k} | Accuracy: {accuracy:.4f}")
```

```python
# ---------------------- a) Glass Dataset ----------------------
glass_df = pd.read_csv('glass.csv')
X_glass = glass_df.drop('Type', axis=1).values
y_glass = glass_df['Type'].values

# Run experiments for Glass dataset
for test_size in [0.1, 0.3]:
    run_knn_experiment(X_glass, y_glass, dataset_name="Glass", test_size=test_size,
distance_metric=euclidean_distance, metric_name="Euclidean")
    run_knn_experiment(X_glass, y_glass, dataset_name="Glass", test_size=test_size,
distance_metric=manhattan_distance, metric_name="Manhattan")



# ---------------------- b) Fruit Dataset ----------------------
fruit_df = pd.read_csv('fruits.csv')
X_fruit = fruit_df[['mass', 'width', 'height', 'color_score']].values
y_fruit = fruit_df['fruit_label'].values

# Run experiments for Fruit dataset
for test_size in [0.1, 0.3]:
    run_knn_experiment(X_fruit, y_fruit, dataset_name="Fruit", test_size=test_size,
distance_metric=euclidean_distance, metric_name="Euclidean")
    run_knn_experiment(X_fruit, y_fruit, dataset_name="Fruit", test_size=test_size,
distance_metric=manhattan_distance, metric_name="Manhattan")
```

Sample Output:

```
--- Glass Dataset | Split: 90-10 | Distance: Euclidean ---
k=3 | Accuracy: 0.7209
k=5 | Accuracy: 0.6977
k=7 | Accuracy: 0.6977

--- Glass Dataset | Split: 90-10 | Distance: Manhattan ---
k=3 | Accuracy: 0.6977
k=5 | Accuracy: 0.6977
k=7 | Accuracy: 0.7209

--- Glass Dataset | Split: 70-30 | Distance: Euclidean ---
k=3 | Accuracy: 0.7385
k=5 | Accuracy: 0.7077
k=7 | Accuracy: 0.7077

--- Glass Dataset | Split: 70-30 | Distance: Manhattan ---
k=3 | Accuracy: 0.6923
k=5 | Accuracy: 0.7077
k=7 | Accuracy: 0.6923

--- Fruit Dataset | Split: 90-10 | Distance: Euclidean ---
```

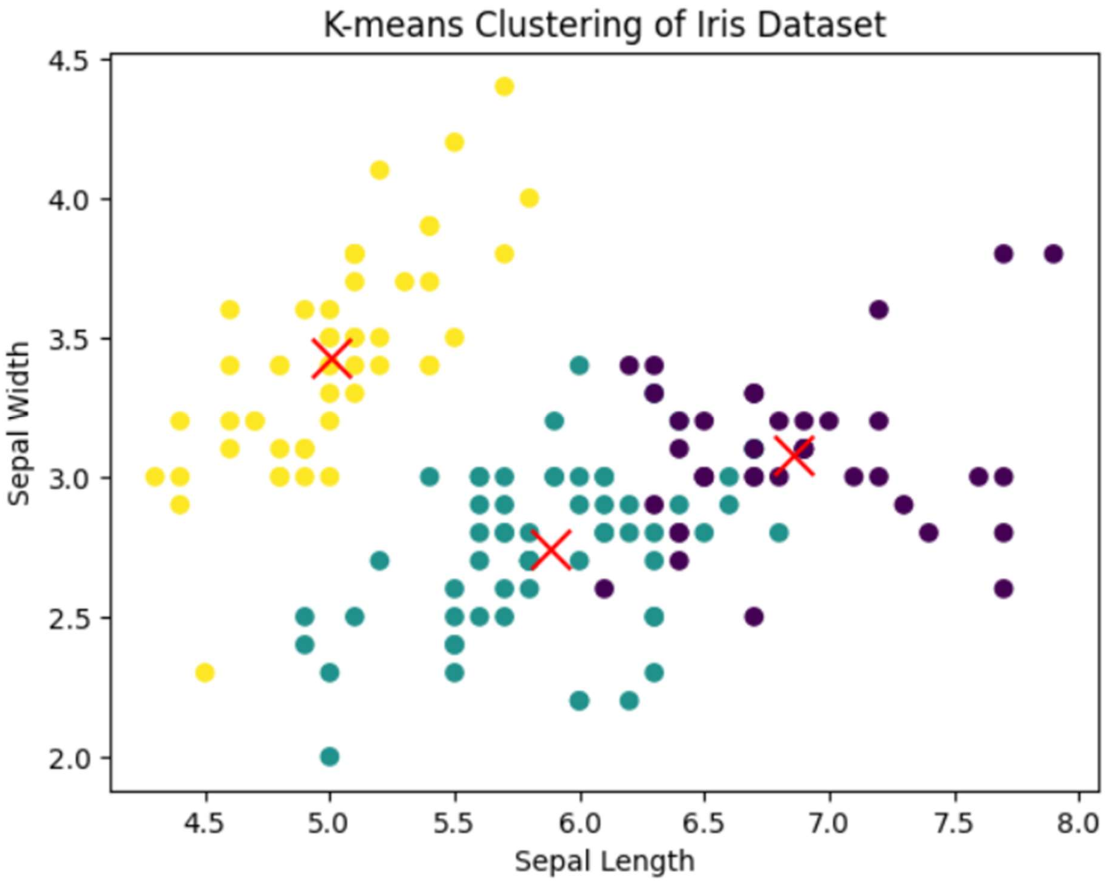
```
k=3 | Accuracy: 1.0000
k=5 | Accuracy: 1.0000
k=7 | Accuracy: 1.0000

--- Fruit Dataset | Split: 90-10 | Distance: Manhattan ---
k=3 | Accuracy: 1.0000
k=5 | Accuracy: 1.0000
k=7 | Accuracy: 1.0000

--- Fruit Dataset | Split: 70-30 | Distance: Euclidean ---
k=3 | Accuracy: 0.9333
k=5 | Accuracy: 0.9333
k=7 | Accuracy: 0.9333

--- Fruit Dataset | Split: 70-30 | Distance: Manhattan ---
k=3 | Accuracy: 0.9333
k=5 | Accuracy: 0.9333
k=7 | Accuracy: 0.9333
```

8) Write a program to perform unsupervised K-means clustering techniques
(pip install numpy matplotlib scikit-learn)

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

def kmeans(X, K, max_iters=100):
  centroids = X[:K]

  for _ in range(max_iters):
    # Assign each data point to the nearest centroid

    expanded_x = X[:, np.newaxis]
    euc_dist = np.linalg.norm(expanded_x - centroids, axis=2)
    labels = np.argmin(euc_dist, axis=1)

    # Update the centroids based on the assigned point
    new_centroids = np.array([X[labels == k].mean(axis=0) for k in range(K)])

    # If the centroids did not change, stop iterating
    if np.all(centroids == new_centroids):
      break

    centroids = new_centroids

  return labels, centroids


X = load_iris() .data
K=3
labels, centroids = kmeans(X, K)
print("Labels:", labels)
print("Centroids:", centroids)

plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', color='red', s=200)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('K-means Clustering of Iris Dataset')
plt.show()
```

Output:

```
Labels: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 1 0 0 0 0
 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0
 0 1]
Centroids: [[6.85384615 3.07692308 5.71538462 2.05384615]
 [5.88360656 2.74098361 4.38852459 1.43442623]
 [5.006      3.428      1.462      0.246     ]]
```

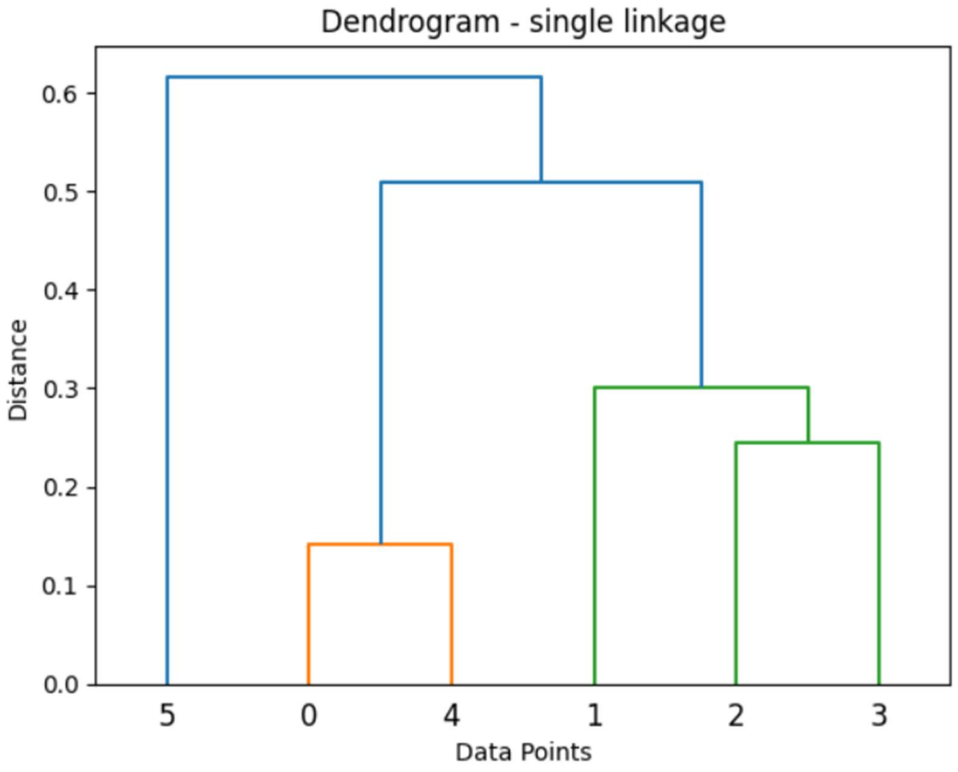


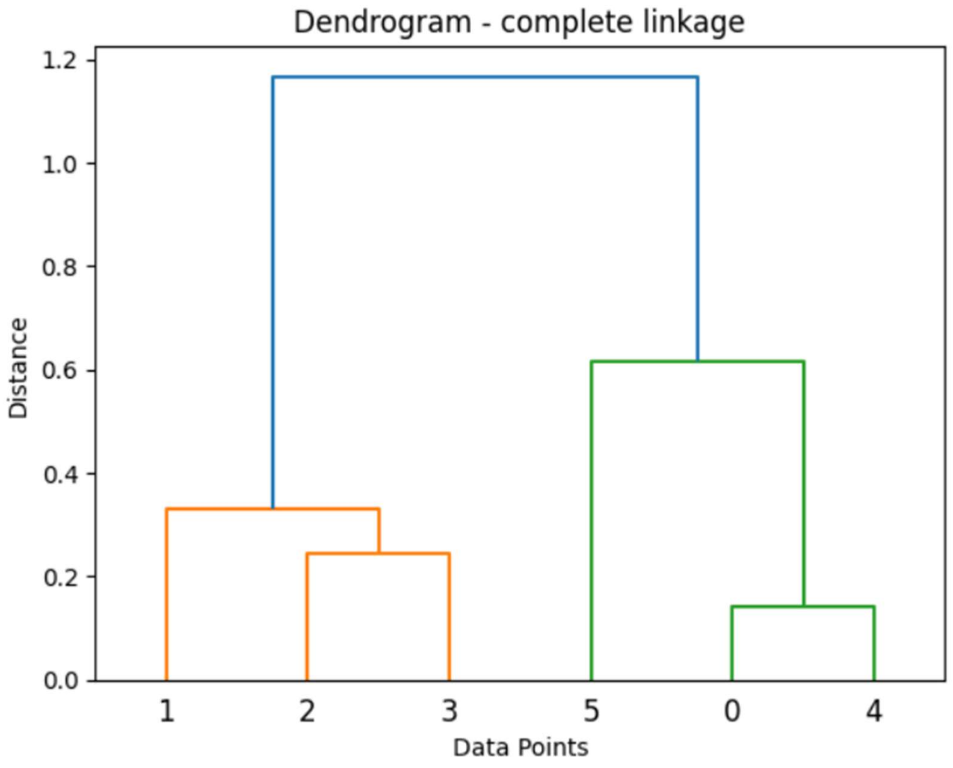
9) Write a program to perform agglomerative clustering based on single linkage, complete-linkage criteria.

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

iris = load_iris()
data = iris.data[:6]

def proximity_matrix(data):
  n = data.shape[0]
  proximity_matrix = np.zeros((n, n))
  for i in range(n):
    for j in range(i+1, n):
        proximity_matrix[i, j] = np.linalg.norm(data[i] - data[j])
        proximity_matrix[j, i] = proximity_matrix[i, j]
  return proximity_matrix

def plot_dendrogram(data, method):
  linkage_matrix = linkage(data, method=method)
  dendrogram(linkage_matrix)
  plt.title(f'Dendrogram - {method} linkage')
  plt.xlabel('Data Points')
  plt.ylabel('Distance')
  plt.show()

# Calculate the proximity matrix
print("Proximity matrix:")
print(proximity_matrix(data))

# Plot the dendrogram using single-linkage
plot_dendrogram(data, 'single')

# Plot the dendrogram using complete-linkage
plot_dendrogram(data, 'complete')
```

Output:

```
Proximity matrix:
[[0.         0.53851648 0.50990195 0.64807407 0.14142136 0.6164414 ]
 [0.53851648 0.         0.3        0.33166248 0.60827625 1.09087121]
 [0.50990195 0.3        0.         0.24494897 0.50990195 1.08627805]
 [0.64807407 0.33166248 0.24494897 0.         0.64807407 1.16619038]
 [0.14142136 0.60827625 0.50990195 0.64807407 0.         0.6164414 ]
 [0.6164414  1.09087121 1.08627805 1.16619038 0.6164414  0.        ]]
```

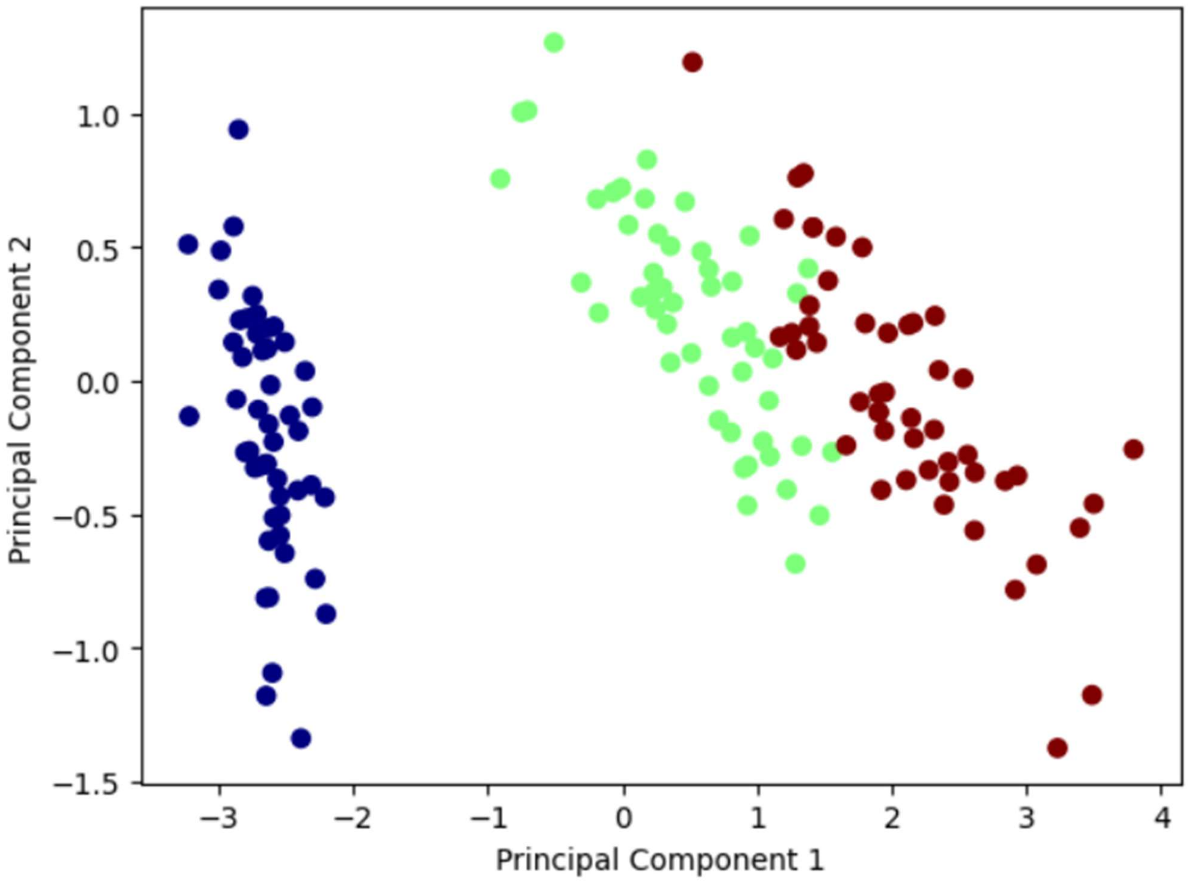




10) Write a program to develop Principal Component Analysis (PCA) algorithms.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

class PCA:

  def __init__(self, n_components):
    self.n_components = n_components
    self.components = None
    self.mean = None

  def fit(self, X):
    # Mean center the data
    self.mean = np.mean(X, axis=0)
    X = X - self.mean

    #Calculate covariance matrix
    cov = np.cov(X.T)

    #Calculate eigenvalues and eigen vectors
    eigenvalues, eigenvectors, = np.linalg.eig(cov)

    # Sort the vectors in decreasing order of eigenvalues
    eigenvectors = eigenvectors.T
    idxs = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idxs]
    eigenvectors = eigenvectors[idxs]

    # Take required number of components
    self.components = eigenvectors[:self.n_components]


  def transform(self, X):
    X = X - self.mean
    return np.dot(X, self.components.T)


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

pca = PCA(2)
pca.fit(X)
X_projected = pca.transform(X)

print("Shape of Data:", X.shape)
print("Shape of transformed Data:", X_projected.shape)

pc1 = X_projected[:, 0]
pc2 = X_projected[:, 1]

plt.scatter(pc1, pc2, c=y, cmap="jet")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

```
Shape of Data: (150, 4)
Shape of transformed Data: (150, 2)
```

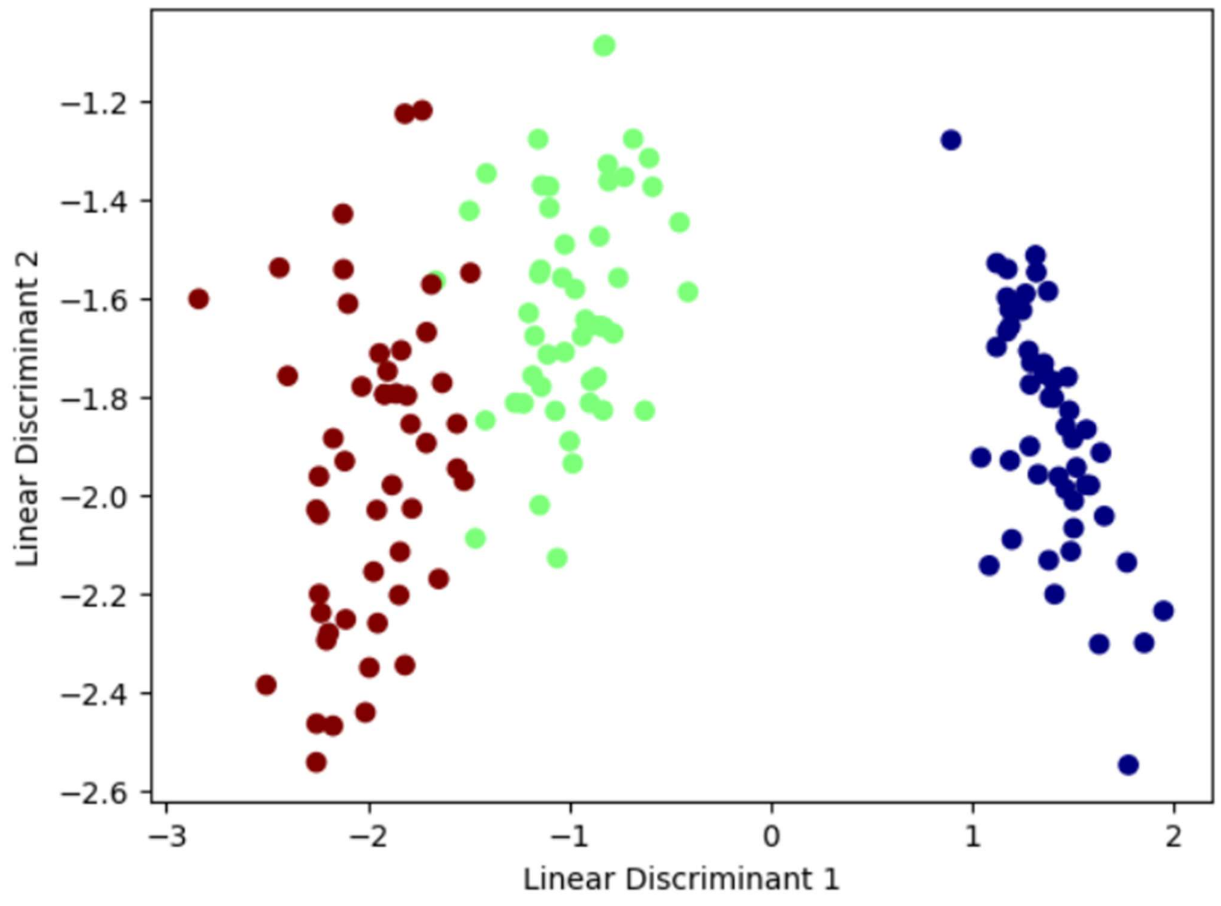


11) Write a program to develop Linear Discriminant Analysis (LDA) algorithms.

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

class LDA:
  def __init__(self, n_components):
    self.n_components = n_components
    self.linear_discriminants = None

  def fit(self, X, y):
    n_features = X.shape[1]
    class_labels = np.unique(y)

    # Calculate SB and SW
    mean_overall = np.mean(X, axis=0)
    SW = np.zeros((n_features, n_features))
    SB = np.zeros((n_features, n_features))

    for c in class_labels:
      X_c = X[y == c]
      mean_c = np.mean(X_c, axis=0)
      SW += (X_c - mean_c).T.dot((X_c - mean_c))

      n_c = X_c.shape[0]
      mean_diff = (mean_c - mean_overall).reshape(n_features, 1)
      SB += n_c * (mean_diff).dot(mean_diff.T)

    # Determine SW^-1 * SB
    A = np.linalg.inv(SW).dot(SB)

    #Calculate eigenvalues and eigen vectors
    eigenvalues, eigenvectors = np.linalg.eig(A)

    # Sort the vectors in decreasing order of eigenvalues
    eigenvectors = eigenvectors.T
    idxs = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idxs]
    eigenvectors = eigenvectors[idxs]

    # Take required number of components
    self.linear_discriminants = eigenvectors[:self.n_components]

  def transform(self, X):
    return np.dot(X, self.linear_discriminants.T)


X = load_iris().data
Y = load_iris().target

lda = LDA(2)
lda.fit(X, Y)
X_projected = lda.transform(X)
```

```
print("Shape of Data:", X.shape)
print("Shape of transformed Data:", X_projected.shape)

ld1 = X_projected[:, 0]
ld2 = X_projected[:, 1]

plt.scatter(ld1, ld2, c=Y, cmap="jet")
plt.xlabel("Linear Discriminant 1")
plt.ylabel("Linear Discriminant 2")

plt.show()
```

```
Shape of Data: (150, 4)
Shape of transformed Data: (150, 2)
```



12) Write a Program to develop simple single layer perceptron to implement AND, OR Boolean functions.

```python
import numpy as np

# Define the Sigmoid activation function and its derivative (for backpropagation)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

class Perceptron:
    def __init__(self, input_size):
        # Initialize the weights with random values and a bias term
        self.weights = np.random.rand(input_size)  # Random initialization of weights
        self.bias = np.random.rand(1)  # Random bias initialization

    def forward(self, inputs):
        # Weighted sum (dot product) + bias
        total_input = np.dot(inputs, self.weights) + self.bias
        # Apply the activation function (sigmoid)
        output = sigmoid(total_input)
        return output

    def train(self, X, y, epochs=1000, learning_rate=0.1):
        # Training the perceptron with the perceptron learning rule
        for epoch in range(epochs):
            for i in range(X.shape[0]):
                # Forward pass
                output = self.forward(X[i])
                # Calculate the error (difference between expected and predicted output)
                error = y[i] - output
                # Update the weights and bias using the perceptron learning rule
                self.weights += learning_rate * error * X[i]
                self.bias += learning_rate * error

# AND and OR dataset
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input for AND/OR functions
y_and = np.array([0, 0, 0, 1])  # Expected output for AND function
y_or = np.array([0, 1, 1, 1])  # Expected output for OR function

# Create perceptron instances for AND and OR
perceptron_and = Perceptron(input_size=2)
perceptron_or = Perceptron(input_size=2)

# Train the perceptrons
perceptron_and.train(X_and, y_and, epochs=1000, learning_rate=0.1)
perceptron_or.train(X_and, y_or, epochs=1000, learning_rate=0.1)

# Test the perceptrons
print("AND Function Predictions:")
for i in range(X_and.shape[0]):
    print(f"Input: {X_and[i]} - Predicted Output:
{round(perceptron_and.forward(X_and[i]))}")
```

```python
print("\nOR Function Predictions:")
for i in range(X_and.shape[0]):
    print(f"Input: {X_and[i]} - Predicted Output:
{round(perceptron_or.forward(X_and[i]))}")
```

```
AND Function Predictions:
Input: [0 0] - Predicted Output: 0
Input: [0 1] - Predicted Output: 0
Input: [1 0] - Predicted Output: 0
Input: [1 1] - Predicted Output: 1

OR Function Predictions:
Input: [0 0] - Predicted Output: 0
Input: [0 1] - Predicted Output: 1
Input: [1 0] - Predicted Output: 1
Input: [1 1] - Predicted Output: 1
```