

## INDEX

S. N o	Date	LIST OF EXPERIMENTS	Page No.	Marks	Sign
		A python program to implement univariate regression, bivariate regression and multivariate regression.			
		A python program to implement Simple linear regression using Least Square Method			
		A python program to implement logistic model.			
		A python program to implement single layer perceptron.			
		A python program to implement multi layer perceptron with back propagation			
		A python program to do face recognition using SVM classifier			
		A python program to implement decision tree.			
		A python program to implement boosting			
		A python program to implement KNN and K-means			
		A python program to implement dimensionality reduction – PCA.			
		Mini project – develop a simple application using tensorflow / keras.			
		Cbs:Traffic Sign Recognition Using CNN			

<b>Ex. No. 1</b>	<b>UNIVARIATE ,BIVARIATE AND MULTIVARIATE REGRESSION</b>

### **Aim**

To write a Python program for Univariate, Bivariate and Multivariate Regression.

### **Algorithm**

#### **Univariate Linear Regression**

1. Start
2. Import required libraries
3. Load or create the dataset
4. Select one independent variable X
5. Select dependent variable y
6. Create a Linear Regression model
7. Train the model using X and y
8. Predict y pred from X
9. Calculate performance metrics ( $R^2$ , MSE)
10. Display coefficients and performance
11. End

#### **Algorithm for Bivariate Linear Regression**

1. Start
2. Import required libraries
3. Load or create the dataset
4. Select two independent variables X1 and X2
5. Combine X1 and X2 into feature matrix X
6. Select dependent variable y
7. Create a Linear Regression model
8. Train the model using X and y
9. Predict y pred from X
10. Calculate performance metrics ( $R^2$ , MSE)
11. Display coefficients and performance
12. End

## Algorithm for Multivariate Linear Regression

1. Start
2. Import required libraries
3. Load or create the dataset
4. Select more than two independent variables  $X_1, X_2, \dots, X_n$
5. Combine all independent variables into feature matrix  $X$
6. Select dependent variable  $y$
7. Create a Linear Regression model
8. Train the model using  $X$  and  $y$
9. Predict  $y_{\text{pred}}$  from  $X$
10. Calculate performance metrics ( $R^2$ , MSE)
11. Display coefficients and performance
12. End

## Program

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

def evaluate_model(model, X, y, name="Model"):
    predictions = model.predict(X)
    print(f"\n{name} Performance:")
    print("Coefficients:", model.coef_)
    print("Intercept:", model.intercept_)
    print("MSE:", mean_squared_error(y, predictions))
    print("R2 Score:", r2_score(y, predictions))
    return predictions

print("=== Univariate Regression ===")
X_uni = np.random.rand(100, 1) * 10
y_uni = 3 * X_uni.squeeze() + 4 + np.random.randn(100)
model_uni = LinearRegression()
model_uni.fit(X_uni, y_uni)
pred_uni = evaluate_model(model_uni, X_uni, y_uni, "Univariate Regression")
plt.figure(figsize=(6, 4))
plt.scatter(X_uni, y_uni, color="blue", label="Actual")
```

```
plt.plot(X_uni, pred_uni, color="red", label="Predicted")
plt.title("Univariate Linear Regression")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()

print("\n=== Bivariate Regression ===")
X_bi = np.random.rand(100, 2) * 10
y_bi = 2 * X_bi[:, 0] + 5 * X_bi[:, 1] + 3 + np.random.randn(100)
model_bi = LinearRegression()
model_bi.fit(X_bi, y_bi)
pred_bi = evaluate_model(model_bi, X_bi, y_bi, "Bivariate Regression")
print("\n=== Multivariate Regression ===")

# Generate synthetic data
X_multi = np.random.rand(100, 5) * 10
# Create target with 5 predictors
y_multi = (1.5 * X_multi[:, 0] + 2.2 * X_multi[:, 1] +
0.5 * X_multi[:, 2] - 1.2 * X_multi[:, 3] +
3.3 * X_multi[:, 4] + 5 + np.random.randn(100))
model_multi = LinearRegression()
model_multi.fit(X_multi, y_multi)
pred_multi = evaluate_model(model_multi, X_multi, y_multi, "Multivariate Regression")
```

## Output

=== Univariate Regression ===

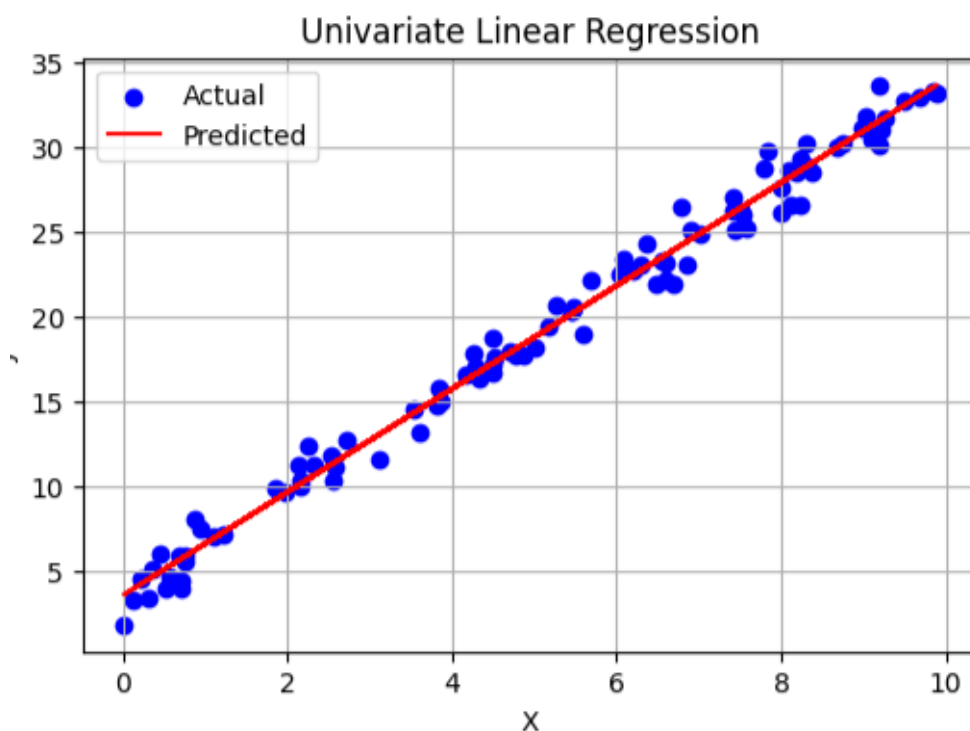
Univariate Regression Performance:

Coefficients: [3.04241453]

Intercept: 3.586891995302219

MSE: 0.9630107279246621

R2 Score: 0.9881435560730226



=== Bivariate Regression ===

Bivariate Regression Performance:

Coefficients: [2.01962213 4.96940366]

Intercept: 3.044137108247348

MSE: 1.0027759173747084

R2 Score: 0.9960031925945165

=== Multivariate Regression ===

Multivariate Regression Performance:

Coefficients: [ 1.48614347 2.24173582 0.51087548 -1.12956551 3.23143593]

Intercept: 4.86045813338108

MSE: 0.9578935289935879

R2 Score: 0.994364056625958

**Result**

Thus the python program for executing Univariate Regression ,Bivraite Regression and Multivariate Regression has been executed successfully.

**Aim**

To implement **Simple Linear Regression** using the **Least Squares Method** in Python and analyze the relationship between two variables.

**Algorithm**

- Start
- Import required libraries
- Input or load the dataset (X and Y values)
- Calculate the means of X and Y
- Compute slope (m) using the formula:

$$m = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{\sum(X_i - \bar{X})^2}$$

- Compute intercept (c) using:
- $$c = \bar{Y} - m\bar{X}$$
- Calculate predicted values ( $\hat{Y}$ ) for each X using:

$$\hat{Y} = mX + c$$

- Evaluate model using metrics like R<sup>2</sup> Score or Mean Squared Error (MSE)
- Display results (slope, intercept, predicted values, error metrics)
- End

**Program**

```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 5, 4, 5])
mean_x = np.mean(X)
mean_y = np.mean(y)
numerator = np.sum((X - mean_x) * (y - mean_y))
denominator = np.sum((X - mean_x) ** 2)
slope = numerator / denominator
intercept = mean_y - slope * mean_x
```

```
print("Slope (m):", slope)
print("Intercept (c):", intercept)
y_pred = slope * X + intercept
plt.scatter(X, y, color='blue', label='Actual data')
plt.plot(X, y_pred, color='red', label='Regression line')
plt.title('Simple Linear Regression (Least Squares)')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()

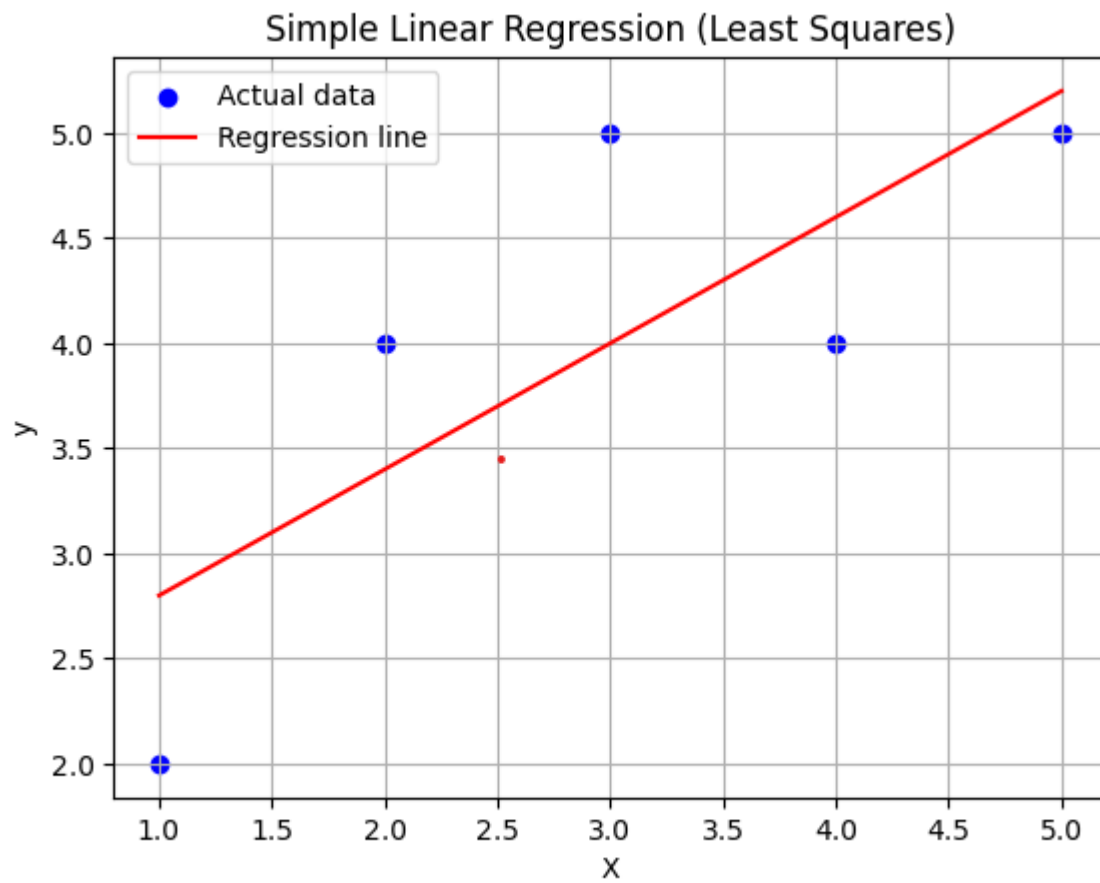
ss_total = np.sum((y - mean_y) ** 2)
ss_residual = np.sum((y - y_pred) ** 2)
r2_score = 1 - (ss_residual / ss_total)
print("R2 Score:", r2_score)
```

## Output

Slope (m): 0.6

Intercept (c): 2.2

Intercept (c): 2.2



$R^2$  Score: 0.6000000000000001

## **Result**

Thus the python program for to implement Simple Linear Regression using the Least Squares Method in python and analyze the relationship between two variables.

<b>Ex. No. 3</b>	<b>LOGISTIC MODEL</b>

### **Aim**

To implement a Logistic Regression Model in Python to classify binary outcomes and evaluate its performance using appropriate metrics.

### **Algorithm**

- Start
- Import required libraries
- Load or create the dataset with independent variables X and binary target variable y (0 or 1)
- Preprocess the data (if needed: clean, normalize, split)
- Split the dataset into training and testing sets
- Initialize the Logistic Regression model
- Train the model using the training data
- Predict the output for the test data
- Evaluate the model using metrics:
- Accuracy
- Confusion Matrix
- Precision, Recall, F1-score
- Display coefficients, intercept, and evaluation results
- End

### **Program**

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

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

class LogisticRegressionScratch:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        self.m, self.n = X.shape
```

```

self.weights = np.zeros(self.n)
self.bias = 0
for _ in range(self.epochs):
    linear_model = np.dot(X, self.weights) + self.bias
    y_pred = sigmoid(linear_model)
    dw = (1 / self.m) * np.dot(X.T, (y_pred - y))
    db = (1 / self.m) * np.sum(y_pred - y)
    self.weights -= self.learning_rate * dw
    self.bias -= self.learning_rate * db
def predict(self, X):
    linear_model = np.dot(X, self.weights) + self.bias
    y_pred = sigmoid(linear_model)
    return [1 if i > 0.5 else 0 for i in y_pred]
X, y = make_classification(n_samples=100, n_features=2, n_redundant=0,
                           n_clusters_per_class=1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = LogisticRegressionScratch(learning_rate=0.1, epochs=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy (scratch):", accuracy_score(y_test, y_pred))
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression()
clf.fit(X_train, y_train)
y_sklearn = clf.predict(X_test)
print("Accuracy (scikit-learn):", accuracy_score(y_test, y_sklearn))
def plot_decision_boundary(model, X, y):
    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.linspace(x_min, x_max, 100),
                          np.linspace(y_min, y_max, 100))
    grid = np.c_[xx.ravel(), yy.ravel()]
    preds = np.array(model.predict(grid)).reshape(xx.shape)
    plt.contourf(xx, yy, preds, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k')
    plt.title("Logistic Regression Decision Boundary")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")

```

```
plt.grid(True)
```

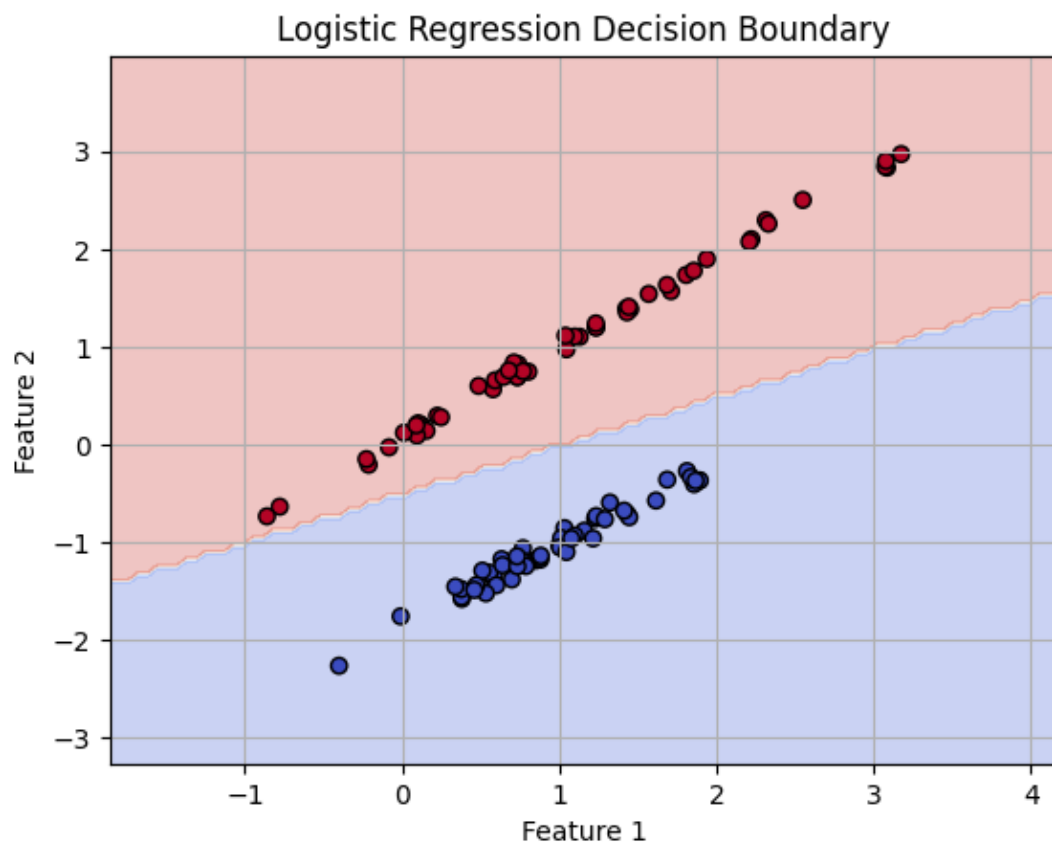
```
plt.show()
```

```
plot_decision_boundary(model, X, y)
```

## Output

Accuracy (scratch): 1.0

Accuracy (scikit-learn): 1.0



## **Result**

Thus the python program to implement a Logistic Regression Model in Python to classify binary outcomes and evaluate its performance using appropriate metrics

**Aim**

To implement a Single Layer Perceptron in Python for binary classification and demonstrate its learning using a simple linearly separable dataset.

**Algorithm**

1. Start
2. Import necessary libraries (e.g., NumPy)
3. Initialize input features X and target output y
4. Set learning rate ( $\alpha$ ), number of epochs, and initialize weights and bias to small random values or zeros
5. For each epoch (iteration over the entire dataset):
  - a. For each input sample:
    - i. Calculate the weighted sum:

$$z = w \cdot x + b$$

- ii. Apply the activation function (step function):

$$\text{output} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- iii. Update the weights and bias using the Perceptron learning rule:

$$w = w + \alpha(y - \hat{y})x$$
$$b = b + \alpha(y - \hat{y})$$

6. Repeat until convergence or max epochs reached
7. Test the perceptron on new inputs
8. Display final weights, bias, and outputs
9. End

**Program**

```

import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

def step_function(x):
    return np.where(x >= 0, 1, 0)

class Perceptron:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def fit(self, X, y):
        self.n_samples, self.n_features = X.shape
        self.weights = np.zeros(self.n_features)
        self.bias = 0

        for _ in range(self.epochs):
            for idx, x_i in enumerate(X):
                linear_output = np.dot(x_i, self.weights) + self.bias
                y_pred = step_function(linear_output)
                update = self.learning_rate * (y[idx] - y_pred)
                self.weights += update * x_i
                self.bias += update

    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        return step_function(linear_output)

X, y = make_classification(n_samples=100, n_features=2,
                           n_informative=2, n_redundant=0,
                           n_clusters_per_class=1, random_state=42)

y = np.where(y <= 0, 0, 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = Perceptron(learning_rate=0.01, epochs=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))

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

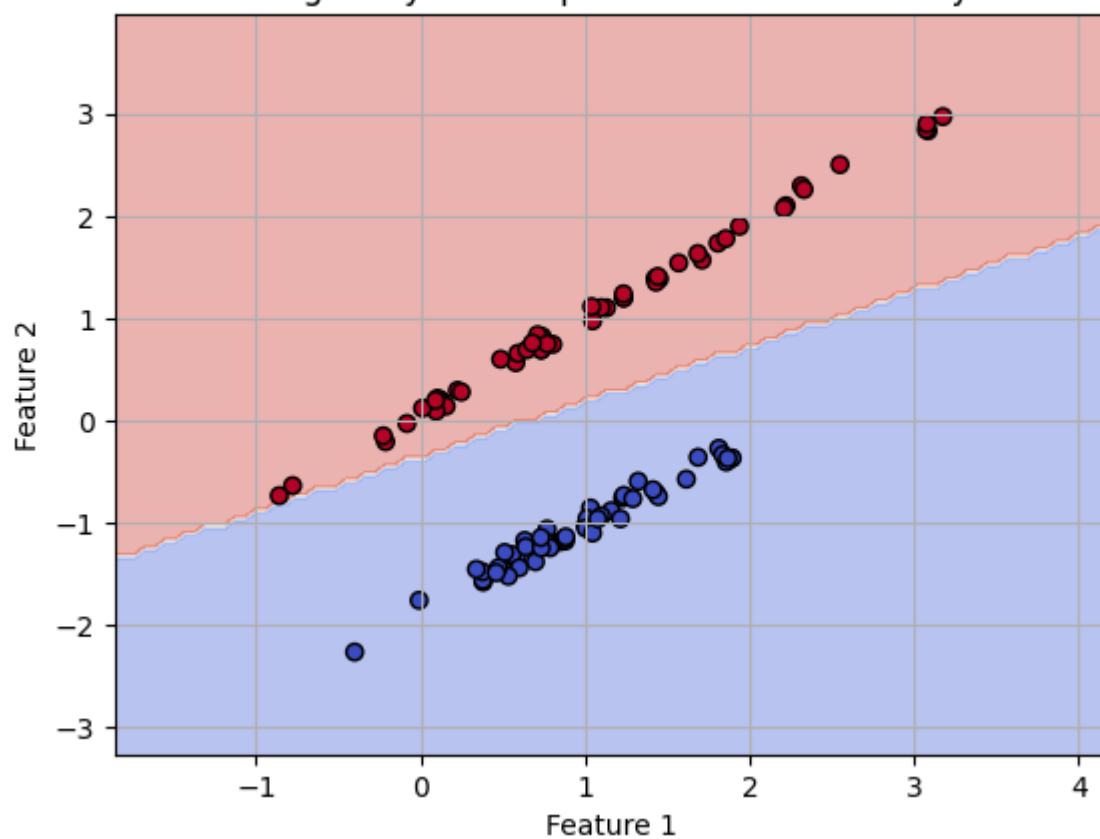
```

```
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                      np.linspace(y_min, y_max, 100))
grid = np.c_[xx.ravel(), yy.ravel()]
preds = model.predict(grid).reshape(xx.shape)
plt.contourf(xx, yy, preds, alpha=0.4, cmap='coolwarm')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm', edgecolors='k')
plt.title("Single-Layer Perceptron Decision Boundary")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.show()
plot_decision_boundary(X, y, model)
```

## Output

Accuracy: 1.0

Single-Layer Perceptron Decision Boundary



## **Result**

Thus the python program to implement a Single Layer Perceptron in Python for binary classification and demonstrate its learning using a simple linearly separable dataset has been executed successfully.

**Aim**

To implement a Multilayer Perceptron (MLP) using the Backpropagation algorithm in Python for solving classification problems.

**Algorithm**

- Start
- Import required libraries (e.g., NumPy or TensorFlow/Keras)
- Initialize the dataset (features X, labels y)
- Initialize network architecture:
  - Input layer size (based on input features)
  - One or more hidden layers with neurons
  - Output layer size (usually 1 for binary, n for multi-class)
  - Initialize weights and biases randomly for all layers
  - Set hyperparameters: learning rate  $\alpha$ , number of epochs, activation functions (e.g., sigmoid, ReLU)
- For each epoch:
  - For each training example:
    - a. Forward Propagation:
      - Compute activations for each layer using:
 
$$z = w \cdot x + b, \quad a = \text{activation}(z)$$
    - Compute Loss (e.g., Mean Squared Error or Cross Entropy)
    - c. Backward Propagation:
      - Compute gradients of the loss with respect to weights and biases (using chain rule)
      - Propagate the error backward layer by layer
    - d. Update Weights and Biases using:
 
$$w = w - \alpha \cdot \frac{\partial \text{Loss}}{\partial w}, \quad b = b - \alpha \cdot \frac{\partial \text{Loss}}{\partial b}$$
- Repeat until all epochs are completed or convergence
- Test the trained model on new data
- Display final weights, bias, and accuracy

- End

## Program

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

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases
        self.W1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size)
        self.b2 = np.zeros((1, output_size))

    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = sigmoid(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = sigmoid(self.Z2)
        return self.A2

    def backward(self, X, y, output, learning_rate):
        m = y.shape[0]
        error = output - y
        dZ2 = error * sigmoid_derivative(output)
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m
        dZ1 = np.dot(dZ2, self.W2.T) * sigmoid_derivative(self.A1)
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
```

```

def train(self, X, y, epochs=1000, learning_rate=0.1):
    for epoch in range(epochs):
        output = self.forward(X)
        self.backward(X, y, output, learning_rate)
        if epoch % 100 == 0:
            loss = np.mean((y - output) ** 2)
            print(f'Epoch {epoch}, Loss: {loss:.4f}')
    def predict(self, X):
        output = self.forward(X)
        return (output > 0.5).astype(int)

X, y = make_moons(n_samples=500, noise=0.2, random_state=42)
y = y.reshape(-1, 1) # Make y 2D for output layer compatibility
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
mlp = MLP(input_size=2, hidden_size=5, output_size=1)
mlp.train(X_train, y_train, epochs=1000, learning_rate=0.1)
y_pred = mlp.predict(X_test)
accuracy = np.mean(y_pred == y_test)
print(f'\nTest Accuracy: {accuracy:.2f}')

def plot_decision_boundary(model, X, y):
    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                          np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]
    preds = model.predict(grid).reshape(xx.shape)
    plt.contourf(xx, yy, preds, alpha=0.3, cmap="coolwarm")
    plt.scatter(X[:, 0], X[:, 1], c=y.ravel(), cmap="coolwarm", edgecolors='k')
    plt.title("MLP Decision Boundary")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.grid(True)
    plt.show()
plot_decision_boundary(mlp, X, y)

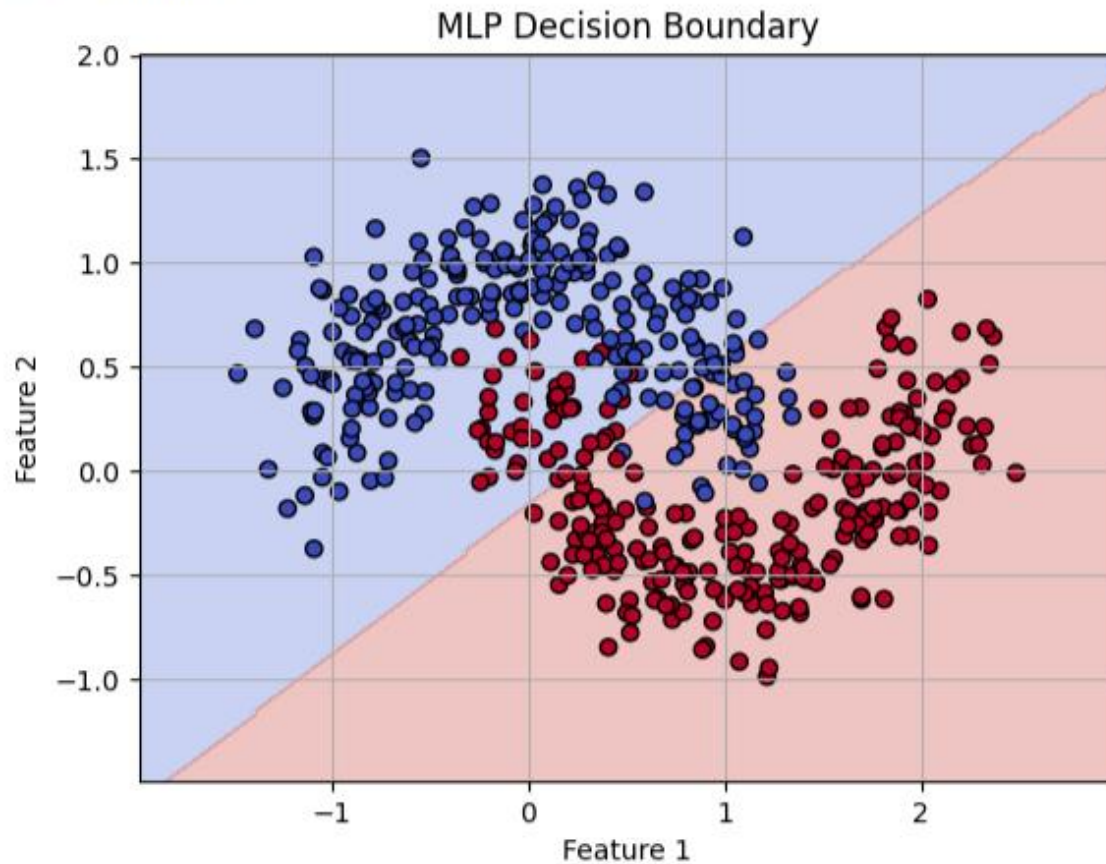
```

## Output

Matplotlib is building the font cache; this may take a moment.

Epoch 0, Loss: 0.3359  
Epoch 100, Loss: 0.2293  
Epoch 200, Loss: 0.2013  
Epoch 300, Loss: 0.1865  
Epoch 400, Loss: 0.1739  
Epoch 500, Loss: 0.1631  
Epoch 600, Loss: 0.1541  
Epoch 700, Loss: 0.1465  
Epoch 800, Loss: 0.1402  
Epoch 900, Loss: 0.1348

Test Accuracy: 0.86



## **Result**

Thus the python program to implement a Multilayer Perceptron (MLP) using the Backpropagation algorithm in Python for solving classification problems.

<b>Ex. No. 6</b>	<b>FACE RECOGNITION USING SVM CLASSIFIER</b>

### **Aim**

To implement a Face Recognition System using a Support Vector Machine (SVM) classifier in Python and classify facial images based on their encoded features.

### **Algorithm**

- Start
- Import required libraries:
  - face\_recognition for face detection and encoding
  - os for directory/file handling
  - sklearn.svm.SVC for SVM model
  - cv2 (OpenCV) for webcam/image capture (optional)
- Load dataset of face images (organized into subfolders, one per person)
- For each image in the dataset:
  - a. Load the image
  - b. Detect the face in the image
  - c. Compute the face encoding (128-d feature vector)
  - d. Store the encoding with the corresponding label (person's name)
- Split the data into training and testing sets (optional)
- Train the SVM classifier on the face encodings and labels
- Test the model by:
  - a. Capturing or loading a new image
  - b. Detecting and encoding the face
  - c. Predicting the identity using the trained SVM classifier
- Display the result (predicted name with confidence)
- End

### **Program**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.decomposition import PCA
```

```

from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

X = lfw_people.data      # Flattened image data
y = lfw_people.target     # Target labels
target_names = lfw_people.target_names
n_classes = target_names.shape[0]
print("Total samples:", X.shape[0])
print("Image shape:", lfw_people.images[0].shape)
print("Number of classes:", n_classes)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
n_components = 100

pca = PCA(n_components=n_components, whiten=True).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

clf = SVC(kernel='rbf', class_weight='balanced', C=1000, gamma=0.001)
clf.fit(X_train_pca, y_train)
y_pred = clf.predict(X_test_pca)
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=target_names))

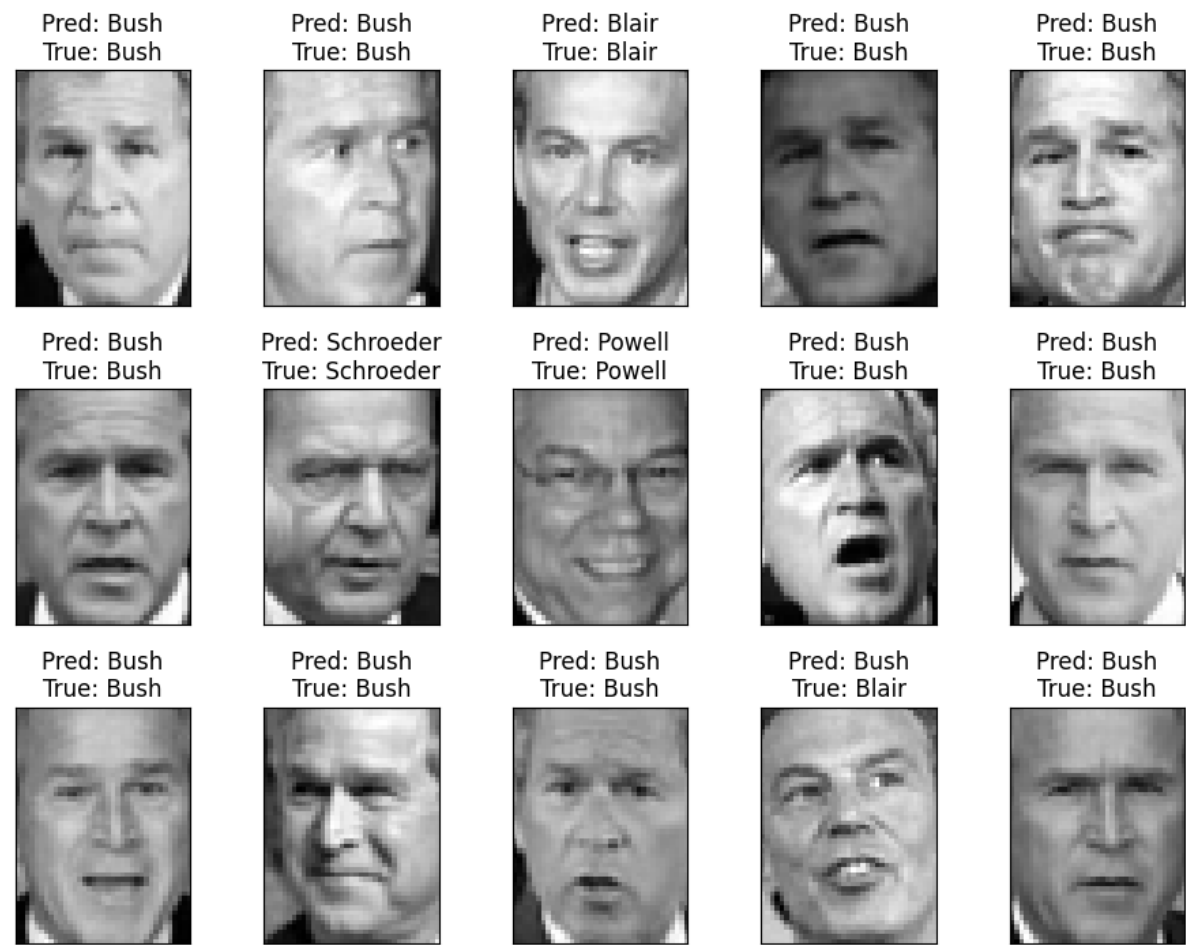
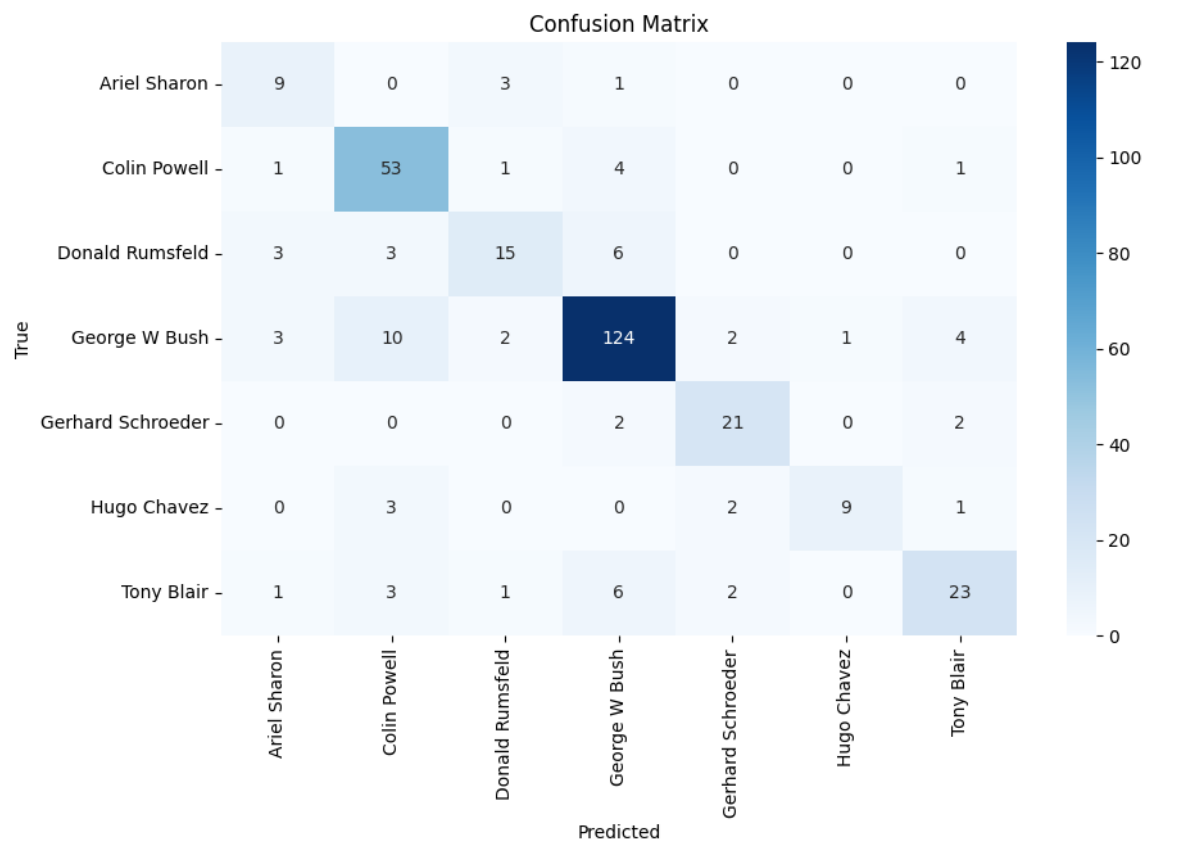
plt.figure(figsize=(10, 6))
conf_mat = confusion_matrix(y_test, y_pred, labels=range(n_classes))
sns.heatmap(conf_mat, annot=True, fmt="d", cmap="Blues",
            xticklabels=target_names, yticklabels=target_names)
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

def plot_gallery(images, titles, h, w, n_row=3, n_col=5):
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0.01, left=0.01, right=0.99, top=0.90, hspace=0.35)
    for i in range(n_row * n_col):
        plt.subplot(n_row, n_col, i + 1)
        plt.imshow(images[i].reshape((h, w)), cmap=plt.cm.gray)
        plt.title(titles[i], size=12)
        plt.xticks(())
        plt.yticks(())

```

```
def title(y_pred, y_true, target_names, i):  
    pred_name = target_names[y_pred[i]].split()[-1]  
    true_name = target_names[y_true[i]].split()[-1]  
    return f'Pred: {pred_name}\nTrue: {true_name}'  
prediction_titles = [title(y_pred, y_test, target_names, i) for i in range(y_pred.shape[0])]  
plot_gallery(X_test, prediction_titles, h=lfw_people.images.shape[1], w=lfw_people.images.shape[2])  
plt.show()
```

Output



## **Result**

Thus the program to implement a Face Recognition System using a Support Vector Machine (SVM) classifier in Python and classify facial images based on their encoded features has been executed successfully.

<b>Ex. No. 7</b>	<b>DECISION TREE</b>

### **Aim**

To implement a Decision Tree Classifier in Python using a standard dataset and evaluate its performance using classification metrics.

### **Algorithm**

1. Start
2. Import necessary libraries (sklearn, pandas, numpy, etc.)
3. Load the dataset (e.g., Iris, Breast Cancer, or any CSV file)
4. Split the dataset into input features X and target labels y
5. Preprocess the data if necessary (e.g., encoding, normalization)
6. Divide the dataset into training and testing sets using train\_test\_split()
7. Initialize the Decision Tree Classifier from sklearn.tree
8. Train the model using the training data
9. Make predictions using the test data
10. Evaluate the model using metrics like:
  - Accuracy
  - Confusion matrix
  - Classification report
11. Visualize the tree (optional, using plot\_tree)
12. Display the results
13. End

### **Program**

```

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

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)

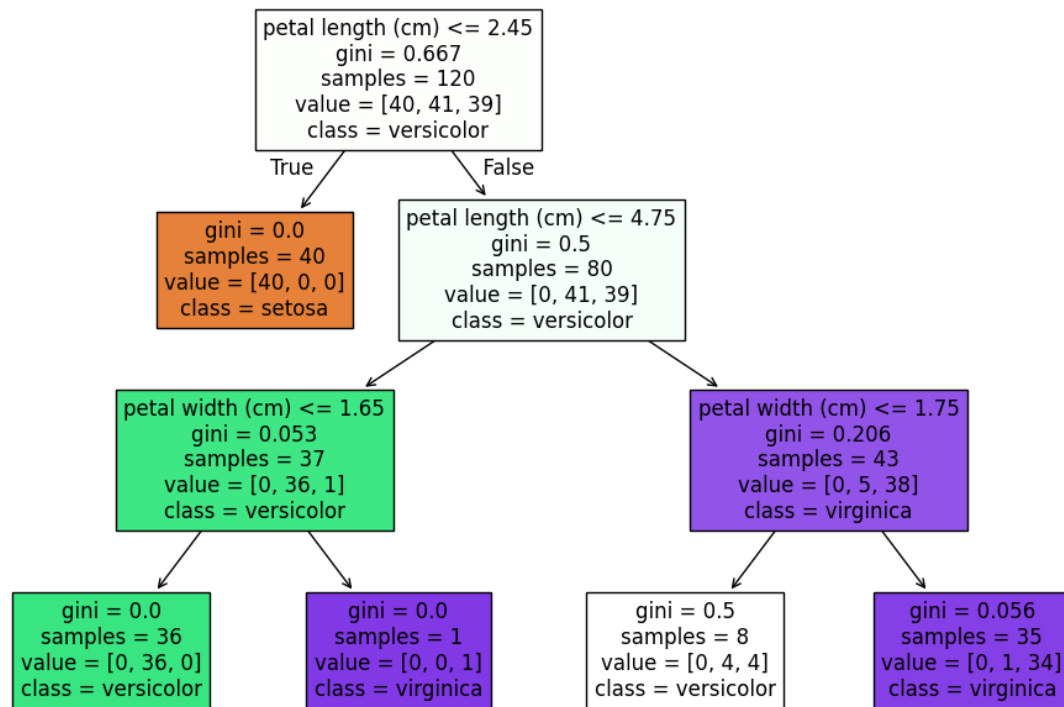
```

```
clf = DecisionTreeClassifier(criterion='gini', max_depth=3, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
plt.figure(figsize=(12,8))
plot_tree(clf, feature_names=iris.feature_names, class_names=iris.target_names, filled=True)
plt.title("Decision Tree Visualization")
plt.show()
```

## Output

Accuracy: 1.00

Decision Tree Visualization



## **Result**

Thus the python program to implement a Decision Tree Classifier in Python using a standard dataset and evaluate its performance using classification metrics has been executed successfully.

<b>Ex. No. 8</b>	<b>BOOSTING</b>

### Aim

To implement the AdaBoost (Adaptive Boosting) algorithm using Decision Stump (a one-level Decision Tree) as the weak learner and evaluate its performance on a synthetic classification dataset..

### Algorithm

- Start the process.
- Generate or load a classification dataset.
- Split the dataset into training and testing sets.
- Create a decision stump using a decision tree classifier with max depth set to 1.
- Initialize the AdaBoost classifier using the decision stump as the base estimator, along with parameters like number of estimators and learning rate.
- Train the AdaBoost model on the training data.
- Use the trained model to predict the output on the test data.
- Evaluate the model using accuracy score, confusion matrix, and classification report.
- Plot the feature importances learned by the model.
- End the process.

### Program

```

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt

X, y = make_classification(n_samples=1000, n_features=10,
                          n_informative=5, n_redundant=2,
                          random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.3,
                                                    random_state=42)

base_estimator = DecisionTreeClassifier(max_depth=1)
model = AdaBoostClassifier(estimator=base_estimator, # <- changed here
                          n_estimators=50,
                          learning_rate=1.0,
                          random_state=42)

model.fit(X_train, y_train)

```

```
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
plt.figure(figsize=(10, 6))
plt.bar(range(X.shape[1]), model.feature_importances_)
plt.xlabel("Feature Index")
plt.ylabel("Importance")
plt.title("Feature Importances from AdaBoost")
plt.show()
```

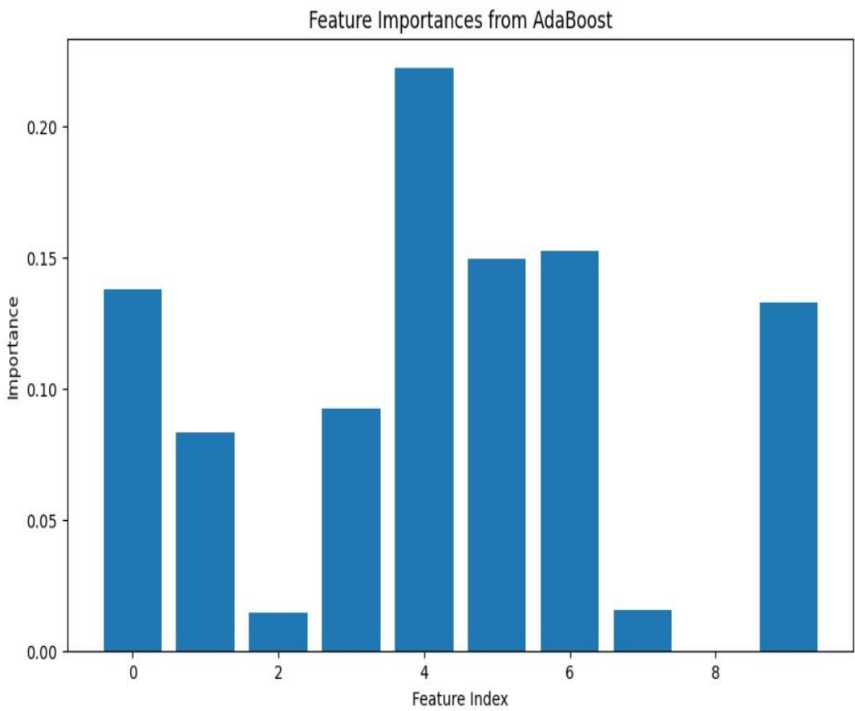
# Output

Accuracy: 0.9033333333333333

Confusion Matrix:  
[[146 12]  
[ 17 125]]

Classification Report:

	precision	recall	f1-score	support
0	0.90	0.92	0.91	158
1	0.91	0.88	0.90	142
accuracy			0.90	300
macro avg	0.90	0.90	0.90	300
weighted avg	0.90	0.90	0.90	300



## **Result**

Thus the program to implement a Boosting algorithm (e.g., AdaBoost) in Python using a standard classification dataset and evaluate its performance.

**Aim**

To implement the K-Nearest Neighbors (KNN) algorithm for classification and K-means clustering using a standard labeled dataset and evaluate its accuracy.

**Algorithm****KNN**

- Start
- Import required libraries (numpy, sklearn, pandas)
- Load the labeled dataset (e.g., Iris)
- Preprocess the data if required (e.g., normalization)
- Split the dataset into training and testing sets
- Choose the value of k (number of neighbors)
- For each test data point:
  - a. Compute the distance from the test point to all training points
  - b. Select the k nearest neighbors
  - c. Assign the class most common among the neighbors
- Evaluate the model using accuracy or confusion matrix
- End

**Program****KNN**

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

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print("KNN Accuracy:", accuracy_score(y_test, y_pred))
```

## Output

KNN Accuracy: 1.0

## K means Clustering

### Algorithm

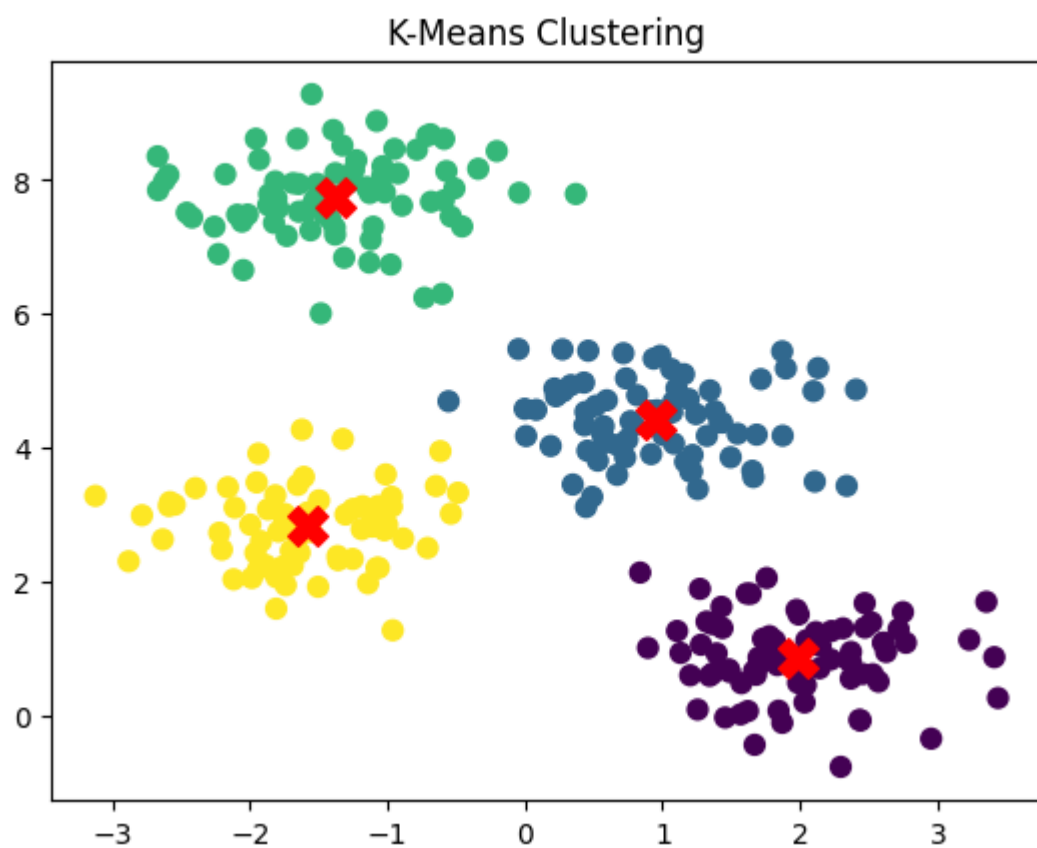
- Start
- Import required libraries (numpy, sklearn, matplotlib)
- Load the unlabeled dataset (or remove labels from a labeled dataset)
- Select the number of clusters k
- Randomly initialize k cluster centroids
- Repeat until convergence or max iterations:
  - a. Assign each data point to the nearest centroid
  - b. Recalculate centroids as the mean of assigned points
- Visualize the clusters (if 2D or 3D)
- End

### Program

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)
y_kmeans = kmeans.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_kmeans, s=50, cmap='viridis')
plt.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1],
            s=200, c='red', marker='X')
plt.title("K-Means Clustering")
plt.show()
```

## Output

Matplotlib is building the font cache; this may take a moment.



## **Result**

Thus the program to implement the K-Nearest Neighbors (KNN) algorithm for classification and K-means clustering using a standard labeled dataset and evaluate its accuracy has been executed successfully.

**Aim**

To implement Principal Component Analysis (PCA) for dimensionality reduction and visualize the transformed dataset in reduced dimensions

**Algorithm**

- Start
- Import required libraries (numpy, pandas, matplotlib, sklearn)
- Load the dataset (e.g., Iris, or any high-dimensional data)
- Preprocess the data (handle missing values, normalize if needed)
- Standardize the features (mean = 0, std = 1)
- Compute the covariance matrix of the features
- Calculate eigenvalues and eigenvectors of the covariance matrix
- Sort eigenvectors by decreasing eigenvalues
- Select top k eigenvectors to form the projection matrix
- Transform the original dataset into the new subspace using the projection matrix
- Visualize the reduced data (e.g., in 2D using a scatter plot)
- End

**Program**

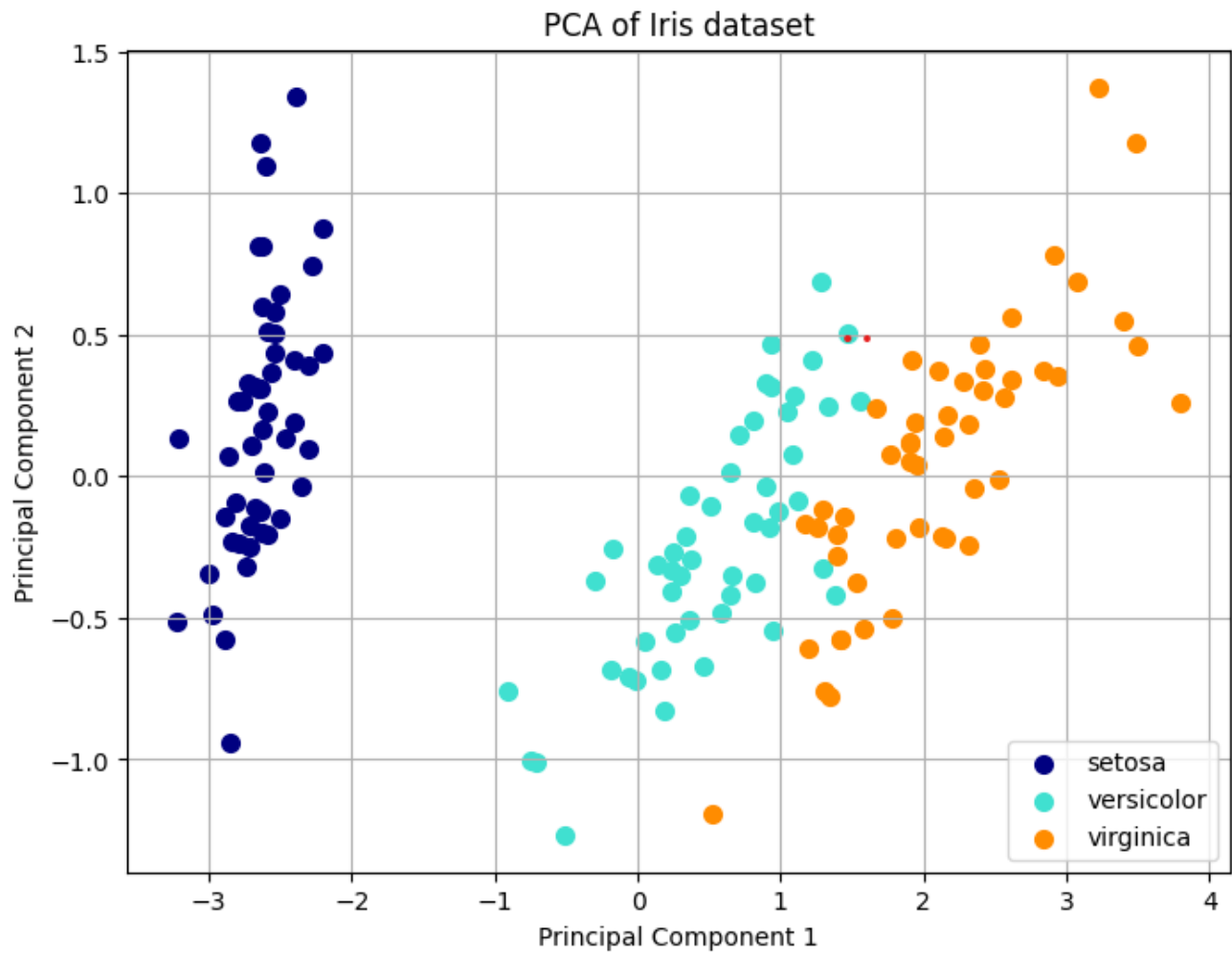
```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names
pca = PCA(n_components=2)
X_r = pca.fit_transform(X)
print(f"Explained variance ratio of the 2 components: {pca.explained_variance_ratio_}")
plt.figure(figsize=(8,6))
colors = ['navy', 'turquoise', 'darkorange']
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], color=color, lw=2, label=target_name)
```

```
plt.legend()  
plt.title('PCA of Iris dataset')  
plt.xlabel('Principal Component 1')  
plt.ylabel('Principal Component 2')  
plt.grid(True)  
plt.show()
```

## Output

Matplotlib is building the font cache; this may take a moment.

Explained variance ratio of the 2 components: [0.92461872 0.05306648]



## **Result**

Thus the python program to implement Principal Component Analysis (PCA) for dimensionality reduction and visualize the transformed dataset in reduced dimensions has been executed successfully

**Aim**

To design and implement a Convolutional Neural Network (CNN) using Python and TensorFlow/Keras to recognize and classify traffic signs from the GTSRB dataset, thereby assisting in autonomous driving and traffic monitoring systems.

**Algorithm**

Import necessary libraries.

- ☐ Set the dataset path.
- ☐ Initialize empty lists for images and labels.
- ☐ For each class folder from 0 to 42:
  - a. Read each image.
  - b. Resize the image to 32×32.
  - c. Append image to the image list.
  - d. Append the corresponding label to the label list.
- ☐ Convert image and label lists to NumPy arrays.
- ☐ Normalize the image data.
- ☐ One-hot encode the labels.
- ☐ Split the dataset into training and testing sets.
- ☐ Define the CNN model:
  - a. Add Conv2D layer.
  - b. Add MaxPooling2D layer.
  - c. Add another Conv2D layer.
  - d. Add another MaxPooling2D layer.
  - e. Add Flatten layer.
  - f. Add Dense layer with ReLU activation.
  - g. Add Dropout layer.
  - h. Add output Dense layer with softmax activation.
- ☐ Compile the model.
- ☐ Train the model using training data.
- ☐ Evaluate the model using test data.
- ☐ Save the trained model.
- ☐ Predict class for a new image.
- ☐ Plot training and validation accuracy.

## Program

```
import numpy as np
import pandas as pd
import os
import cv2
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
data_dir = './gtsrb/Train' # Update if different
num_classes = 43
image_data = []
labels = []
print("Loading images...")
for class_id in range(num_classes):
    class_path = os.path.join(data_dir, str(class_id))
    if not os.path.exists(class_path):
        continue
    for img_file in os.listdir(class_path):
        try:
            img_path = os.path.join(class_path, img_file)
            img = cv2.imread(img_path)
            img = cv2.resize(img, (32, 32))
            image_data.append(img)
            labels.append(class_id)
        except:
            continue
X = np.array(image_data)
y = np.array(labels)
X = X / 255.0
y = to_categorical(y, num_classes)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
model = Sequential([
```

```

Conv2D(32, (3,3), activation='relu', input_shape=(32,32,3)),
MaxPooling2D(pool_size=(2,2)),

Conv2D(64, (3,3), activation='relu'),
MaxPooling2D(pool_size=(2,2)),
Flatten(),
Dense(128, activation='relu'),
Dropout(0.5),
Dense(num_classes, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.summary()
print("Training model...")
history = model.fit(X_train, y_train, epochs=10, batch_size=64, validation_data=(X_test, y_test))
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
model.save("traffic_sign_cnn.h5")
print("Model saved as traffic_sign_cnn.h5")
plt.plot(history.history['accuracy'], label='Training accuracy')
plt.plot(history.history['val_accuracy'], label='Validation accuracy')
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
def predict_sign(image_path):
    img = cv2.imread(image_path)
    img = cv2.resize(img, (32, 32))
    img = img / 255.0
    img = img.reshape(1, 32, 32, 3)
    prediction = model.predict(img)
    class_index = np.argmax(prediction)
    confidence = np.max(prediction)

```

```
print(f"Predicted Class: {class_index}, Confidence: {confidence:.2f}")
```

```
# predict_sign('./gtsrb/Test/00014.png')
```

## Output

Loading images...

Total images loaded: 39209

Training model...

Epoch 1/10

492/492 [=====] - 12s 23ms/step - loss: 1.6132 - accuracy: 0.5724 -  
val\_loss: 0.5310 - val\_accuracy: 0.8621

Epoch 2/10

492/492 [=====] - 11s 23ms/step - loss: 0.4605 - accuracy: 0.8720 -  
val\_loss: 0.2906 - val\_accuracy: 0.9265

Epoch 3/10

492/492 [=====] - 11s 23ms/step - loss: 0.2938 - accuracy: 0.9195 -  
val\_loss: 0.2062 - val\_accuracy: 0.9462

Epoch 4/10

492/492 [=====] - 11s 23ms/step - loss: 0.2127 - accuracy: 0.9423 -  
val\_loss: 0.1598 - val\_accuracy: 0.9584

Epoch 5/10

492/492 [=====] - 11s 23ms/step - loss: 0.1683 - accuracy: 0.9543 -  
val\_loss: 0.1304 - val\_accuracy: 0.9647

Epoch 6/10

492/492 [=====] - 11s 22ms/step - loss: 0.1380 - accuracy: 0.9617 -  
val\_loss: 0.1182 - val\_accuracy: 0.9675

Epoch 7/10

492/492 [=====] - 11s 22ms/step - loss: 0.1141 - accuracy: 0.9677 -  
val\_loss: 0.1079 - val\_accuracy: 0.9701

Epoch 8/10

492/492 [=====] - 11s 22ms/step - loss: 0.0977 - accuracy: 0.9718 -  
val\_loss: 0.1044 - val\_accuracy: 0.9707

Epoch 9/10

492/492 [=====] - 11s 23ms/step - loss: 0.0877 - accuracy: 0.9744 -  
val\_loss: 0.0990 - val\_accuracy: 0.9721

Epoch 10/10

492/492 [=====] - 11s 23ms/step - loss: 0.0779 - accuracy: 0.9772 -  
val\_loss: 0.0938 - val\_accuracy: 0.9733

Evaluating on test set...

Test Accuracy: 0.9733

Model saved as traffic\_sign\_cnn.h5

## **Result**

Thus the python program to implement a Convolutional Neural Network (CNN) using TensorFlow/Keras to recognize and classify traffic signs from the GTSRB dataset, thereby assisting in autonomous driving and traffic monitoring systems has been executed successfully.