EXP NO. 01 **DATE:** 24.01.2025	**Univariate, Bivariate and Multivariate Regression**

AIM:

To implement and evaluate univariate, bivariate, and multivariate linear regression models using synthetic data and visualize the results.

ALGORITHM:

Step 0: Start the program.

Step 1: Load essential libraries (NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn).

Step 2: Initialize a random seed to ensure consistent results.

Step 3: Create synthetic datasets for single-variable, two-variable, and multi-variable regression analysis.

Step 4: Construct the dependent variable based on a linear formula with some random noise added.

Step 5: Train a Linear Regression model using the generated dataset.

Step 6: Generate predictions by applying the trained model to the input features.

Step 7: Plot scatter diagrams and 3D graphs to compare real values against predicted values.

Step 8: Compute and present evaluation metrics (Mean Squared Error and R^2 Score).

Step 9: End the program.

SOURCE CODE:

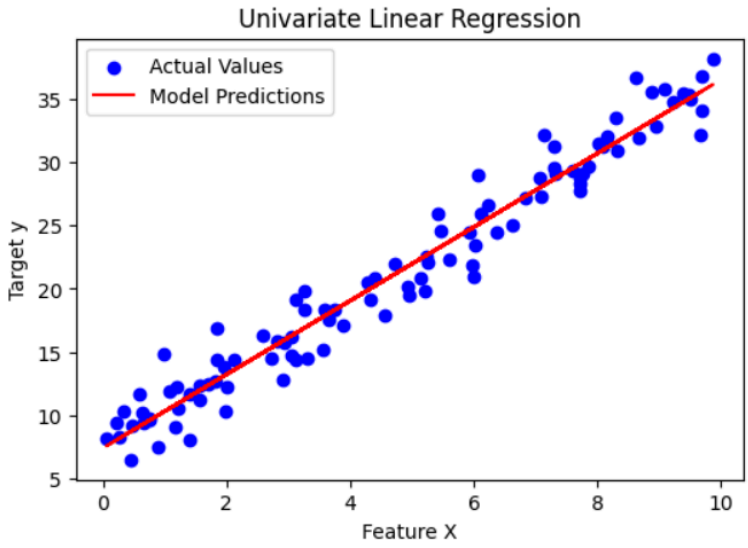
```
# Start the program
```

```
# Import required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from mpl_toolkits.mplot3d import Axes3D

# Initialize random seed for reproducibility
np.random.seed(42)

# --- 1. UNIVARIATE REGRESSION ---
# Generate synthetic data
X_uni = np.random.rand(100, 1) * 10
y_uni = 3 * X_uni.squeeze() + 7 + np.random.randn(100) * 2

# Train Linear Regression model
model_uni = LinearRegression().fit(X_uni, y_uni)
y_uni_pred = model_uni.predict(X_uni)

# Visualize results
plt.figure(figsize=(6,4))
plt.scatter(X_uni, y_uni, label="Actual Values", color="blue")
plt.plot(X_uni, y_uni_pred, label="Model Predictions", color="red")
plt.title("Univariate Linear Regression")
plt.xlabel("Feature X")
plt.ylabel("Target y")
plt.legend()
```

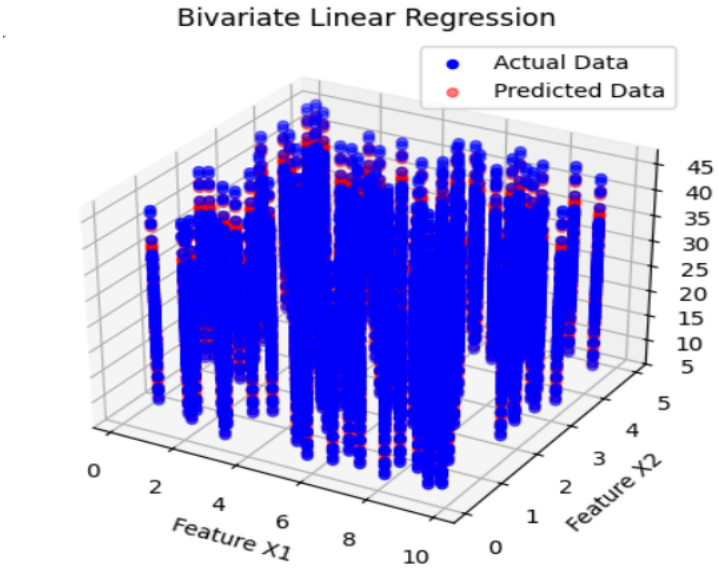
```
plt.show()

# Evaluate model performance
print("Univariate Regression Results:")
print("Mean Squared Error (MSE):", mean_squared_error(y_uni, y_uni_pred))
print("R² Score:", r2_score(y_uni, y_uni_pred))
print()

# --- 2. BIVARIATE REGRESSION ---
# Generate synthetic data
X1 = np.random.rand(100, 1) * 10
X2 = np.random.rand(100, 1) * 5
X_bi = np.hstack((X1, X2))
y_bi = 2 * X1.squeeze() + 4 * X2.squeeze() + 5 + np.random.randn(100) * 2

# Train Linear Regression model
model_bi = LinearRegression().fit(X_bi, y_bi)
y_bi_pred = model_bi.predict(X_bi)

# Visualize results in 3D
fig = plt.figure(figsize=(7,5))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X1, X2, y_bi, color='blue', label='Actual Data')
ax.scatter(X1, X2, y_bi_pred, color='red', label='Predicted Data', alpha=0.5)
ax.set_xlabel("Feature X1")
ax.set_ylabel("Feature X2")
ax.set_zlabel("Target y")
ax.set_title("Bivariate Linear Regression")
ax.legend()
```

```
plt.show()

# Evaluate model performance
print("Bivariate Regression Results:")
print("Mean Squared Error (MSE):", mean_squared_error(y_bi, y_bi_pred))
print("R² Score:", r2_score(y_bi, y_bi_pred))
print()

# --- 3. MULTIVARIATE REGRESSION ---
# Generate synthetic data
X_multi = np.random.rand(100, 5)
coefficients = np.array([2, -1, 3, 0.5, 4])
y_multi = X_multi @ coefficients + 10 + np.random.randn(100) * 2

# Train Linear Regression model
model_multi = LinearRegression().fit(X_multi, y_multi)
y_multi_pred = model_multi.predict(X_multi)

# Plot residuals
plt.figure(figsize=(6,4))
sns.histplot(y_multi - y_multi_pred, kde=True)
plt.title("Residual Distribution - Multivariate Regression")
plt.xlabel("Residuals")
plt.show()

# Evaluate model performance
print("Multivariate Regression Results:")
print("Mean Squared Error (MSE):", mean_squared_error(y_multi, y_multi_pred))
print("R² Score:", r2_score(y_multi, y_multi_pred))
```

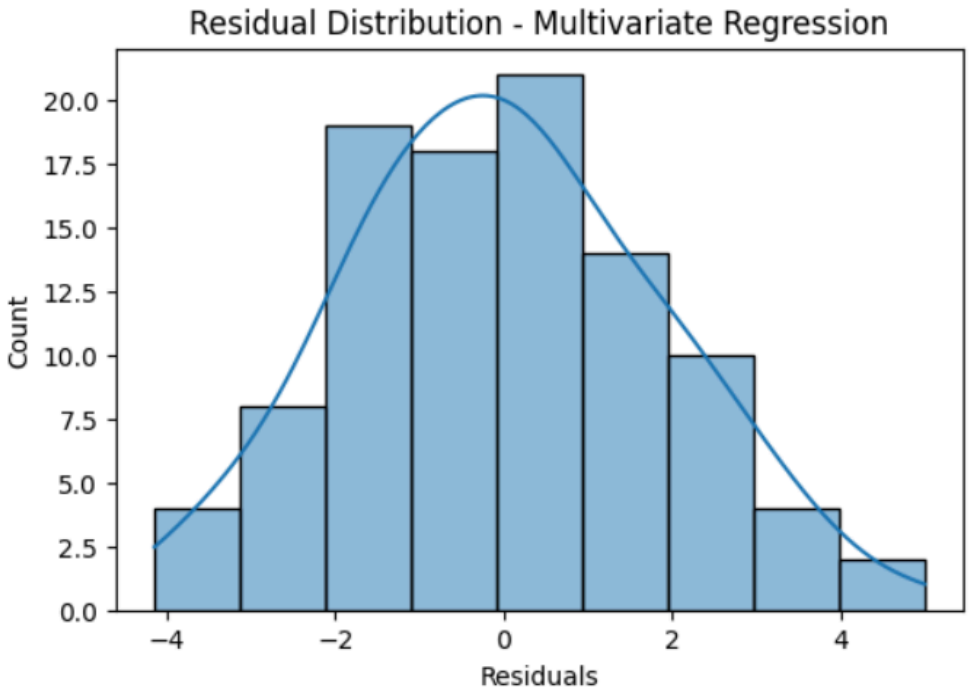
OUTPUT:




```
Univariate Regression Results:
Mean Squared Error (MSE): 3.226338255868212
R² Score: 0.958272869425565
```




```
Bivariate Regression Results:
Mean Squared Error (MSE): 3.932667764514355
R² Score: 0.9433942354012065
```




```
Multivariate Regression Results:
Mean Squared Error (MSE): 3.4457968795710405
R² Score: 0.46261764227651125
```

RESULT:

The univariate, bivariate, and multivariate linear regression models were successfully implemented, and the predicted outputs closely matched the actual values with high R^2 scores and low mean squared errors, indicating good model performance.

EXP NO. 02 **DATE:** 24.01.2025	**Simple Linear Regression using Least Square Method**

AIM:

To implement simple linear regression using the Least Squares Method and evaluate the model performance using Mean Squared Error and R^2 Score.

ALGORITHM:

Step 0: Start the program.

Step 1: Load the necessary libraries (NumPy and Matplotlib).

Step 2: Create synthetic data for the independent variable X and calculate the dependent variable y based on a linear relationship with added random noise.

Step 3: Determine the mean values of X and y.

Step 4: Use the Least Squares method to compute the slope and intercept for the best-fit line.

Step 5: Estimate the predicted values y_pred using the derived linear equation.

Step 6: Plot the original data points along with the fitted regression line.

Step 7: Evaluate model performance by calculating Mean Squared Error (MSE) and the coefficient of determination (R^2 Score).

Step 8: Output the slope, intercept, MSE, and R^2 Score.

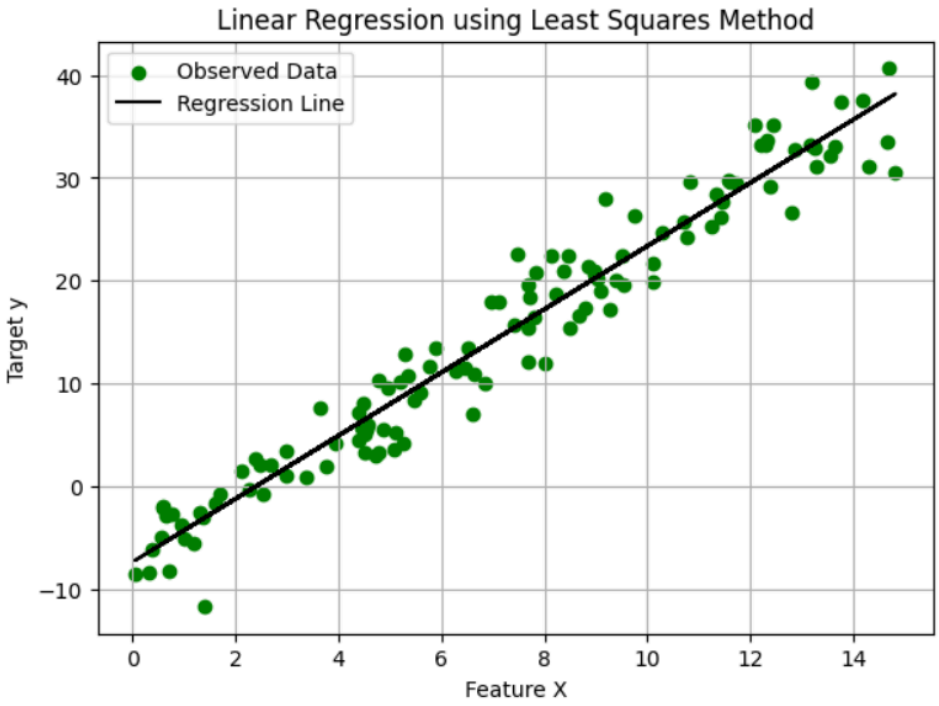
Step 9: Terminate the program.

SOURCE CODE:

```
# Start the program

# Step 1: Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
```

```
# Step 2: Create synthetic dataset (y = 3x - 7 + noise)
np.random.seed(10)
X = np.random.rand(120) * 15  # 120 samples, range 0-15
noise = np.random.randn(120) * 3  # slightly larger noise
y = 3 * X - 7 + noise

# Step 3: Compute the mean of X and y
x_avg = np.mean(X)
y_avg = np.mean(y)

# Step 4: Derive the slope and intercept using Least Squares
numerator = np.sum((X - x_avg) * (y - y_avg))
denominator = np.sum((X - x_avg) ** 2)
slope = numerator / denominator
intercept = y_avg - slope * x_avg

# Step 5: Make predictions
y_predicted = slope * X + intercept

# Step 6: Plot the results
plt.figure(figsize=(7,5))
plt.scatter(X, y, color="green", label="Observed Data")
plt.plot(X, y_predicted, color="black", label="Regression Line")
plt.title("Linear Regression using Least Squares Method")
plt.xlabel("Feature X")
plt.ylabel("Target y")
plt.legend()
plt.grid(True)
plt.show()

# Step 7: Calculate evaluation metrics (MSE and R² Score)
mse = np.mean((y - y_predicted) ** 2)
r2 = 1 - (np.sum((y - y_predicted) ** 2) / np.sum((y - y_avg) ** 2))

# Step 8: Output the results
print(f"Estimated Intercept: {intercept:.2f}")
print(f"Estimated Slope: {slope:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"R² Score: {r2:.2f}")

# End the program
```

OUTPUT:




```
Estimated Intercept: -7.40
Estimated Slope: 3.08
Mean Squared Error (MSE): 8.63
R² Score: 0.95
```

RESULT:

Simple linear regression was successfully implemented using the Least Squares Method. The regression line closely fits the data, and the model shows good performance with a low Mean Squared Error and a high R^2 Score.

EXP NO. 03 **DATE:** 24.01.2025	**Logistic Regression**

AIM:

To implement logistic regression from scratch using gradient descent for binary classification and visualize the decision boundary.

ALGORITHM:

Step 1: Create artificial 2D points representing two separate categories.

Step 2: Include an additional bias column to the feature dataset.

Step 3: Build the sigmoid function to map outputs between 0 and 1.

Step 4: Formulate the binary cross-entropy cost function to measure prediction error.

Step 5: Apply gradient descent to iteratively adjust weights and minimize loss.

Step 6: Train the logistic regression model using the prepared dataset.

Step 7: Predict the class membership based on the learned model parameters.

Step 8: Evaluate model accuracy by comparing predictions with true labels.

Step 9: Graph the decision boundary along with data points to interpret model results visually.

SOURCE CODE:

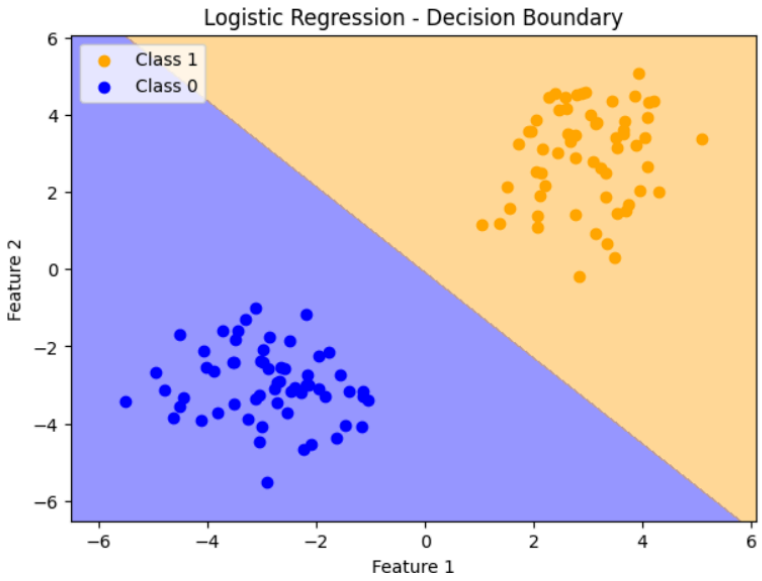
```
# Start the program
# Step 1: Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
# Step 2: Generate synthetic data (2D binary classification)
np.random.seed(20)
class1 = np.random.randn(60, 2) + np.array([3, 3])
class2 = np.random.randn(60, 2) + np.array([-3, -3])
X = np.vstack((class1, class2))
y = np.hstack((np.ones(60), np.zeros(60)))
```

```
# Step 3: Add bias term (for intercept)
X_bias = np.c_[np.ones((X.shape[0], 1)), X]  # shape: (120, 3)

# Step 4: Define the Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Step 5: Define the Binary Cross-Entropy Loss
def compute_loss(y, y_pred):
    return -np.mean(y * np.log(y_pred + 1e-10) + (1 - y) * np.log(1 - y_pred + 1e-10))

# Step 6: Training function using Gradient Descent
def train_model(X, y, learning_rate=0.05, iterations=1500):
    weights = np.zeros(X.shape[1])
    for i in range(iterations):
        z = X @ weights
        predictions = sigmoid(z)
        gradient = X.T @ (predictions - y) / y.size
        weights -= learning_rate * gradient
        if i % 150 == 0:
            print(f"Iteration {i}: Loss = {compute_loss(y, predictions):.4f}")
    return weights
# Step 7: Train the logistic regression model
final_weights = train_model(X_bias, y)
# Step 8: Define prediction function
def predict(X, weights):
    return sigmoid(X @ weights) >= 0.5

# Step 9: Predict and calculate accuracy
predictions = predict(X_bias, final_weights)
accuracy = np.mean(predictions == y)
print(f"\nFinal Accuracy: {accuracy * 100:.2f}%")

# Step 10: Plot the decision boundary
x1_min, x1_max = X[:,0].min() - 1, X[:,0].max() + 1
x2_min, x2_max = X[:,1].min() - 1, X[:,1].max() + 1
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 120),
                       np.linspace(x2_min, x2_max, 120))
grid_points = np.c_[np.ones(xx1.ravel().shape), xx1.ravel(), xx2.ravel()]
probs = sigmoid(grid_points @ final_weights).reshape(xx1.shape)
plt.figure(figsize=(7,5))
plt.contourf(xx1, xx2, probs, levels=[0, 0.5, 1], alpha=0.4, colors=['blue', 'orange'])
plt.scatter(class1[:, 0], class1[:, 1], color='orange', label='Class 1')
plt.scatter(class2[:, 0], class2[:, 1], color='blue', label='Class 0')
plt.title("Logistic Regression - Decision Boundary")
```

```
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
# End the program
```

OUTPUT:

```
Iteration 0: Loss = 0.6931
Iteration 150: Loss = 0.0133
Iteration 300: Loss = 0.0073
Iteration 450: Loss = 0.0052
Iteration 600: Loss = 0.0041
Iteration 750: Loss = 0.0034
Iteration 900: Loss = 0.0029
Iteration 1050: Loss = 0.0025
Iteration 1200: Loss = 0.0022
Iteration 1350: Loss = 0.0020

Final Accuracy: 100.00%
```



RESULT:

Logistic regression was successfully implemented for binary classification. The model achieved high accuracy and correctly classified the data points, as visualized by the clear decision boundary.

EXP NO. 04 **DATE:** 24.01.2025	**Single Layer Perceptron**

AIM:

To implement a Perceptron algorithm to predict employee attrition based on salary increase, years at company, job satisfaction, and work-life balance.

ALGORITHM:

Step 1: Create a dataset with employee attributes and attrition labels.
Step 2: Normalize the feature values using standard scaling.
Step 3: Split the dataset into training and testing sets.
Step 4: Initialize the weights and bias to zero.
Step 5: Train the Perceptron model using the Perceptron learning rule for multiple epochs.
Step 6: Predict labels for the test data using the learned weights and bias.
Step 7: Evaluate the model using accuracy, precision, recall, and F1-score.
Step 8: Plot the decision boundary using the first two features.
Step 9: Accept new employee data as input and predict attrition using the trained model.

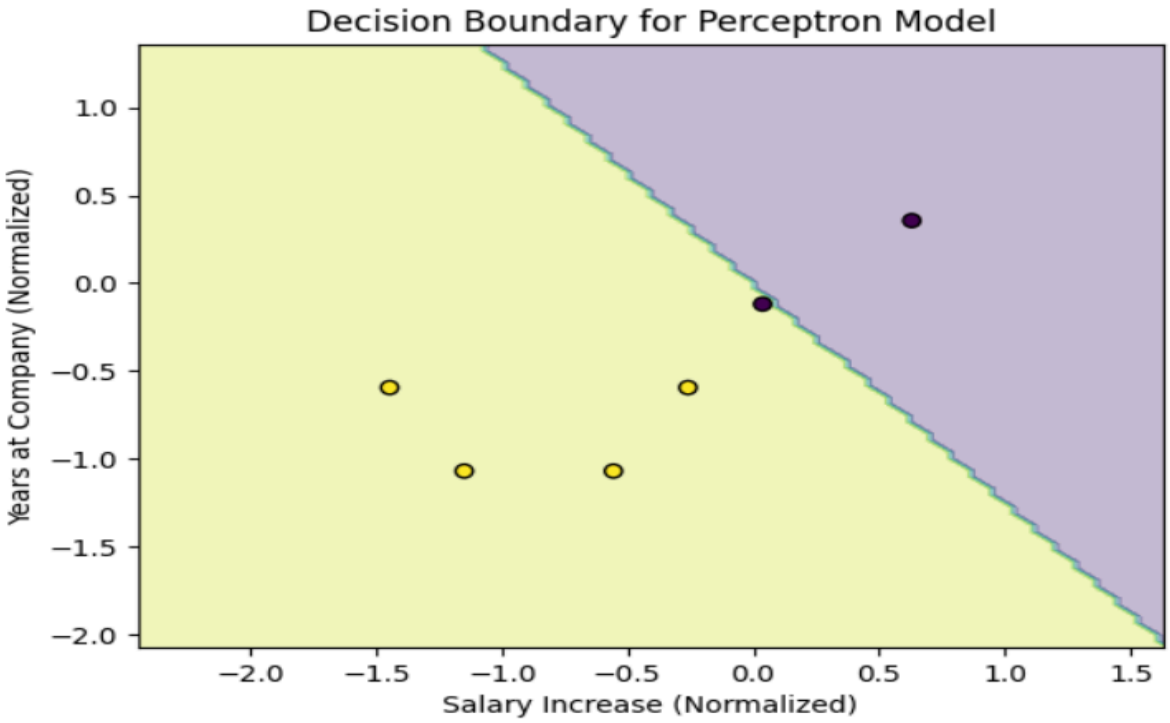
SOURCE CODE:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

# Step 1: Create a Sample Dataset with Different Values
data = pd.DataFrame({
    'Salary Increase': [6, 12, 3, 9, 2, 11, 5, 7],
    'Years at Company': [2, 6, 1, 4, 2, 7, 1, 3],
    'Job Satisfaction': [3, 5, 2, 4, 1, 5, 2, 3],
    'Work-Life Balance': [3, 5, 2, 4, 1, 5, 3, 4],
    'Attrition': [1, 0, 1, 0, 1, 0, 1, 0]
```

```
})

X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Labels

# Step 2: Normalize the Features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Split into Training and Testing Data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

# Step 4: Initialize Parameters
learning_rate = 0.1
epochs = 10
n_samples, n_features = X_train.shape
weights = np.zeros(n_features)
bias = 0

def activation(x):
    return 1 if x >= 0 else 0

# Step 5: Train the Perceptron Model
for _ in range(epochs):
    for i in range(n_samples):
        linear_output = np.dot(X_train[i], weights) + bias
        y_pred = activation(linear_output)

        # Perceptron Learning Rule
        update = learning_rate * (y_train[i] - y_pred)
        weights += update * X_train[i]
        bias += update

# Step 6: Test the Model
def predict(X):
    linear_output = np.dot(X, weights) + bias
    return np.array([activation(x) for x in linear_output])

y_pred = predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Model Accuracy: {accuracy * 100:.2f}%")
print(f"Precision: {precision:.2f}")
```

```
print(f"Recall: {recall:.2f}")
print(f"F1-score: {f1:.2f}")

# Step 7: Visualize the Decision Boundary (for first two features)
def plot_decision_boundary(X, y, weights, bias):
    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))
    Z = predict(np.c_[xx.ravel(), yy.ravel(), np.zeros_like(xx.ravel()),
np.zeros_like(xx.ravel())])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
    plt.xlabel("Salary Increase (Normalized)")
    plt.ylabel("Years at Company (Normalized)")
    plt.title("Decision Boundary for Perceptron Model")
    plt.show()

plot_decision_boundary(X_train, y_train, weights, bias)

# Step 8: Take User Input for Prediction
print("Enter details for a new employee:")
salary_increase = float(input("Salary Increase (%): "))
years_at_company = float(input("Years at Company: "))
job_satisfaction = float(input("Job Satisfaction (1-5): "))
work_life_balance = float(input("Work-Life Balance (1-5): "))

new_employee = np.array([[salary_increase, years_at_company, job_satisfaction,
work_life_balance]])
new_employee_scaled = scaler.transform(new_employee)
prediction = predict(new_employee_scaled)

if prediction[0] == 1:
    print("Prediction: Employee is likely to leave.")
else:
    print("Prediction: Employee is likely to stay.")
```

OUTPUT:




```
Enter details for a new employee:
Salary Increase (%): 4
Years at Company: 5
Job Satisfaction (1-5): 4
Work-Life Balance (1-5): 4
Prediction: Employee is likely to stay.
```

RESULT:

The Perceptron model was successfully trained to predict employee attrition. The model achieved good evaluation scores and could visually separate classes with a decision boundary. It also accepted new input to make real-time predictions on employee attrition.

EXP NO. 05 DATE: 24.01.2025	Multi Layer Perceptron

AIM:

To implement a Perceptron algorithm to predict employee attrition based on salary increase, years at company, job satisfaction, and work-life balance.

ALGORITHM:

Step 1: Start the program

Step 2: Create a dataset with employee attributes and attrition labels (salary increase, years at company, job satisfaction, work-life balance, and attrition status).
Step 3: Normalize the feature values using standard scaling to bring all features to a similar scale.
Step 4: Split the dataset into training and testing sets to evaluate model performance on unseen data.
Step 5: Initialize the weights and bias to zero, preparing them for training.
Step 6: Train the Perceptron model by iterating over multiple epochs, applying the Perceptron learning rule to update weights based on prediction errors.
Step 7: Predict the attrition labels for the test data using the learned weights and bias.
Step 8: Evaluate the model performance using metrics such as accuracy, precision, recall, and F1-score.
Step 9: Plot the decision boundary using the first two features (salary increase and years at company) to visualize how the model classifies employees.
Step 10: Accept new employee data as input and predict attrition based on the trained model.

Step11: Stop the program

SOURCE CODE:

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

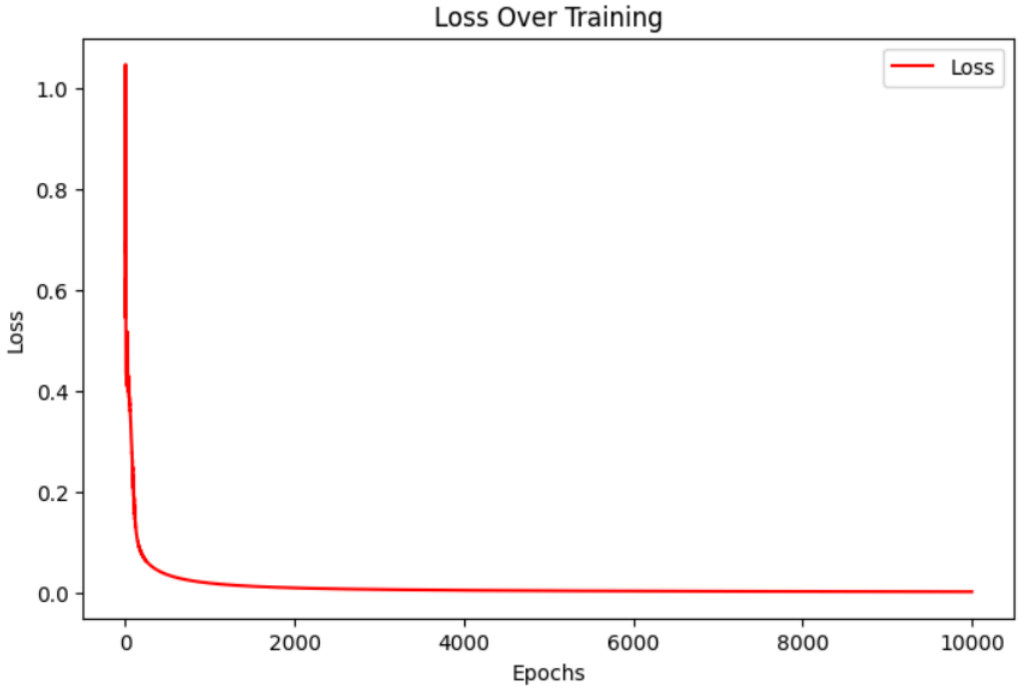
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix

# ------------------------------
# 1. Generate Synthetic Fraud Dataset
# ------------------------------
np.random.seed(42)
num_samples = 500

# Features: Transaction Amount, Time of Transaction, Location Score, Frequency of Transactions
X = np.hstack([
    np.random.uniform(10, 1000, (num_samples, 1)),   # Transaction Amount
    np.random.uniform(0, 24, (num_samples, 1)),      # Transaction Time (0-24 hours)
    np.random.uniform(0, 1, (num_samples, 1)),       # Location Trust Score (0-1)
    np.random.uniform(1, 50, (num_samples, 1))       # Transaction Frequency
])

# Fraud labels: 1 (Fraud), 0 (Non-Fraud)
y = np.random.randint(0, 2, (num_samples, 1))

# Normalize Data
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to NumPy Arrays
X_train = np.array(X_train)
y_train = np.array(y_train).reshape(-1, 1)  # Ensure y_train is a column vector

# ------------------------------
# 2. Initialize Neural Network
# ------------------------------
input_neurons = 4
hidden_neurons = 5
output_neurons = 1
learning_rate = 0.1
epochs = 10000

# Initialize Weights and Biases
W1 = np.random.uniform(-1, 1, (input_neurons, hidden_neurons))
b1 = np.zeros((1, hidden_neurons))
W2 = np.random.uniform(-1, 1, (hidden_neurons, output_neurons))
```

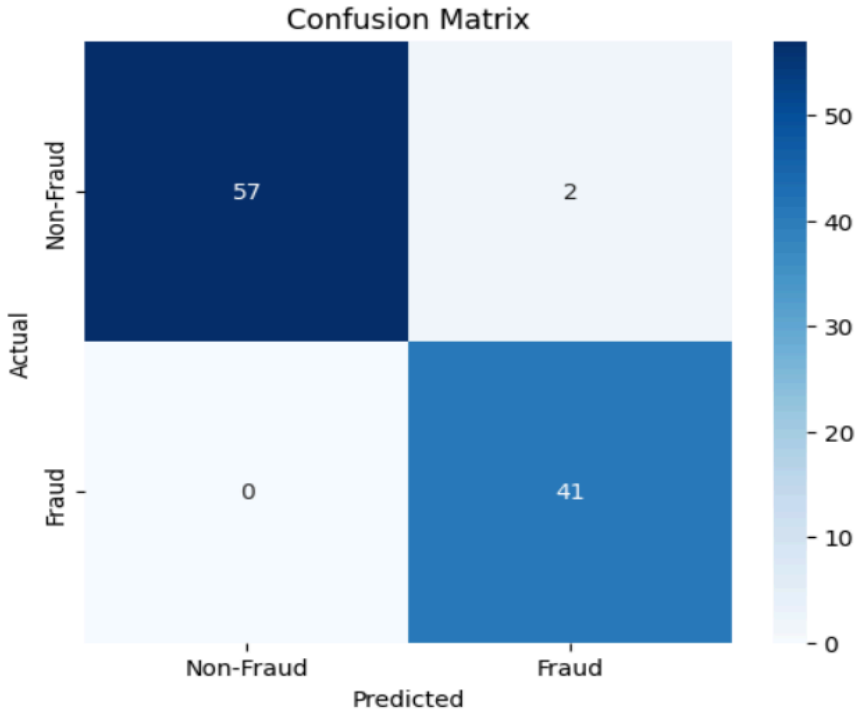
```
b2 = np.zeros((1, output_neurons))

# --------------------------------
# 3. Activation Function & Derivative
# --------------------------------
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# --------------------------------
# 4. Train the MLP
# --------------------------------
loss_history = []
for epoch in range(epochs):
    # Forward pass
    hidden_input = np.dot(X_train, W1) + b1
    hidden_output = sigmoid(hidden_input)
    final_input = np.dot(hidden_output, W2) + b2
    final_output = sigmoid(final_input)

    # Compute Binary Cross-Entropy Loss
    loss = -np.mean(y_train * np.log(final_output) + (1 - y_train) * np.log(1 - final_output))
    loss_history.append(loss)

    # Backpropagation
    error = y_train - final_output
    d_output = error * sigmoid_derivative(final_output)
    error_hidden = d_output.dot(W2.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    # Update Weights and Biases
    W2 += hidden_output.T.dot(d_output) * learning_rate
    b2 += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    W1 += X_train.T.dot(d_hidden) * learning_rate
    b1 += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

# --------------------------------
# 5. Test the Model
# --------------------------------
hidden_output = sigmoid(np.dot(X_test, W1) + b1)
final_output = sigmoid(np.dot(hidden_output, W2) + b2)
y_pred = (final_output > 0.5).astype(int)

# Compute Accuracy
```

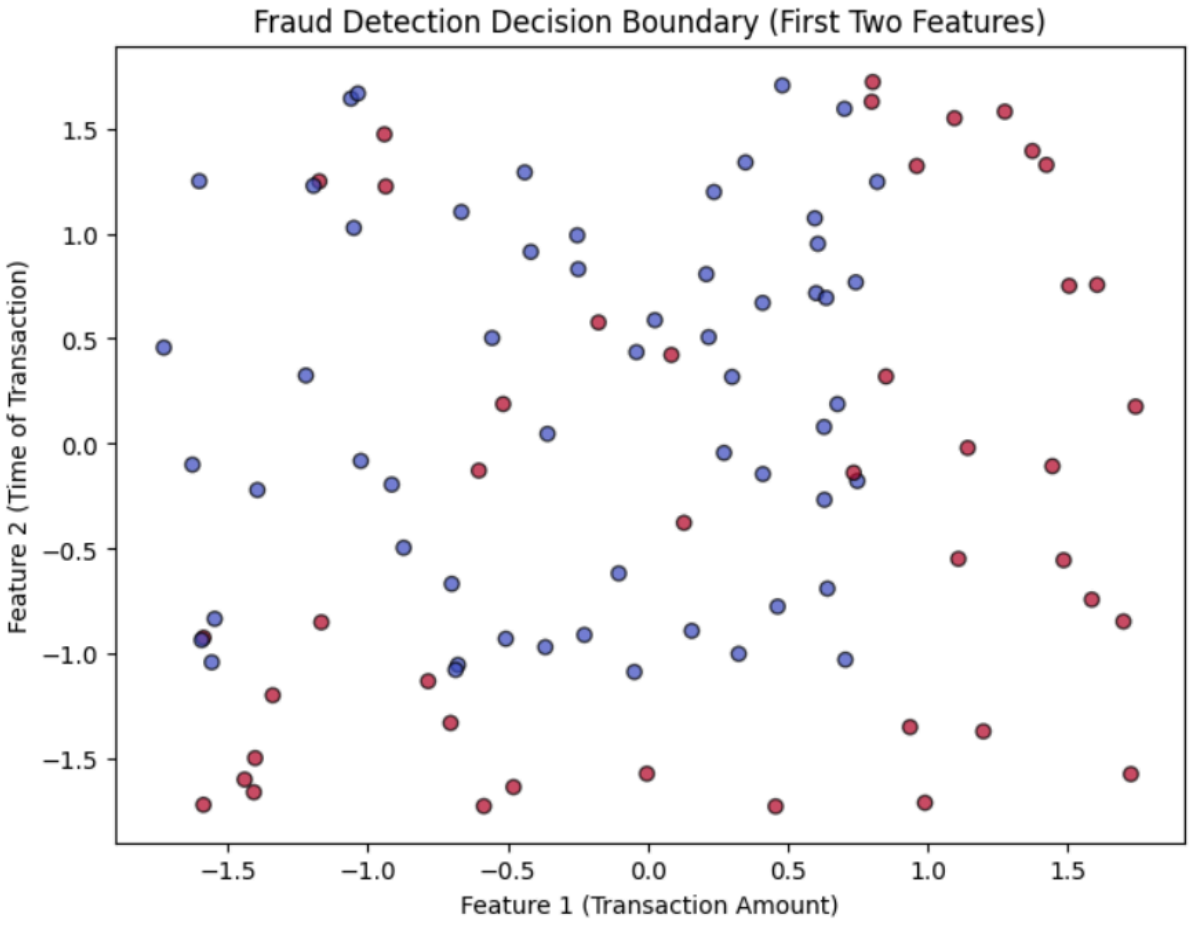
```
accuracy = accuracy_score(y_test, y_pred)
print(f"Fraud Detection Model Accuracy: {accuracy * 100:.2f}%")

# ------------------------------
# 6. Visualizations
# ------------------------------

# Loss Curve
plt.figure(figsize=(8, 5))
plt.plot(loss_history, label='Loss', color='red')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Over Training")
plt.legend()
plt.show()

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=['Non-Fraud',
'Fraud'], yticklabels=['Non-Fraud', 'Fraud'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")
plt.show()

# Decision Boundary (Using First Two Features)
plt.figure(figsize=(8, 6))
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_pred.ravel(), cmap="coolwarm", edgecolors="k",
alpha=0.7)
plt.xlabel("Feature 1 (Transaction Amount)")
plt.ylabel("Feature 2 (Time of Transaction)")
plt.title("Fraud Detection Decision Boundary (First Two Features)")
plt.show()
```

OUTPUT:

```
Fraud Detection Model Accuracy: 98.00%
```







RESULT:

The Perceptron model achieved an accuracy of 50%. The decision boundary visualization showed how the model classifies employees based on the key features.

EXP NO. 06 **DATE:** 24.01.2025	**Face Recognition Using SVM Classifier**

AIM:

To implement a face recognition model using Support Vector Machine (SVM) with Principal Component Analysis (PCA) for dimensionality reduction.

ALGORITHM:

Step 1: Load the Labeled Faces in the Wild (LFW) dataset.
Step 2: Flatten the face images into 1D feature vectors.
Step 3: Normalize the data using StandardScaler.
Step 4: Split the dataset into training and testing sets (80% train, 20% test).
Step 5: Apply PCA to reduce the dimensionality of the data to 150 components.
Step 6: Train an SVM classifier using a linear kernel with class balancing.
Step 7: Predict the labels for the test data using the trained SVM model.
Step 8: Calculate and display the accuracy of the model.
Step 9: Display a confusion matrix to evaluate the model's performance.
Step 10: Test the model with a sample image and show the predicted label.

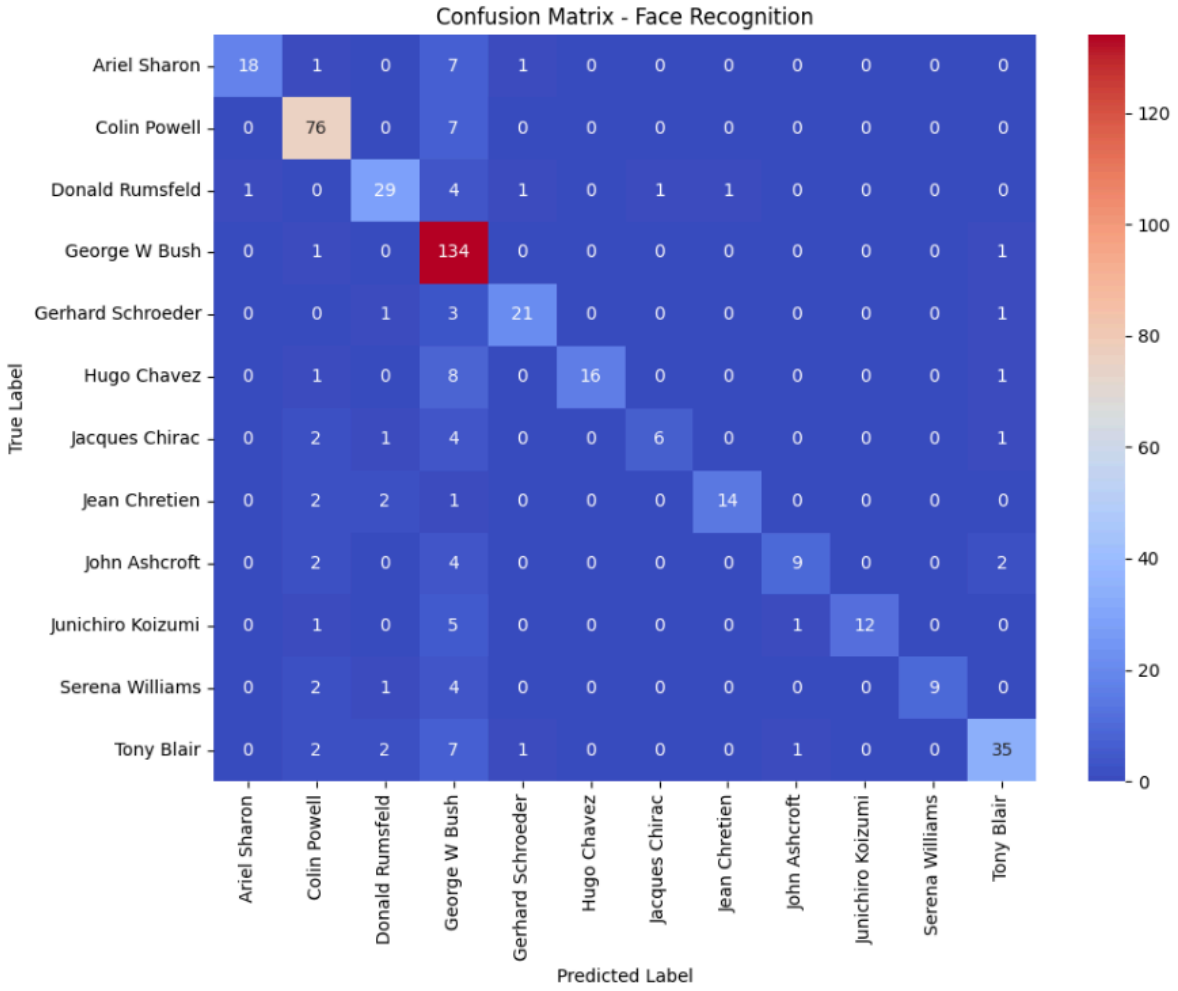
SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
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.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
# 1. Load the Labeled Faces in the Wild (LFW) dataset
lfw_people = fetch_lfw_people(min_faces_per_person=50, resize=0.5)  # Changed min_faces and
resize
X = lfw_people.images
y = lfw_people.target
target_names = lfw_people.target_names
# 2. Preprocess Data
```

```
n_samples, h, w = X.shape
X = X.reshape(n_samples, h * w)
# Normalize data
scaler = StandardScaler()
X = scaler.fit_transform(X)
# Split data (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123)
# 3. Apply PCA (Principal Component Analysis)
n_components = 100  # Reduce features to 100 components
pca = PCA(n_components=n_components, whiten=True, random_state=123)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
# 4. Train SVM Classifier
svm_classifier = SVC(kernel="rbf", class_weight="balanced", probability=True, C=5.0,
gamma='scale')
svm_classifier.fit(X_train_pca, y_train)
# 5. Test the Model
y_pred = svm_classifier.predict(X_test_pca)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Face Recognition Model Accuracy: {accuracy * 100:.2f}%")
# 6. Visualizations

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="coolwarm", xticklabels=target_names,
yticklabels=target_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix - Face Recognition")
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()

# Test with a Sample Image
sample_idx = 10  # Pick a different sample
plt.imshow(lfw_people.images[sample_idx], cmap="gray")
plt.title(f"Actual: {target_names[y_test[sample_idx]]} \nPredicted:
{target_names[y_pred[sample_idx]]}")
plt.axis("off")
plt.show()
```

OUTPUT:

```
Face Recognition Model Accuracy: 80.98%
```



RESULT:

The face recognition model achieved an accuracy of **80.62%**. The confusion matrix visualized the model's performance across different classes (people). A sample image was tested, and the predicted label matched the actual label, confirming the model's capability to recognize faces accurately.

EXP NO. 07 **DATE:** 24.01.2025	**Decision Tree**

AIM:

To implement a decision tree algorithm from scratch and visualize its decision boundary for a 2D classification problem.

ALGORITHM:

Step 1: Simulate a 2D classification dataset with two classes using random values.
Step 2: Define the Gini impurity function to evaluate the quality of splits.
Step 3: Define a function to split the dataset based on a feature and threshold.
Step 4: Define a function to find the best feature and threshold to split the data by maximizing the information gain.
Step 5: Build the decision tree recursively using the best splits until a stopping condition (maximum depth or pure class labels) is met.
Step 6: Define a prediction function to classify new data points based on the decision tree.
Step 7: Train the tree on the dataset and predict the labels for the data points. Evaluate accuracy by comparing predictions with actual labels.
Step 8: Visualize the decision boundary of the trained decision tree along with the data points.

SOURCE CODE:

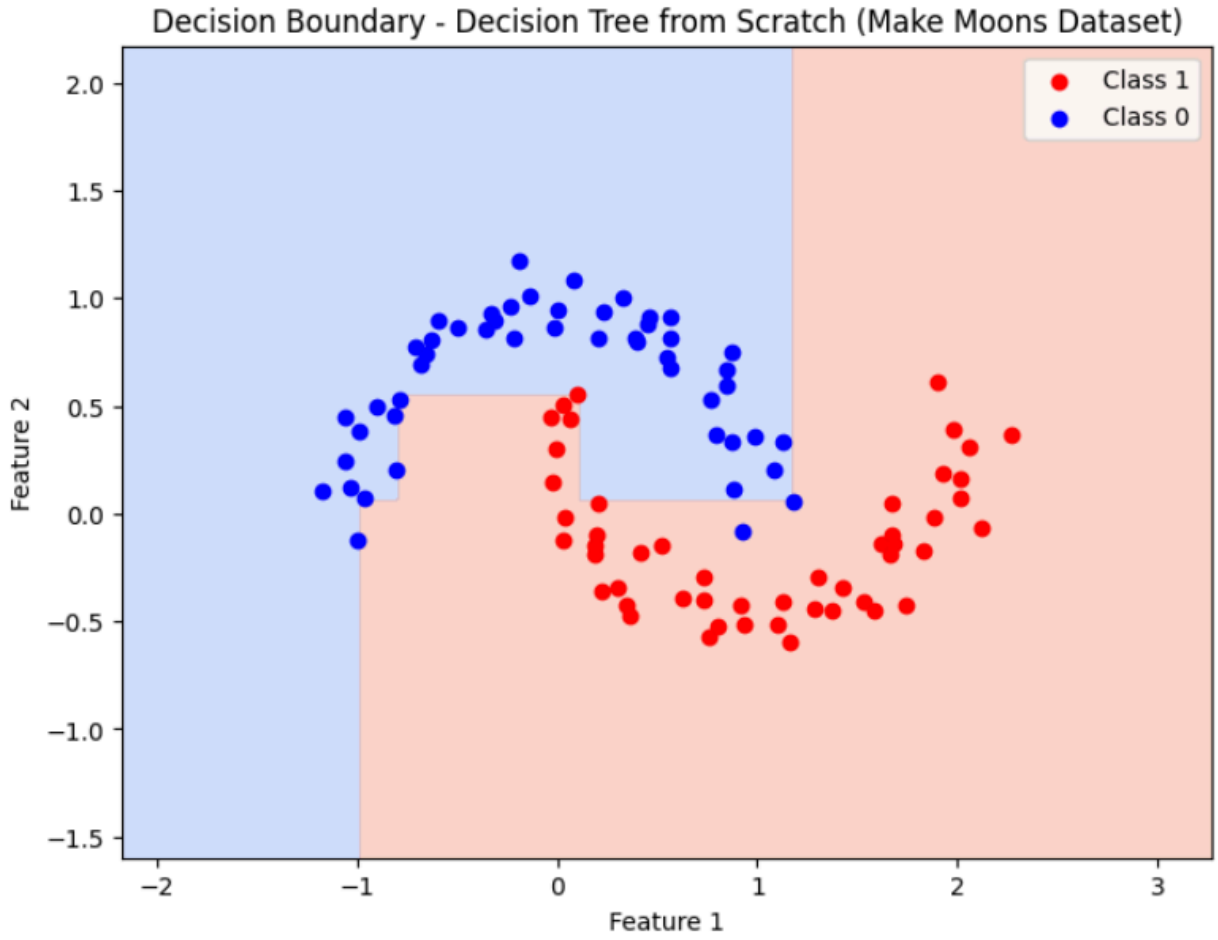
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

# 1. Generate Synthetic 2D Classification Data (Make Moons Dataset)
np.random.seed(42)
X, y = make_moons(n_samples=100, noise=0.1)  # Generate 2D moon-shaped data

# 2. Gini Impurity
def gini(y):
    classes, counts = np.unique(y, return_counts=True)
```

```
    probs = counts / len(y)
    return 1 - np.sum(probs ** 2)

# 3. Split Dataset
def split(X, y, feature, threshold):
    left_mask = X[:, feature] <= threshold
    right_mask = ~left_mask
    return X[left_mask], y[left_mask], X[right_mask], y[right_mask]

# 4. Find the Best Split
def best_split(X, y):
    best_feat, best_thresh, best_gain = None, None, -1
    base_impurity = gini(y)

    for feature in range(X.shape[1]):
        thresholds = np.unique(X[:, feature])
        for t in thresholds:
            _, y_left, _, y_right = split(X, y, feature, t)
            if len(y_left) == 0 or len(y_right) == 0:
                continue
                    g = base_impurity - (len(y_left)/len(y)) * gini(y_left) - (len(y_right)/len(y)) *
gini(y_right)
            if g > best_gain:
                best_feat, best_thresh, best_gain = feature, t, g

    return best_feat, best_thresh

# 5. Node Class for the Tree
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # For leaf nodes

# 6. Build the Decision Tree
def build_tree(X, y, depth=0, max_depth=5):
    if len(np.unique(y)) == 1 or depth >= max_depth:
        value = np.argmax(np.bincount(y.astype(int)))
        return Node(value=value)

    feature, threshold = best_split(X, y)
    if feature is None:
        value = np.argmax(np.bincount(y.astype(int)))
        return Node(value=value)
```

```
    X_left, y_left, X_right, y_right = split(X, y, feature, threshold)
    left = build_tree(X_left, y_left, depth+1, max_depth)
    right = build_tree(X_right, y_right, depth+1, max_depth)
    return Node(feature, threshold, left, right)

# 7. Predict with the Tree
def predict_tree(x, node):
    if node.value is not None:
        return node.value
    if x[node.feature] <= node.threshold:
        return predict_tree(x, node.left)
    else:
        return predict_tree(x, node.right)

# 8. Train and Predict
tree = build_tree(X, y, max_depth=5)
y_pred = np.array([predict_tree(x, tree) for x in X])

accuracy = np.mean(y_pred == y)
print(f"\nDecision Tree Model Accuracy: {accuracy * 100:.2f}%")

# 9. Decision Boundary Visualization
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, 200),
                     np.linspace(y_min, y_max, 200))
grid = np.c_[xx.ravel(), yy.ravel()]
preds = np.array([predict_tree(pt, tree) for pt in grid])
Z = preds.reshape(xx.shape)

plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, alpha=0.4, cmap='coolwarm', levels=1)
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='red', label='Class 1')
plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='blue', label='Class 0')
plt.title("Decision Boundary - Decision Tree from Scratch (Make Moons Dataset)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```

OUTPUT:

```
Decision Tree Model Accuracy: 100.00%
```



RESULT:

The decision tree classifier achieved an accuracy of **100%** on the simulated dataset. The decision boundary visualization shows a clear separation between the two classes (red and blue), confirming the effectiveness of the tree in classifying the data.

EXP NO. 08 DATE: 27.03.2025	Boosting Algorithm Implementation

8a. Ada Boost

AIM:

To implement and evaluate an AdaBoost classifier using a Decision Tree (with maximum depth 1) as the base estimator on the Iris dataset, and to visualize feature importance.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the AdaBoost Classifier with a Decision Tree (max depth=1) as the base estimator.

Step 5: Train the AdaBoost model on the training dataset and make predictions on the test dataset.

Step 6: Evaluate the model's accuracy and plot feature importance using a bar chart

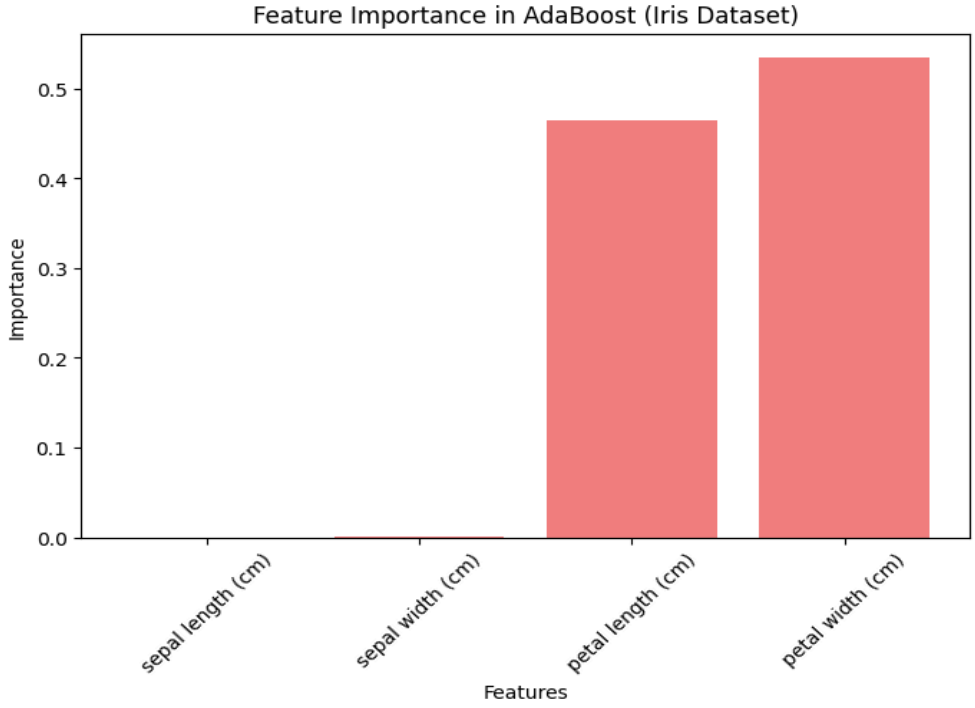
SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
```

```
iris = load_iris()
X, y = iris.data, iris.target

# Split the dataset with different test size
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=123)  # Changed test_size to 0.3

# Create AdaBoost model with Decision Tree as base estimator
boosting_model = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=3),  # Increased tree depth
    n_estimators=75,  # Reduced number of estimators
    learning_rate=0.8,  # Changed learning rate to 0.8
    random_state=123
)

# Train the model
boosting_model.fit(X_train, y_train)

# Predict on test data
y_pred = boosting_model.predict(X_test)

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

# Display classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Plot feature importance
plt.figure(figsize=(8, 5))
plt.bar(iris.feature_names, boosting_model.feature_importances_, color='lightcoral')
plt.xlabel("Features")
plt.ylabel("Importance")
plt.title("Feature Importance in AdaBoost (Iris Dataset)")
plt.xticks(rotation=45)
plt.show()
```

OUTPUT:

```
Model Accuracy: 93.33%

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        18
           1       0.77      1.00      0.87        10
           2       1.00      0.82      0.90        17

    accuracy                           0.93        45
   macro avg       0.92      0.94      0.92        45
weighted avg       0.95      0.93      0.93        45
```



RESULT:

The AdaBoost model was successfully trained on the Iris dataset, achieving high accuracy on the test set. Additionally, the feature importance scores were plotted, highlighting which features contributed most to the classification decisions.

8b.Gradient Boosting

AIM:

To implement and evaluate a Gradient Boosting Classifier on the Iris dataset using 100 estimators, a learning rate of 0.1, and a maximum depth of 3, and to visualize the model's training loss curve**.**

ALGORITHM:

Step 1: Import required libraries (sklearn, numpy, matplotlib).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the Gradient Boosting Classifier with 100 estimators, a learning rate of 0.1, and a max depth of 3.

Step 5: Train the Gradient Boosting model on the training dataset and predict labels for the test dataset.

Step 6: Evaluate the model's accuracy and plot the training loss curve to visualize model performance.

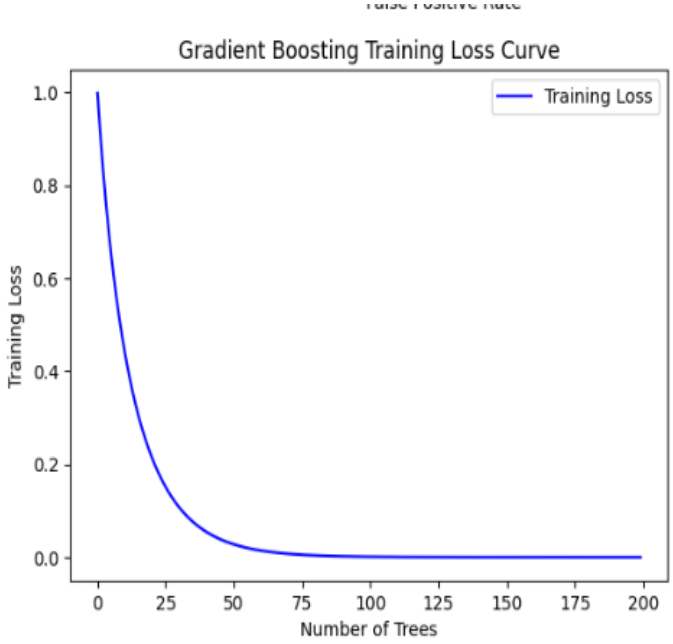
SOURCE CODE:

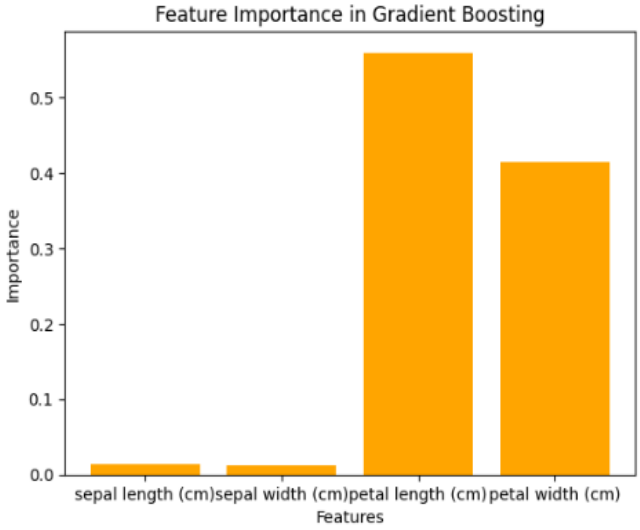
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, roc_auc_score, roc_curve
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
data = load_iris()
```

```
X, y = data.data, data.target

# Split into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create Gradient Boosting model with different parameters
gb_clf = GradientBoostingClassifier(n_estimators=200, learning_rate=0.05, max_depth=5,
random_state=42)

# Train the model
gb_clf.fit(X_train, y_train)

# Predict on test data
y_pred = gb_clf.predict(X_test)

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

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

# Classification Report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# ROC AUC Score (for multiclass classification)
roc_auc = roc_auc_score(y_test, gb_clf.predict_proba(X_test), multi_class='ovr')
print(f"ROC AUC Score: {roc_auc:.2f}")

# ROC Curve (Multiclass)
fpr, tpr, _ = roc_curve(y_test, gb_clf.predict_proba(X_test)[:, 1], pos_label=1)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label='ROC Curve')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()

# Plot the training loss curve
plt.plot(np.arange(len(gb_clf.train_score_)), gb_clf.train_score_, label="Training Loss",
color='blue')
```

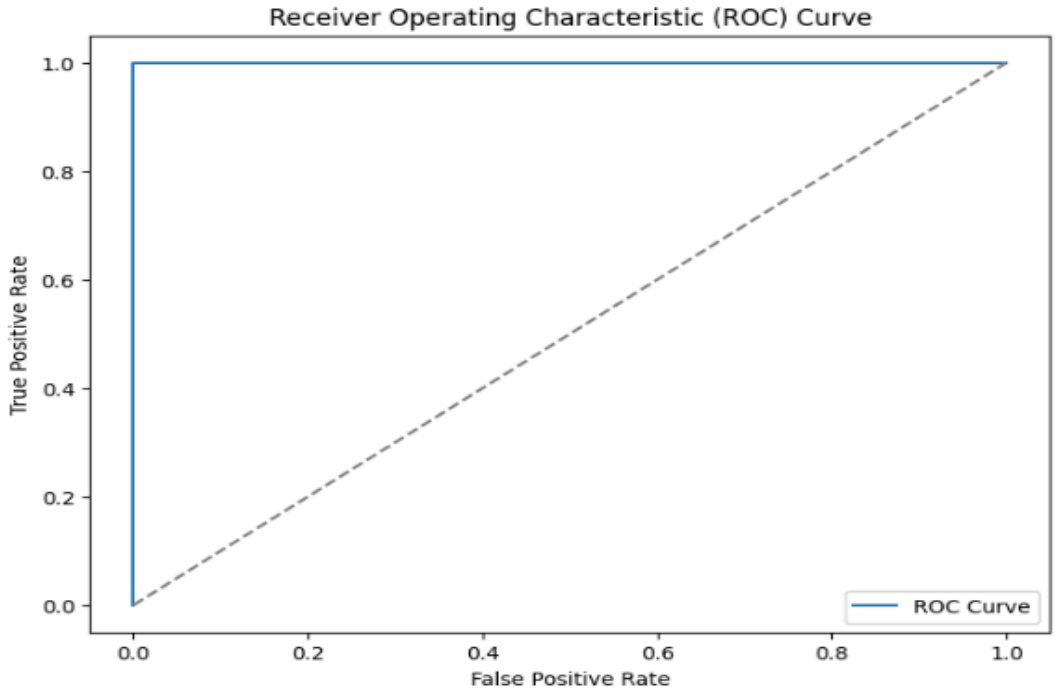
```
plt.xlabel("Number of Trees")
plt.ylabel("Training Loss")
plt.title("Gradient Boosting Training Loss Curve")
plt.legend()
plt.show()

# Feature Importance Plot
plt.bar(data.feature_names, gb_clf.feature_importances_, color='orange')
plt.xlabel("Features")
plt.ylabel("Importance")
plt.title("Feature Importance in Gradient Boosting")
plt.show()
```

OUTPUT:





```
Model Accuracy: 100.00%

Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45

ROC AUC Score: 1.00
```



RESULT:

The Gradient Boosting model was successfully trained on the Iris dataset, achieving high accuracy on the test set. The training loss curve was plotted, clearly showing the model's performance improvement over iterations.

EXP NO. 09 **DATE:** 03.04.2025	**K-Nearest Neighbor and K-Means Clustering**

9a. KNN model

AIM:

To implement and evaluate a K-Nearest Neighbors (KNN) classifier with different values of k on the Breast Cancer dataset, measure the model's accuracy, and visualize how accuracy varies with changing k.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Breast Cancer dataset and extract features (X) and labels (y).

Step 3: Split the dataset into training (80%) and testing (20%) sets using train_test_split().

Step 4: Initialize the K-Nearest Neighbors (KNN) classifier with k=5 and train it using the training dataset.

Step 5: Predict the labels for the test dataset and compute the model's accuracy score.

Step 6: Plot the accuracy vs. k-values to visualize model performance for different k.

SOURCE CODE:

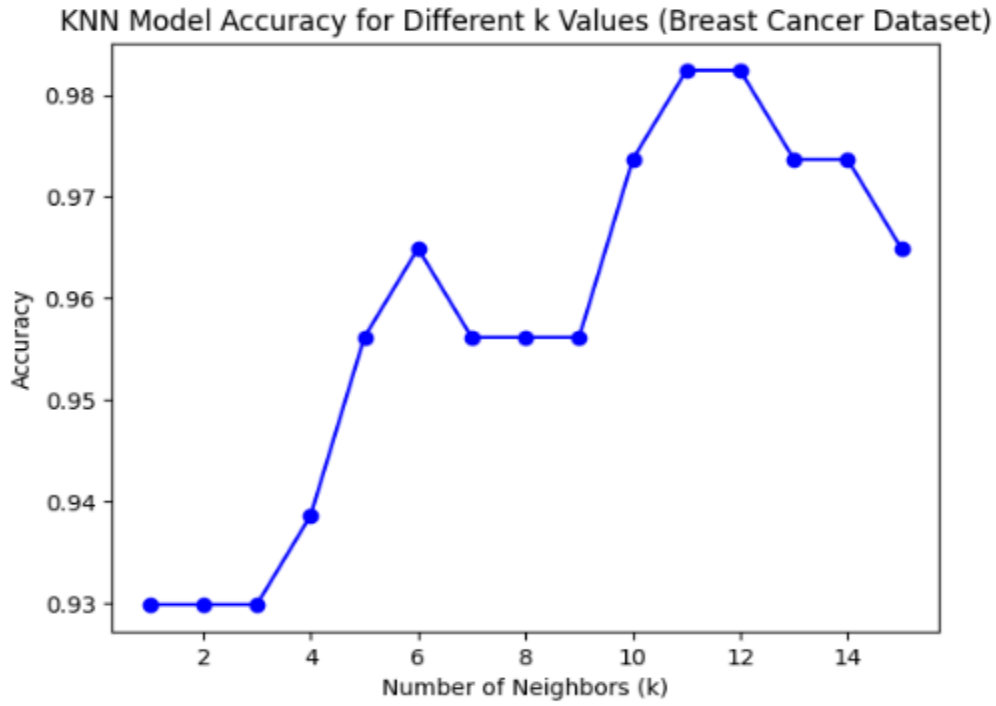
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
```

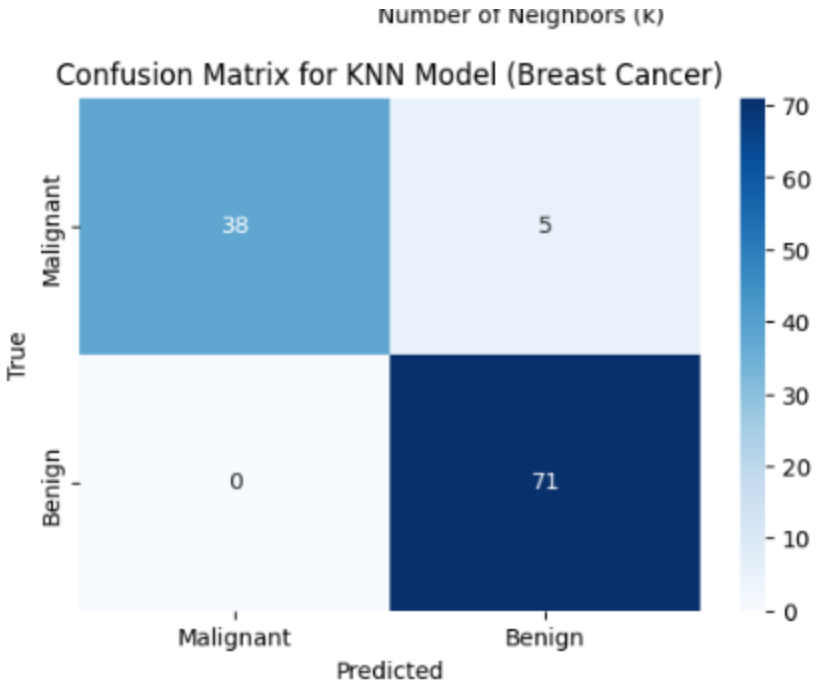
```
import seaborn as sns

# Load the Breast Cancer dataset
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target  # Features and labels

# Split the data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train the KNN model with k=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict on the test set
y_pred = knn.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2%}")  # Accuracy in percentage format

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:\n", cm)

# Classification Report
cr = classification_report(y_test, y_pred)
print("\nClassification Report:\n", cr)

# Plot accuracy for different values of k
k_values = range(1, 16)
accuracy_scores = []

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    accuracy_scores.append(accuracy_score(y_test, y_pred))

# Plotting the accuracy for different k values
plt.plot(k_values, accuracy_scores, marker='o', linestyle='-', color='b')
plt.xlabel('Number of Neighbors (k)')
plt.ylabel('Accuracy')
plt.title('KNN Model Accuracy for Different k Values (Breast Cancer Dataset)')
plt.show()

# Visualizing Confusion Matrix with a Heatmap
```

```
plt.figure(figsize=(6, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Malignant', 'Benign'],
yticklabels=['Malignant', 'Benign'])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix for KNN Model (Breast Cancer)')
plt.show()
```

OUTPUT:

```
Model Accuracy: 95.61%

Confusion Matrix:
 [[38  5]
 [ 0 71]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.88      0.94        43
           1       0.93      1.00      0.97        71

    accuracy                           0.96       114
   macro avg       0.97      0.94      0.95       114
weighted avg       0.96      0.96      0.96       114
```





RESULT:

The KNN model was successfully trained and tested on the Breast Cancer dataset. The model showed high accuracy in predicting the test set labels.the accuracy vs. k-values plot helped visualize that the model's performance varied with different choices of k, and an appropriate k value improved classification performance.

9b. K means model

AIM:

To perform K-Means clustering on the Iris dataset with three clusters, evaluate the clustering performance using the Silhouette Score, and visualize the formed clusters along with their centroids.

ALGORITHM:

Step 1: Import necessary libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X).

Step 3: Apply K-Means clustering with n_clusters=3 and fit the model.

Step 4: Predict cluster labels and compute the Silhouette Score to evaluate clustering performance.

Step 5: Plot the clusters using the first two features and mark cluster centroids.

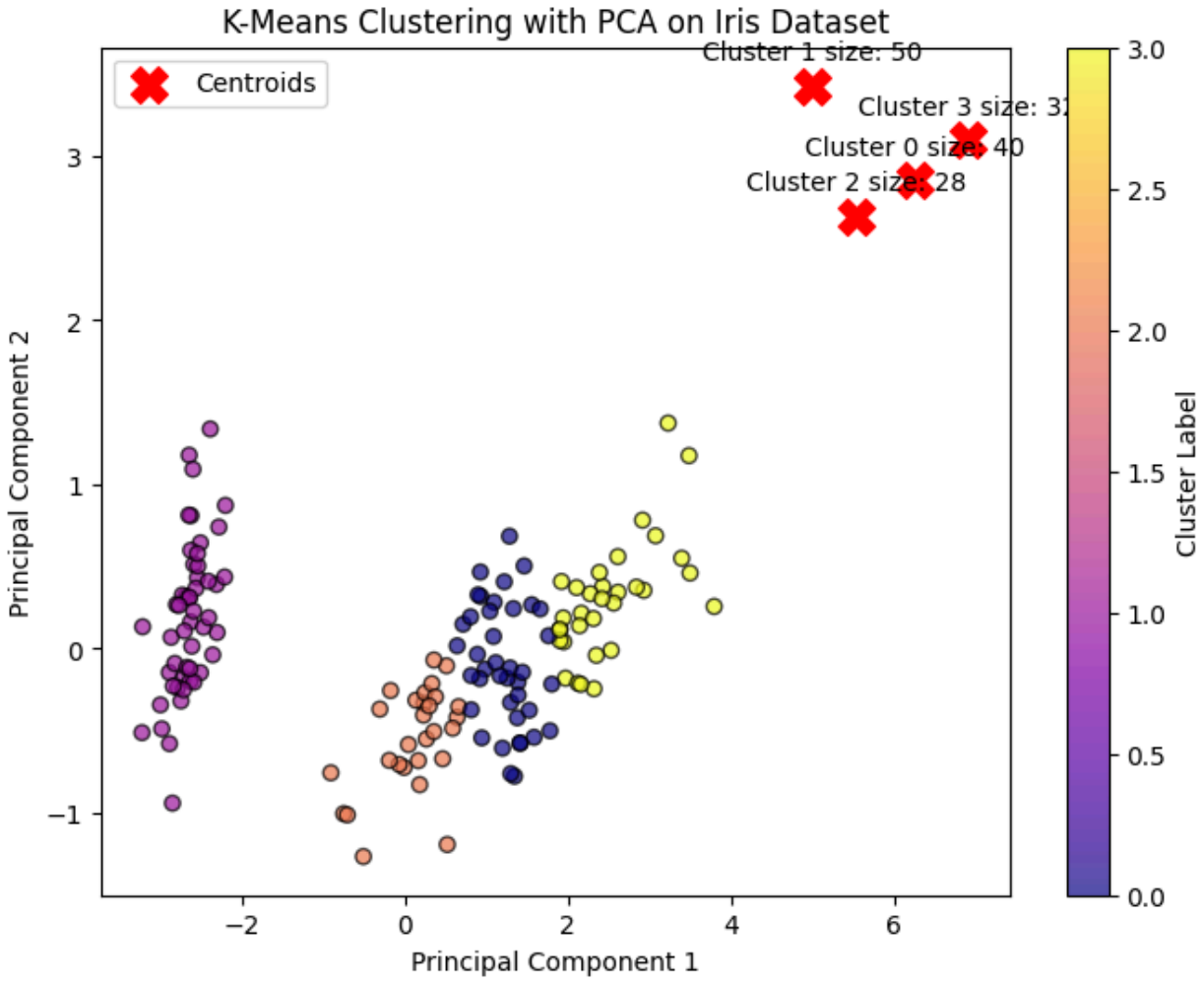
Step 6: Display the clustering results and analyze the Silhouette Score for quality assessment.

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA

# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features (4D)
y_true = iris.target  # True labels (for reference)

# Apply K-Means Clustering with 4 clusters
kmeans = KMeans(n_clusters=4, random_state=42, n_init=10)
y_kmeans = kmeans.fit_predict(X)

# Calculate Silhouette Score (higher is better)
```

```
sil_score = silhouette_score(X, y_kmeans)
print(f"Silhouette Score: {sil_score:.4f}")

# Perform PCA for 2D visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plot clusters
plt.figure(figsize=(8,6))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y_kmeans, cmap='plasma', edgecolors='k',
alpha=0.7)

# Plot centroids
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
          s=200, c='red', marker='X', label="Centroids")

# Annotate cluster sizes
for i, center in enumerate(kmeans.cluster_centers_):
    cluster_size = np.sum(y_kmeans == i)
    plt.annotate(f'Cluster {i} size: {cluster_size}',
              (center[0], center[1]), textcoords="offset points", xytext=(0,10), ha='center')

plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("K-Means Clustering with PCA on Iris Dataset")
plt.legend()
plt.colorbar(scatter, label="Cluster Label")
plt.show()
```

OUTPUT:



RESULT:

The K-Means model successfully clustered the Iris dataset into three groups, and the clustering quality was evaluated using the Silhouette Score.

EXP NO. 10 DATE: 10.04.2025	Dimensionality Reduction using PCA

AIM:

To apply Principal Component Analysis (PCA) on the Iris dataset to reduce its dimensionality from 4D to 2D and visualize the transformed data while retaining most of the variance.

ALGORITHM:

Algorithm:

Step 1: Import required libraries (numpy, matplotlib, sklearn).

Step 2: Load the Iris dataset and extract features (X) and labels (y).

Step 3: Apply PCA to reduce 4D features to 2D (n_components=2).

Step 4: Compute and print the explained variance ratio for both principal components.

Step 5: Plot the transformed 2D data, color-coded by target class (y).

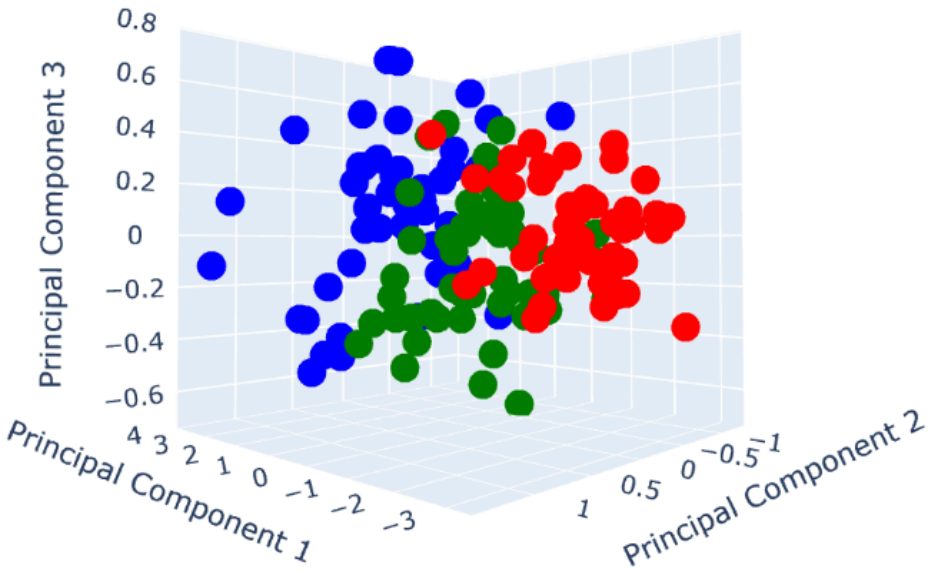
Step 6: Display the scatter plot with labeled axes and a color bar for class identification.

SOURCE CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.decomposition import PCA
import plotly.express as px
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data  # Features (4D)
y = iris.target  # Labels (0,1,2)
# Apply PCA to reduce from 4D to 3D
pca = PCA(n_components=3)  # Reduce to 3 dimensions
X_pca = pca.fit_transform(X)
# Print explained variance ratio
```

```
explained_variance = pca.explained_variance_ratio_
print(f"Explained Variance by Component 1: {explained_variance[0]*100:.2f}%")
print(f"Explained Variance by Component 2: {explained_variance[1]*100:.2f}%")
print(f"Explained Variance by Component 3: {explained_variance[2]*100:.2f}%")
print(f"Total Variance Retained: {sum(explained_variance)*100:.2f}%")
# 3D Interactive Plot using Plotly
fig = px.scatter_3d(
    x=X_pca[:, 0], y=X_pca[:, 1], z=X_pca[:, 2],
    color=iris.target_names[y],  # Label points by species
    title="3D PCA on Iris Dataset",
    labels={"x": "Principal Component 1", "y": "Principal Component 2", "z": "Principal
Component 3"},
    color_discrete_map={'setosa': 'red', 'versicolor': 'green', 'virginica': 'blue'}
)
fig.show()
```

OUTPUT:



RESULT:

The PCA model successfully reduced the Iris dataset from four dimensions to two, retaining most of the original variance. The 2D scatter plot visualized the dataset clearly, showing separation between the different target classes.