MACHINE LEARNING LAB MANUAL
(COCSC17)

SUBMITTED BY:

ANIL KUMAR

2021UCS1698

SUBMITTED TO:

Prof. Vijay Kumar Bohat

INDEX:

EXPERIMENT-1:

Aim: To implement Linear Regression.

Linear regression is a simple and widely used supervised learning algorithm for predicting a continuous outcome variable based on one or more predictor variables.

In [4]:
```
# Linear regression implementation
X_b = np.c_[np.ones((100, 1)), X]  # Add bias term to X
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

# Display the linear regression parameters
print("Theta Best (intercept, slope):", theta_best.ravel())
```

```
Theta Best (intercept, slope): [4.21509616 2.77011339]
```

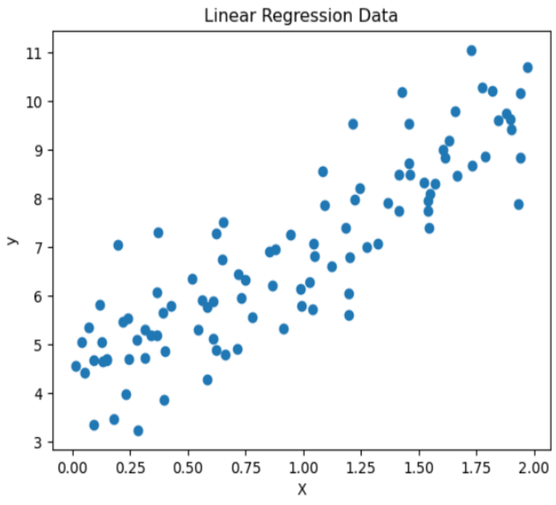
In [5]:
```
# Make predictions using the linear regression model
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new]  # Add bias term to new data
y_predict = X_new_b.dot(theta_best)
```

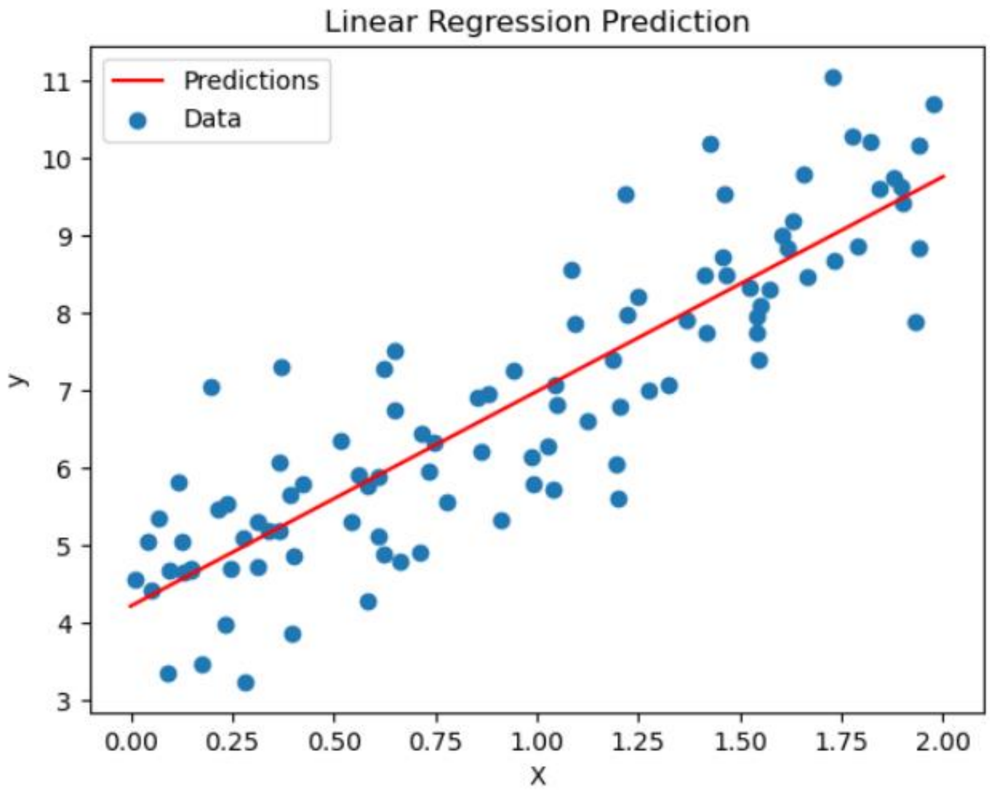
In [6]:
```
# Visualize the linear regression line
plt.plot(X_new, y_predict, 'r-', label='Predictions')
plt.scatter(X, y, label='Data')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Prediction')
plt.legend()
plt.show()
```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

In [3]:
```
plt.scatter(X, y)
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression Data')
plt.show()
```





EXPERIMENT-2:

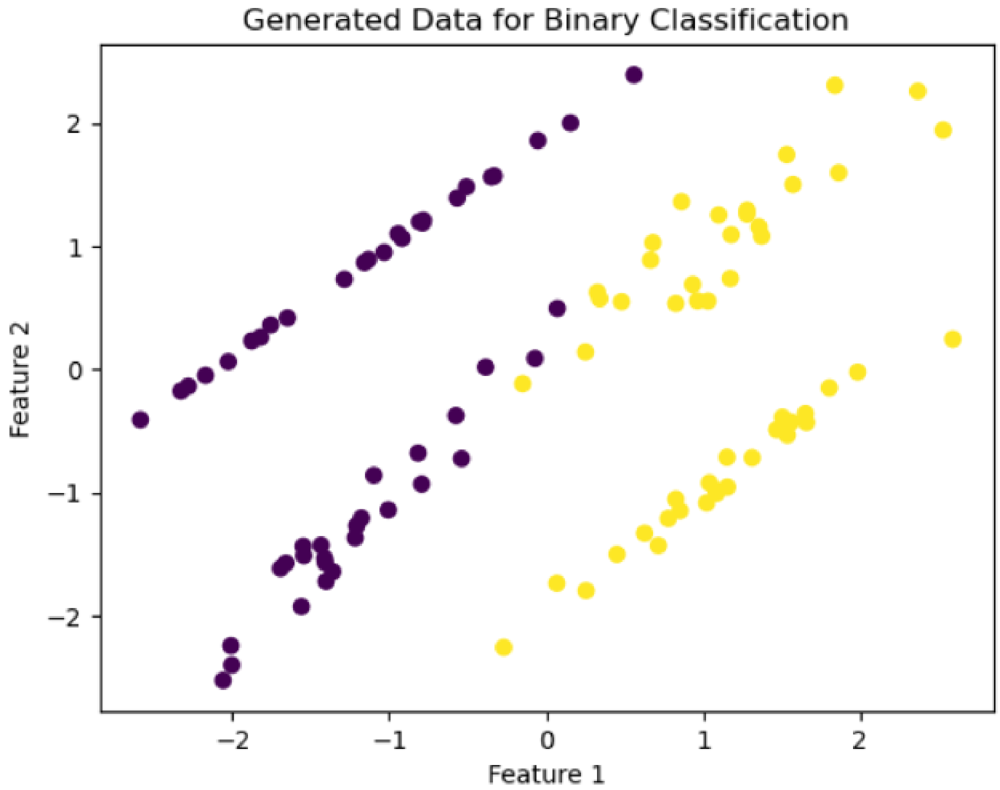
Aim: To implement Logistic Regression.

Logistic Regression is a binary classification algorithm used to predict the probability of an instance belonging to a particular class. Despite its name, logistic regression is used for classification rather than regression.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
```

In [2]:
```
# Generate synthetic data for demonstration
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, random_state=42)
```

In [3]:
```
# Visualize the data
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.title('Generated Data for Binary Classification')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



In [4]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply Logistic Regression
logreg = LogisticRegression(random_state=42)
logreg.fit(X_train, y_train)
```

Out[4]:
```
LogisticRegression(random_state=42)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [5]:
```
# Make predictions
y_pred = logreg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
```

In [6]:
```
# Display results
print("Accuracy:", accuracy)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.95
Confusion Matrix:
[[10  1]
 [ 0  9]]
```

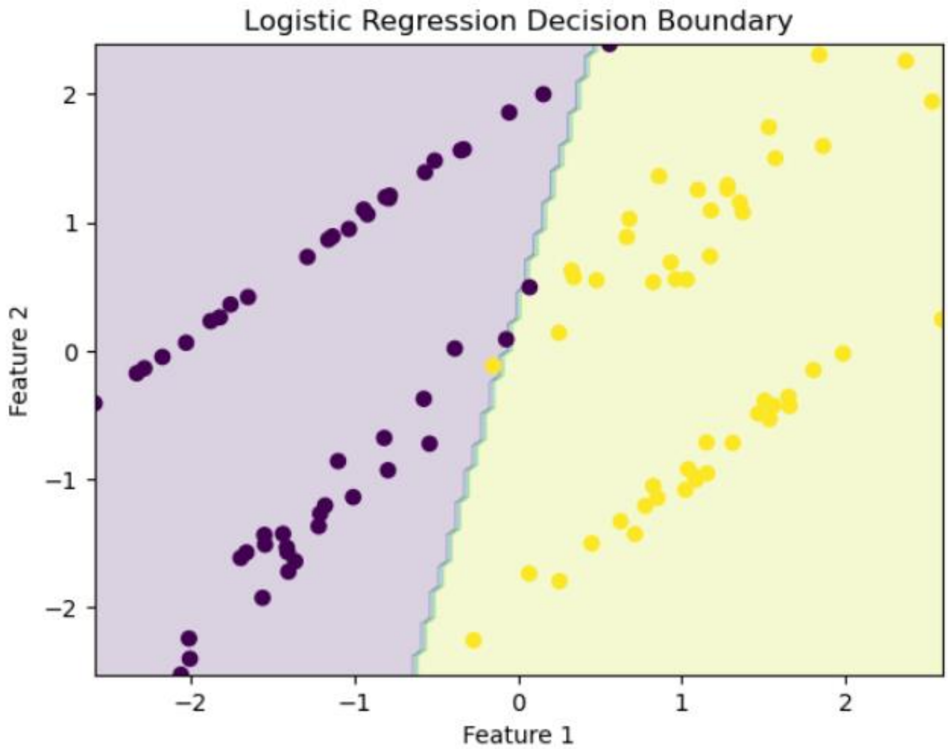
In [7]:

```
xx, yy = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
                     np.linspace(X[:, 1].min(), X[:, 1].max(), 100))

Z = logreg.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

In [8]:

```
plt.contourf(xx, yy, Z, alpha=0.2, cmap='viridis')
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis')
plt.title('Logistic Regression Decision Boundary')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



EXPERIMENT-3:

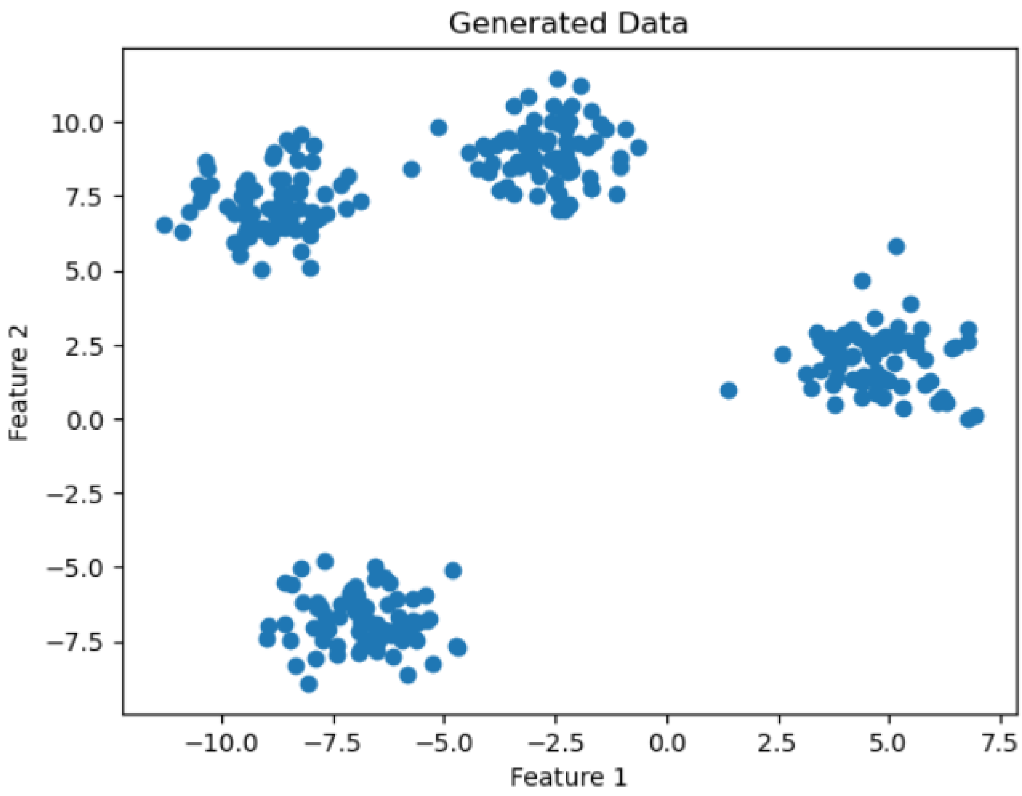
Aim: To implement K-Mean Clustering.

K-means clustering is an unsupervised machine learning algorithm used for partitioning a dataset into K distinct, non-overlapping subgroups or clusters. The algorithm works by iteratively assigning data points to clusters based on the similarity of their features and updating the cluster centroids until convergence.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
```

In [3]:
```
# Generate synthetic data for demonstration
data, _ = make_blobs(n_samples=300, centers=4, random_state=42)
```

In [4]:
```
# Visualize the data
plt.scatter(data[:, 0], data[:, 1])
plt.title('Generated Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



In [5]:

```
# Apply K-means clustering
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(data)
```

```
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the val
ue of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
C:\Users\HP\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when there
are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=2.
  warnings.warn(
```

Out[5]: KMeans(n_clusters=4, random_state=42)

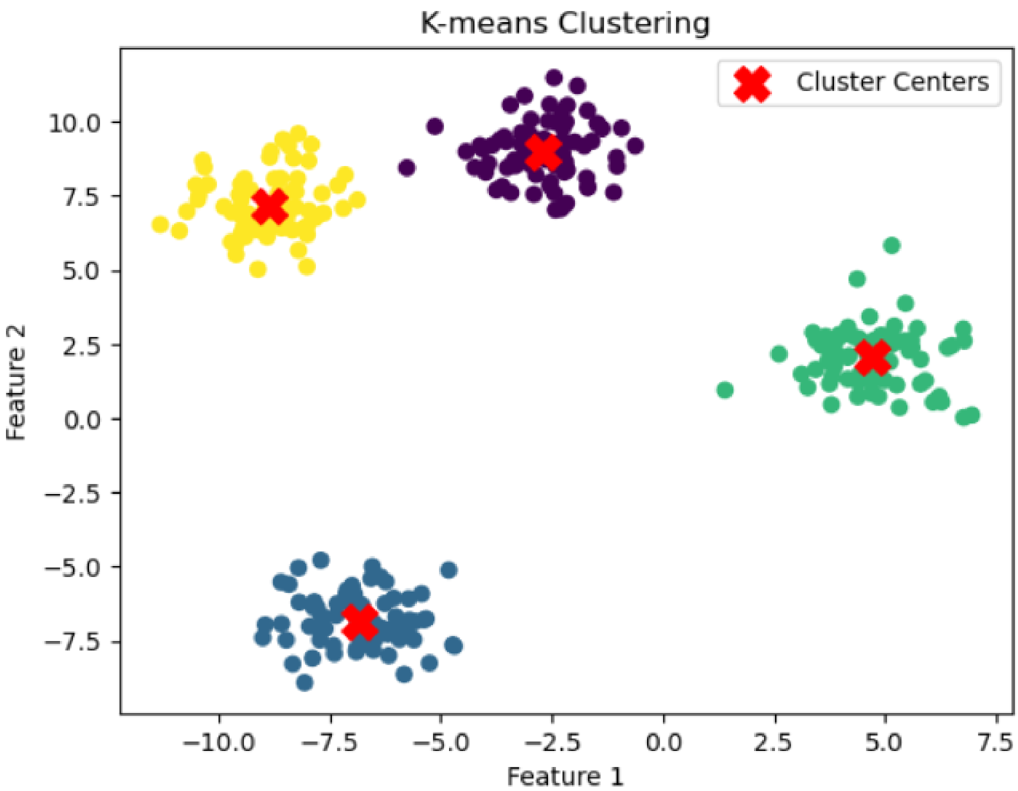
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [6]:

```
# Get cluster centers and labels
cluster_centers = kmeans.cluster_centers_
labels = kmeans.labels_
```

In [7]:

```
# Visualize the clustered data
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='viridis')
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], marker='X', s=200, color='red', label='Cluster Centers')
plt.title('K-means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```



EXPERIMENT-4:

Aim: To implement KNN.

KNN is a simple and intuitive supervised learning algorithm used for classification and regression tasks. It predicts the class or value of a new data point by considering the majority class or average of its k-nearest neighbors in the feature space. KNN is parameterized by k (number of neighbors) and relies on a distance metric (commonly Euclidean distance). It's computationally straightforward but can be sensitive to irrelevant features and requires storing the entire training dataset. KNN is suitable for smaller datasets and applications such as classification, regression.

In [1]:
```
import numpy as np
from collections import Counter
```

In [2]:
```
class KNN:
    def __init__(self, k=3):
        self.k = k

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

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

    def _predict(self, x):
        # Calculate distances between x and all examples in the training set
        distances = [np.linalg.norm(x - x_train) for x_train in self.X_train]

        # Get indices of k-nearest training data points
        k_neighbors_indices = np.argsort(distances)[:self.k]

        # Get the labels of the k-nearest training data points
        k_neighbor_labels = [self.y_train[i] for i in k_neighbors_indices]

        # Return the most common class label among the k neighbors
        most_common = Counter(k_neighbor_labels).most_common(1)
        return most_common[0][0]
```

In [3]:
```
# Example usage:
# Generate some random data for demonstration
np.random.seed(42)
X_train = np.random.rand(10, 2)
y_train = (X_train[:, 0] + X_train[:, 1] > 1).astype(int)

X_test = np.random.rand(5, 2)
```

on, and recommender systems.

In [4]:
```python
# Create and train the KNN classifier
knn = KNN(k=3)
knn.fit(X_train, y_train)

# Make predictions
predictions = knn.predict(X_test)

# Display the results
print("X_test:")
print(X_test)
print("Predictions:")
print(predictions)
```

```
X_test:
[[0.61185289 0.13949386]
 [0.29214465 0.36636184]
 [0.45606998 0.78517596]
 [0.19967378 0.51423444]
 [0.59241457 0.04645041]]
Predictions:
[0 0 1 0 0]
```

EXPERIMENT-5:

Aim: To implement PCA.

PCA is a dimensionality reduction technique used in machine learning. It transforms high-dimensional data into a lower-dimensional space while preserving the most important information. It identifies principal components, which are orthogonal directions capturing the maximum variance in the data. PCA is valuable for visualization, noise reduction, and speeding up machine learning algorithms by reducing feature dimensions.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
```

In [2]:
```python
# Load the Iris dataset for demonstration
iris = load_iris()
X = iris.data
y = iris.target
```

In [3]:
```python
# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Visualize the original and PCA-transformed data
plt.figure(figsize=(12, 5))
```

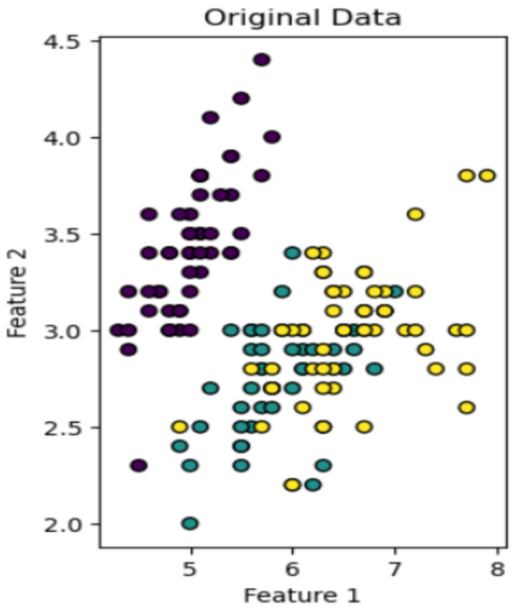
Out[3]:
```
<Figure size 1200x500 with 0 Axes>

<Figure size 1200x500 with 0 Axes>
```

In [4]:

```
# Plot original data
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolor='k')
plt.title('Original Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
```

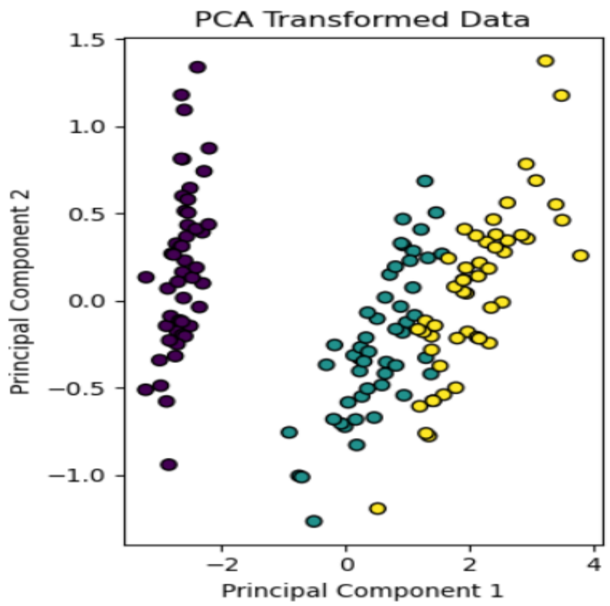
Out[4]: Text(0, 0.5, 'Feature 2')



In [5]:

```
# Plot PCA-transformed data
plt.subplot(1, 2, 2)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', edgecolor='k')
plt.title('PCA Transformed Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')

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



EXPERIMENT-6:

Aim: To implement Q-Learning Algorithm.

Q-Learning is a reinforcement learning algorithm used for making decisions in an environment. It learns a policy to maximize the cumulative reward over time. The algorithm iteratively updates a Q-table, representing the quality of actions in each state, based on observed rewards and transitions. Q-Learning is model-free, allowing it to adapt to unknown environments, and is widely used in solving problems like game playing and robotic control.

In [15]:
```python
import numpy as np

# Define the environment
num_states = 6
num_actions = 2
gamma = 0.8  # Discount factor
alpha = 0.1  # Learning rate
epsilon = 0.1  # Exploration-exploitation trade-off
```

In [16]:
```python
# Initialize Q-table
q_table = np.zeros((num_states, num_actions))

# Define the reward matrix
rewards = np.array([
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, 10]  # Goal state
])
```

In [17]:
```python
# Define the transition matrix
transitions = np.array([
    [1, 0],  # 0
    [2, 1],  # 1
    [3, 2],  # 2
    [4, 3],  # 3
    [5, 4],  # 4
    [5, 5]  # 5 (goal state)
])
```

In [18]:
```python
num_episodes = 1000

for episode in range(num_episodes):
    state = 0  # Initial state

    while state != 5:  # Continue until the goal state is reached
        # Exploration-exploitation trade-off
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(q_table[state, :])

        # Get the next state and reward
        next_state, reward = transitions[state, action], rewards[state, action]

        # Update Q-value using the Q-learning update rule
        q_table[state, action] = q_table[state, action] + alpha * (reward + gamma * np.max(q_table[next_state, :]) - q_table[state, action])

        # Move to the next state
        state = next_state
```

In [19]:
```python
# Display the learned Q-table
q_table
```

Out[19]:
```
array([[-3.3616    , -3.68561869],
       [-2.952     , -3.35686254],
       [-2.44      , -2.9498221 ],
       [-1.8       , -2.43660959],
       [-1.        , -1.7948587 ],
       [ 0.        ,  0.        ]])
```

EXPERIMENT-7:

Aim: To implement SARSA.

SARSA is a reinforcement learning algorithm for making decisions in an environment. Like Q-Learning, it learns a policy to maximize cumulative rewards. However, SARSA updates its Q-values using the current state, action, reward, and the next state and action taken. This on-policy approach allows SARSA to learn directly from its exploration policy, making it suitable for real-time applications where actions are continuously taken and updated.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```python
# Define the environment
num_states = 6
num_actions = 2
q_table = np.zeros((num_states, num_actions))

# Define the reward matrix
rewards = np.array([
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, -1],
    [-1, 10]  # Goal state
])

# Define the transition matrix
transitions = np.array([
    [1, 0],  # 0
    [2, 1],  # 1
    [3, 2],  # 2
    [4, 3],  # 3
    [5, 4],  # 4
    [5, 5]   # 5 (goal state)
])
```

In [3]:
```python
# SARSA algorithm
epsilon = 0.1  # Exploration-exploitation trade-off
alpha = 0.1   # Learning rate
gamma = 0.9    # Discount factor

def select_action(state):
    if np.random.rand() < epsilon:
        return np.random.choice(num_actions)
    else:
        return np.argmax(q_table[state, :])

num_episodes = 1000
```

In [4]:
```python
for episode in range(num_episodes):
    state = 0  # Initial state
    action = select_action(state)

    while state != 5:  # Continue until the goal state is reached
        next_state, reward = transitions[state, action], rewards[state, action]
        next_action = select_action(next_state)

        # SARSA update rule
        q_table[state, action] = q_table[state, action] + alpha * (reward + gamma * q_table[next_state, next_action] - q_table[state, action])

        state = next_state
        action = next_action
```

In [5]:
```python
# Display the learned Q-table
q_table
```

Out[5]:
```
array([[-4.22253515, -4.78426572],
       [-3.59266466, -4.20137798],
       [-2.78914497, -3.50760347],
       [-1.92245805, -2.76866793],
       [-1.        , -1.92944949],
       [ 0.        ,  0.        ]])
```

EXPERIMENT-8:

Aim: To implement Perceptron.

The perceptron is the simplest form of a neural network. It's a binary linear classifier that takes multiple binary inputs, applies weights, and produces a binary output. During training, it adjusts its weights based on misclassifications to learn a decision boundary. Perceptrons are the building blocks of neural networks, but they have limitations in handling non-linear problems.

In [1]:

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

In [24]:

```
class Perceptron:
    def __init__(self, input_size, learning_rate=0.01, epochs=100):
        self.weights = np.zeros(input_size + 1)
        self.learning_rate = learning_rate
        self.epochs = epochs

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        return 1 if summation > 0 else 0

    def train(self, training_inputs, labels):
        for _ in range(self.epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                self.weights[1:] += self.learning_rate * (label - prediction) * inputs
                self.weights[0] += self.learning_rate * (label - prediction)
```

In [25]:

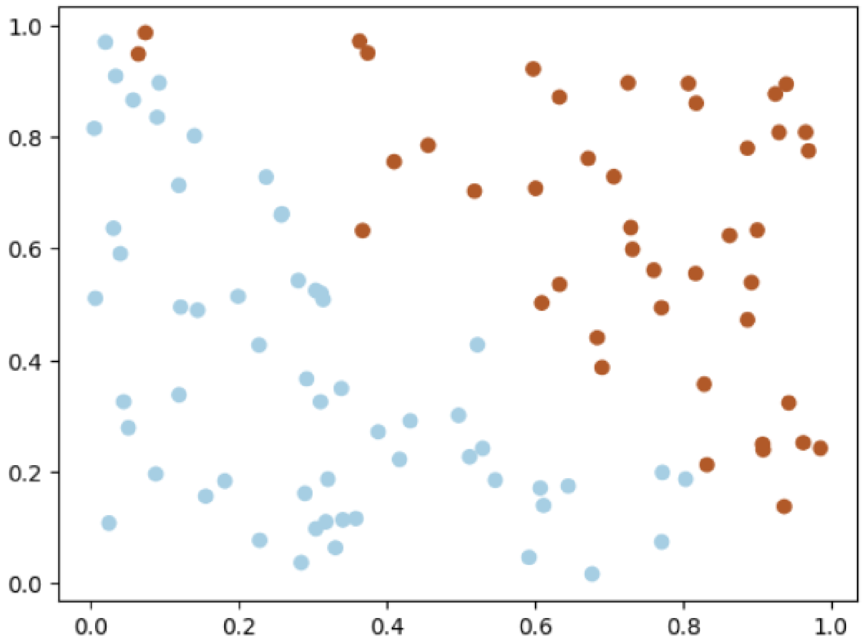
```
# Example usage
# Generate random linearly separable data
np.random.seed(42)
data = np.random.rand(100, 2)
labels = (data[:, 0] + data[:, 1] > 1).astype(int)
```

In [27]:

```
# Create and train the perceptron
perceptron = Perceptron(input_size=2)
perceptron.train(data, labels)
```

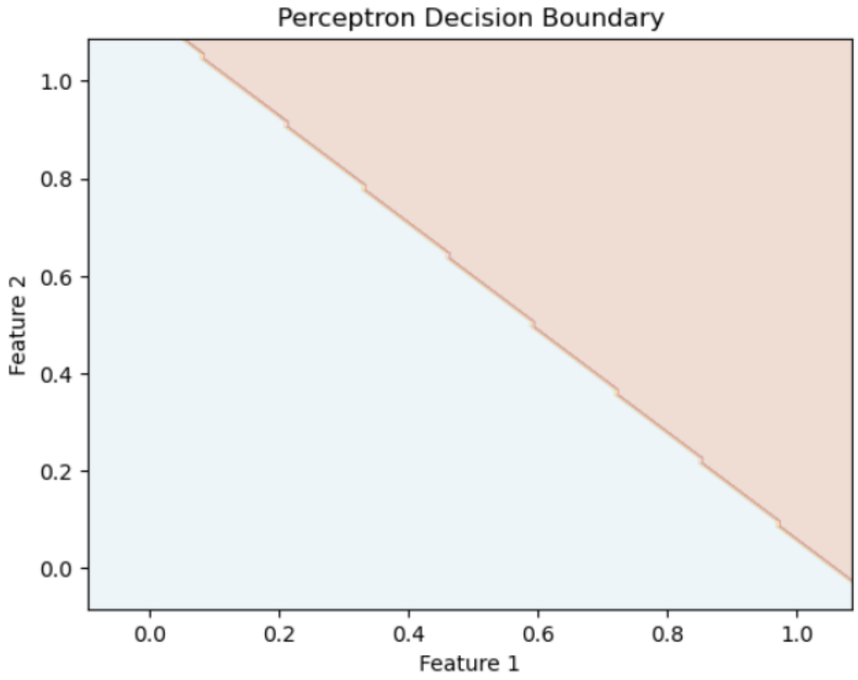
In [28]:

```
plt.scatter(data[:, 0], data[:, 1], c=labels, cmap=plt.cm.Paired)
x_min, x_max = data[:, 0].min() - 0.1, data[:, 0].max() + 0.1
y_min, y_max = data[:, 1].min() - 0.1, data[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
```



In [29]:

```
Z = np.array([perceptron.predict(np.array([x, y])) for x, y in zip(xx.ravel(), yy.ravel())])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.2, cmap=plt.cm.Paired)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Perceptron Decision Boundary')
plt.show()
```



EXPERIMENT-9:

Aim: To implement Multilayer Perceptron.

A Multilayer Perceptron (MLP) is a type of neural network characterized by its architecture, which includes an input layer, one or more hidden layers, and an output layer. Employing nonlinear activation functions, such as ReLU for hidden layers and sigmoid or softmax for the output layer, MLPs are adept at learning intricate patterns in data.

In [1]:
```python
import numpy as np
```

In [3]:
```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

def relu(x):
    return np.maximum(0, x)
def relu_derivative(x):
    return np.where(x > 0, 1, 0)
```

In [10]:
```python
class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, inputs):
        # Forward pass
        self.hidden_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = relu(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.final_output = sigmoid(self.final_input)
        return self.final_output

    def backward(self, inputs, targets, learning_rate):
        # Backward pass
        output_error = targets - self.final_output
        output_delta = output_error * sigmoid_derivative(self.final_output)

        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * relu_derivative(self.hidden_output)

        # Update weights and biases
        self.weights_hidden_output += self.hidden_output.T.dot(output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += inputs.T.dot(hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

```python
def backward(self, inputs, targets, learning_rate):
    # Backward pass
    output_error = targets - self.final_output
    output_delta = output_error * sigmoid_derivative(self.final_output)

    hidden_error = output_delta.dot(self.weights_hidden_output.T)
    hidden_delta = hidden_error * relu_derivative(self.hidden_output)

    # Update weights and biases
    self.weights_hidden_output += self.hidden_output.T.dot(output_delta) * learning_rate
    self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
    self.weights_input_hidden += inputs.T.dot(hidden_delta) * learning_rate
    self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

def train(self, inputs, targets, epochs, learning_rate):
    for epoch in range(epochs):
        # Forward and backward pass for each training example
        for input_data, target_data in zip(inputs, targets):
            input_data = input_data.reshape(1, -1)
            target_data = target_data.reshape(1, -1)

            # Forward pass
            output = self.forward(input_data)

            # Backward pass
            self.backward(input_data, target_data, learning_rate)

        if (epoch + 1) % 100 == 0:
            loss = np.mean(np.square(targets - self.predict(inputs)))
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss:.4f}')

def predict(self, inputs):
    # Make predictions using the trained model
    predictions = []
    for input_data in inputs:
        input_data = input_data.reshape(1, -1)
        output = self.forward(input_data)
        predictions.append(output.flatten())
    return np.array(predictions)
```

In [11]:

```python
# XOR problem
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])

# Create and train the MLP
mlp = MLP(input_size=2, hidden_size=4, output_size=1)
mlp.train(inputs, targets, epochs=1000, learning_rate=0.01)

# Make predictions
predictions = mlp.predict(inputs)
print("Predictions:")
print(predictions)
```

```
Epoch [100/1000], Loss: 0.2757
Epoch [200/1000], Loss: 0.2669
Epoch [300/1000], Loss: 0.2609
Epoch [400/1000], Loss: 0.2574
Epoch [500/1000], Loss: 0.2553
Epoch [600/1000], Loss: 0.2542
Epoch [700/1000], Loss: 0.2535
Epoch [800/1000], Loss: 0.2530
Epoch [900/1000], Loss: 0.2527
Epoch [1000/1000], Loss: 0.2524
Predictions:
[[0.4286887 ]
 [0.4900149 ]
 [0.50322554]
 [0.56467559]]
```