RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

RAJALAKSHMI
ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

AI23331 FUNDAMENTALS OF MACHINE LEARNING LAB

Laboratory Record Notebook

Name : KAVYASHRI Y

Year / Branch / Section : 2ND YEAR / AIML / B

University Register No : 2116231501076

College Roll No : 231501076

Semester : III

Academic Year : 2024 - 2025

INDEX PAGE

SL.NO	DATE	NAME OF THE EXPERIMENT
01	23.08.2024	A PYTHON PROGRAM TO IMPLEMENT UNIVARIATE REGRESSION,BIVARIATE REGRESSION AND MULTIVARIATE REGRESSION.
02	30.08.2024	A PYTHON PROGRAM TO IMPLEMENT SIMPLE LINEAR REGRESSION USING LEAST SQUARE METHOD
03	06.09.2024	A PYTHON PROGRAM TO IMPLEMENT LOGISTIC MODEL
04	13.09.2024	A PYTHON PROGRAM TO IMPLEMENT SINGLE LAYER PERCEPTRON
05	20.09.2024	A PYTHON PROGRAM TO IMPLEMENT MULTI LAYER PERCEPTRON WITH BACKPROPAGATION
06	27.09.2024	A PYTHON PROGRAM TO DO FACE RECOGNITION USING SVM CLASSIFIER
07	04.10.2024	A PYTHON PROGRAM TO IMPLEMENT DECISION TREE
08	18.10.2024	A PYTHON PROGRAM TO IMPLEMENT DECISION TREE
09A	25.10.2024	A PYTHON PROGRAM TO IMPLEMENT KNN MODEL .
09B	25.10.2024	A PYTHON PROGRAM TO IMPLEMENT K-MEANS MODEL
10	04.11.2024	A PYTHON PROGRAM TO IMPLEMENT DIMENSIONALITY REDUCTION -PCA.

EXPT NO: 1 **A python program to implement univariate regression**

DATE: 23.08.2024 **bivariate regression and multivariate regression.**

AIM:

To write a python program to implement univariate regression, bivariate regression and multivariate regression.

PROCEDURE:

Implementing univariate, bivariate, and multivariate regression using the Iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score
```

Step 2: Load the Iris Dataset

The Iris dataset can be loaded and display the first few rows of the dataset .

```
# Load the Iris dataset

iris = sns.load_dataset('iris')



# Display the first few rows of the dataset

print(iris.head())
```

OUTPUT :

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
# Check for missing values
print(iris.isnull().sum())

# Display the basic statistical details
print(iris.describe())
```

OUTPUT :

```
sepal_length    0
sepal_width     0
petal_length    0
petal_width     0
species         0
dtype: int64
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

Step 4: Univariate Regression

Univariate regression involves predicting one variable based on a single predictor.

4.1 : Select the Features

Choose one feature (e.g., sepal_length) and one target variable (e.g., sepal_width).

```
X_uni = iris[['sepal_length']]
y_uni = iris['sepal_width']
```

4.2 : Split the Data

Split the data into training and testing sets.
Fit the linear regression model on the training data.

```
X_uni_train, X_uni_test, y_uni_train, y_uni_test = train_test_split(X_uni, 
y_uni,
test_size=0.2, random_state=42)
```

4.3 : Train the model

```
uni_model = LinearRegression()
uni_model.fit(X_uni_train, y_uni_train)
```

```
LinearRegression
LinearRegression()
```

4.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_uni_pred = uni_model.predict(X_uni_test)
```

4.5 : Evaluate the Model

Evaluate the model performance using metrics like Mean Squared Error (MSE) and R-squared.

```
print(f'Univariate MSE: {mean_squared_error(y_uni_test, y_uni_pred)}')
print(f'Univariate R-squared: {r2_score(y_uni_test, y_uni_pred)}')
```

OUTPUT :

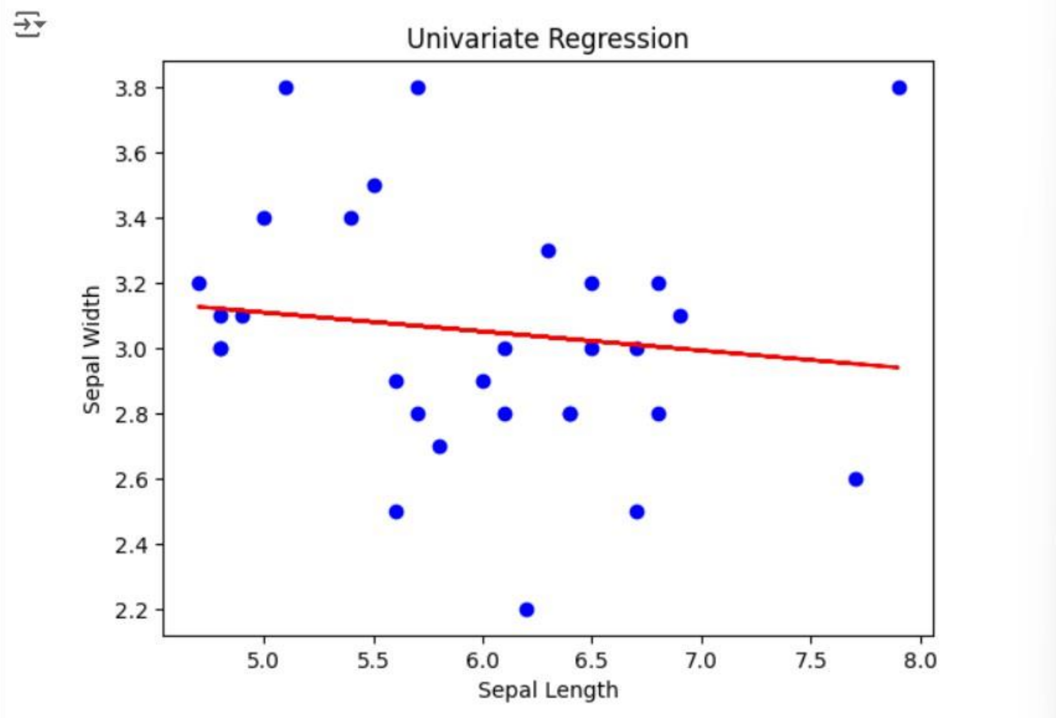
```
Univariate MSE: 0.13961895650579023
Univariate R-squared: 0.024098626473972984
```

4.6 : Visualize the Results

Visualize the relationship between the predictor and the target variable.

```
plt.scatter(X_uni_test, y_uni_test, color='blue')
plt.plot(X_uni_test, y_uni_pred, color='red')
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Univariate Regression')
plt.show()
```

OUTPUT :



Step 5 : Bivariate Regression

Bivariate regression involves predicting one variable based on two predictors.

5.1 : Select the Features

Choose two features (e.g., sepal_length, petal_length) and one target variable (e.g., sepal_width).

```
X_bi = iris[['sepal_length', 'petal_length']]

y_bi = iris['sepal_width']
```

5.2 : Split the Data

Split the data into training and testing sets.

```
X_bi_train, X_bi_test, y_bi_train, y_bi_test = train_test_split(X_bi,
y_bi,

test_size=0.2, random_state=42)
```

5.3 : Train the Model

Fit the linear regression model on the training data.

```
bi_model = LinearRegression()

bi_model.fit(X_bi_train, y_bi_train)
```

OUTPUT :

```
LinearRegression
LinearRegression()
```

5.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_bi_pred = bi_model.predict(X_bi_test)
```

5.5 : Evaluate the Model

Evaluate the model performance using metrics like MSE and R-squared.

```
print(f'Bivariate MSE: {mean_squared_error(y_bi_test, y_bi_pred)}')
print(f'Bivariate R-squared: {r2_score(y_bi_test, y_bi_pred)}')
```

OUTPUT :

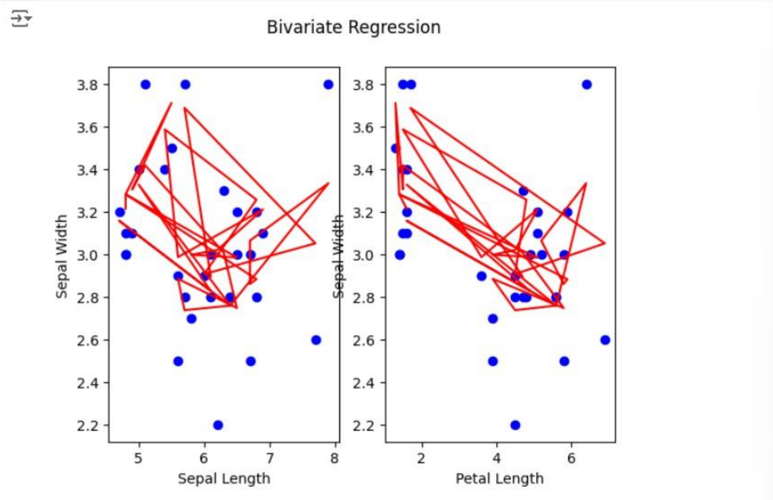
```
Bivariate MSE: 0.08308605032913309
Bivariate R-squared: 0.4192494152204116
```

5.6 : Visualize the Results

Since visualizing in 3D is challenging, we can plot the relationships between the target and each predictor separately.

```
# Sepal Length vs Sepal Width

plt.subplot(1, 2, 1)

plt.scatter(X_bi_test['sepal_length'], y_bi_test, color='blue')

plt.plot(X_bi_test['sepal_length'], y_bi_pred, color='red')

plt.xlabel('Sepal Length')

plt.ylabel('Sepal Width')

# Petal Length vs Sepal Width

plt.subplot(1, 2, 2)

plt.scatter(X_bi_test['petal_length'], y_bi_test, color='blue')

plt.plot(X_bi_test['petal_length'], y_bi_pred, color='red')

plt.xlabel('Petal Length')

plt.ylabel('Sepal Width')

plt.suptitle('Bivariate Regression')

plt.show()
```

OUTPUT :



Step 6: Multivariate Regression

Multivariate regression involves predicting one variable based on multiple predictors.

6.1 : Select the Features

Choose multiple features (e.g., sepal_length, petal_length, petal_width) and one target variable (e.g., sepal_width).

```
X_multi = iris[['sepal_length', 'petal_length', 'petal_width']]

y_multi = iris['sepal_width']
```

6.2 : Split the Data

Split the data into training and testing sets.

```
X_multi_train, X_multi_test, y_multi_train, y_multi_test =
train_test_split(X_multi,

y_multi, test_size=0.2, random_state=42)
```

6.3 : Train the Model

Fit the linear regression model on the training data.

```
multi_model = LinearRegression()
multi_model.fit(X_multi_train, y_multi_train)
```

OUTPUT :

```
LinearRegression
LinearRegression()
```

6.4 : Make Predictions

Use the model to make predictions on the test data.

```
y_multi_pred = multi_model.predict(X_multi_test)
```

6.5 : Evaluate the Model

Evaluate the model performance using metrics like MSE and R-squared.

```
    print(f'Multivariate MSE: {mean_squared_error(y_multi_test, y_multi_pred)}')
    print(f'Multivariate R-squared: {r2_score(y_multi_test, y_multi_pred)}')
```

OUTPUT :

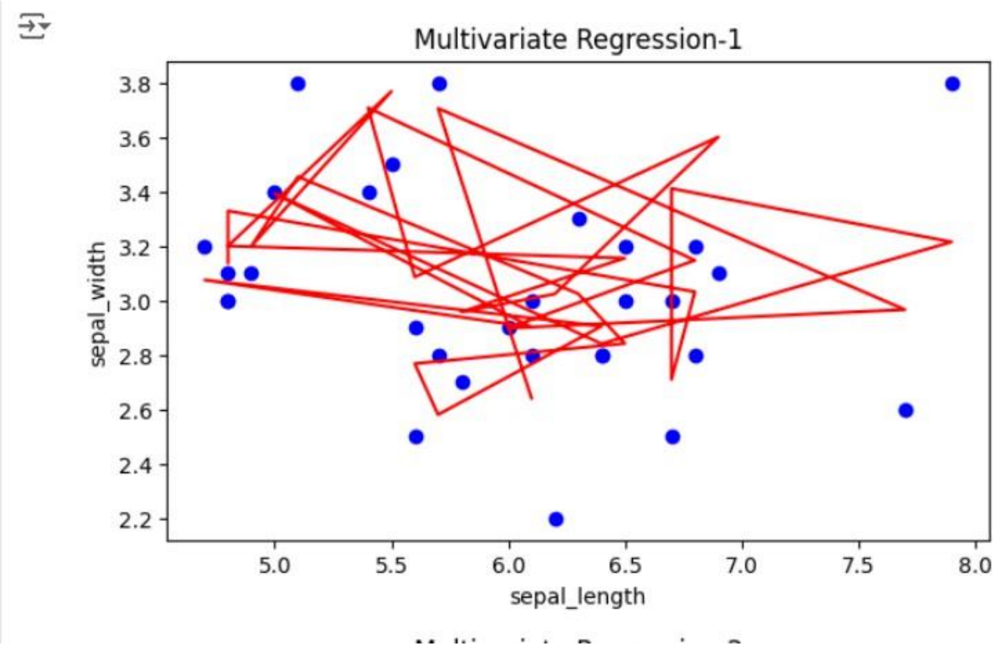
```
Multivariate MSE: 0.0868353771078583
Multivariate R-squared: 0.39304256448374897
```

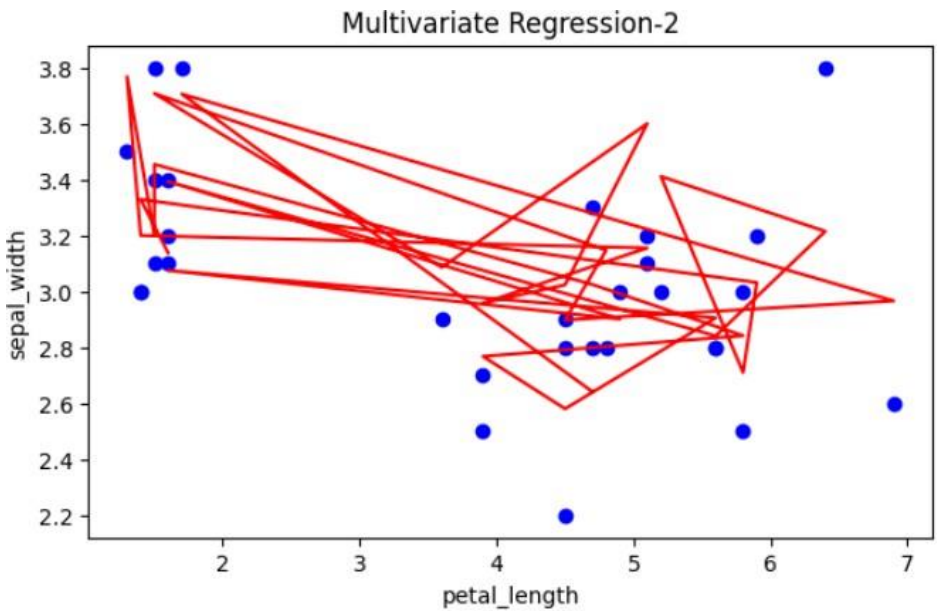
Step 7: Visualize the multivariate regression

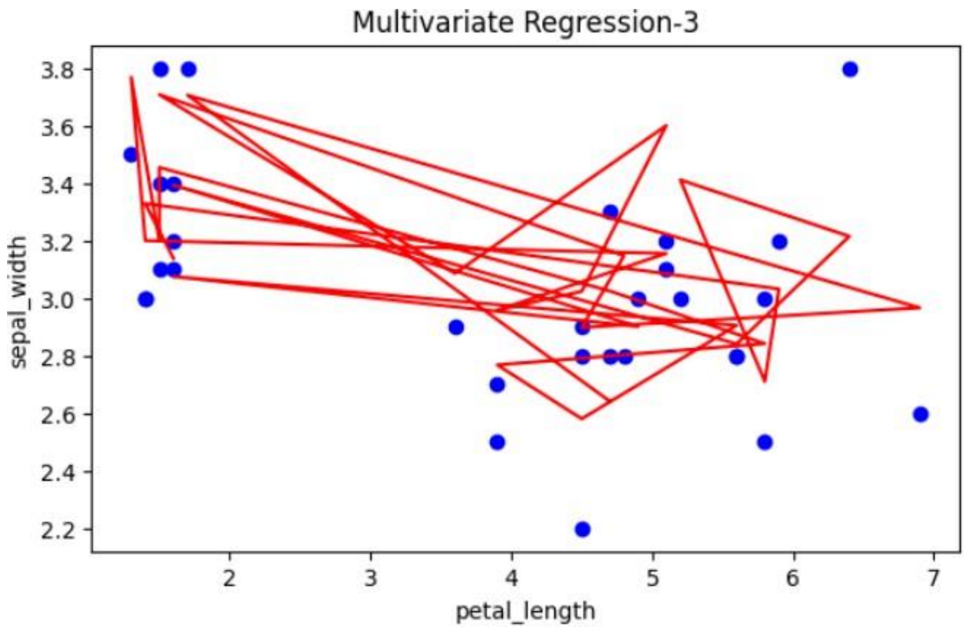
```
plt.figure(figsize=(15,4))

plt.subplot(1, 2, 1)

plt.scatter(X_multi_test['sepal_length'], y_multi_test, color='blue')

plt.plot(X_multi_test['sepal_length'], y_multi_pred, color='red')

plt.xlabel('sepal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-1')

plt.show()

plt.figure(figsize=(15,4))

plt.subplot(1, 2, 1)

plt.scatter(X_multi_test['petal_length'], y_multi_test, color='blue')

plt.plot(X_multi_test['petal_length'], y_multi_pred, color='red')

plt.xlabel('petal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-2')

plt.show()

plt.figure(figsize=(15,4))

plt.subplot(1, 2, 2 )

plt.scatter(X_multi_test['petal_length'], y_multi_test, color='blue')
```

```
plt.plot(X_multi_test['petal_length'], y_multi_pred, color='red')

plt.xlabel('petal_length')

plt.ylabel('sepal_width')

plt.title('Multivariate Regression-3')

plt.show()
```

OUTPUT :




Multivariate Regression-2
sepal_width
petal_length
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2
3
4
5
6
7


Multivariate Regression-3
sepal_width
petal_length
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
2.2
2
3
4
5
6
7

Step 8: Interpret the Results

After implementing and evaluating the models, interpret the coefficients to understand the influence of each predictor on the target variable.

```
print('Univariate Coefficients:', uni_model.coef_)
print('Bivariate Coefficients:', bi_model.coef_)
print('Multivariate Coefficients:', multi_model.coef_)
```

OUTPUT :

```
Univariate Coefficients: [-0.05829418]
Bivariate Coefficients: [ 0.56420418 -0.33942806]
Multivariate Coefficients: [ 0.62934965 -0.63196673  0.6440201 ]
```

RESULT:

This step-by-step process will help us to implement univariate, bivariate, and multivariate regression models using the Iris dataset and analyse their performance.

EXPT NO : 2 **A python program to implement Simple linear**

DATE: 30.08.2024 **Regression using Least Square Method**

AIM:

To write a python program to implement Simple linear regression using Least Square Method.

PROCEDURE:

Implementing Simple linear regression using Least Square method using the headbrain dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import pandas as pd

import matplotlib.pyplot as plt

import numpy as np
```

Step 2: Load the Iris Dataset

The HeadBrain dataset can be loaded.

```
data = pd.read_csv('/content/headbrain.csv')
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
x,y=np.array(list(data['Head Size(cm^3)'])),np.array(list(data['Brain Weight(grams)']))

print(x[:5],y[:5])
```

OUTPUT :

```
[4512 3738 4261 3777 4177] [1530 1297 1335 1282 1590]
```

Step 4 :Compute the Least Squares Solution

Apply the least squares formula to find the regression coefficients.

```
def get_line(x,y):

 x_m,y_m = np.mean(x), np.mean(y)

 print(x_m,y_m)

 x_d,y_d=x-x_m,y-y_m

 m = np.sum(x_d*y_d)/np.sum(x_d**2)

 c = y_m - (m*x_m)

 print(m, c)

 return lambda x : m*x+c

lin=get_line(x,y)
```

OUTPUT :

```
3633.9915611814345 1282.873417721519
0.2634293394893993 325.5734210494428
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
def get_error(line_fuc, x, y):

 y_m = np.mean(y)

 y_pred = np.array([line_fuc(_) for _ in x])

 ss_t = np.sum((y-y_m)**2)

 ss_r = np.sum((y-y_pred)**2)

 return 1-(ss_r/ss_t)
```

```
get_error(lin, x, y)
```

```
from sklearn.linear_model import LinearRegression
x = x.reshape((len(x),1))
reg=LinearRegression()
reg=reg.fit(x, y)
print(reg.score(x, y))
```

OUTPUT :

```
1.0
```

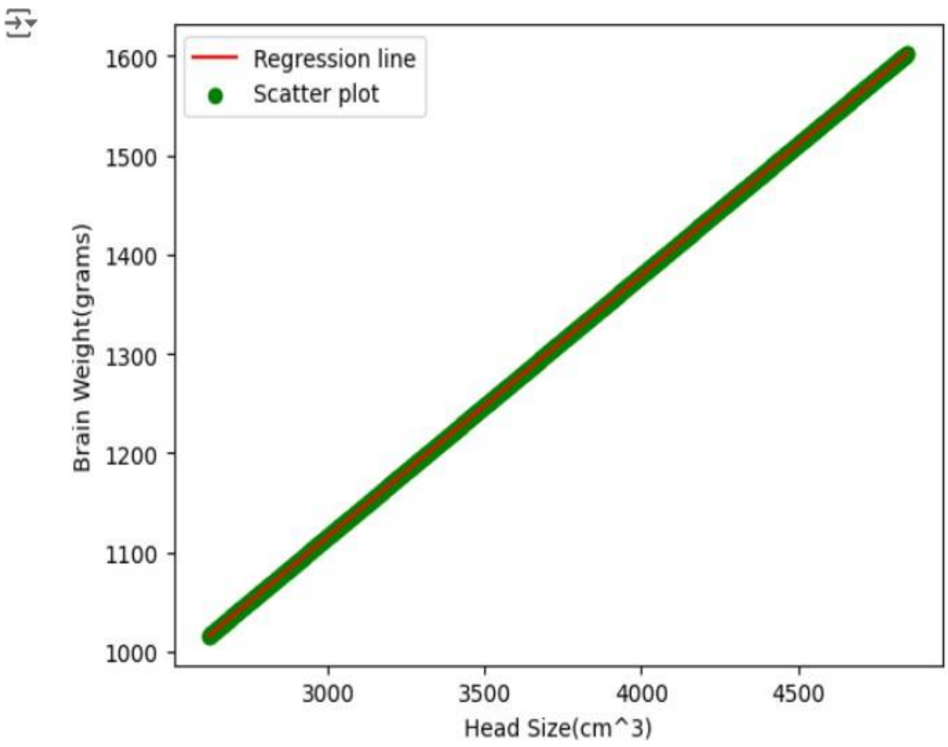
```
1.0
```

Step 6 :Visualize the Results

Plot the original data points and the fitted regression line.

```
x=np.linspace(np.min(x)-100,np.max(x)+100,1000)
y=np.array([lin(x)for x in x])
plt.plot(x, y, color='red', label='Regression line')
plt.scatter(x, y, color='green', label='Scatter plot')
plt.xlabel('Head Size(cm^3)')
plt.ylabel('Brain Weight(grams)')
plt.legend()
plt.show()
```

OUTPUT :



RESULT:

This step-by-step process will help us to implement least square regression models using the HeadBrain dataset and analyze their performance.

EXPT NO : 3 **A python program to implement Logistic Model**

DATE: 06.09.2024

AIM:

To write a python program to implement a Logistic Model.

PROCEDURE:

Implementing Logistic method using the iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualisation, and model building.

```
# Step 1: Import Necessary Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
```

Step 2: Load the Iris Dataset

The iris dataset can be loaded.

```
# Step 2: Load the Dataset

# For this example, we'll use a built-in dataset from sklearn. You can
replace it with your dataset.

from sklearn.datasets import load_iris



# Load the iris dataset
```

```
data = load_iris()

X = data.data

y = (data.target == 0).astype(int)  # For binary classification
(classifying Iris-setosa)
```

Step 3: Data Preprocessing

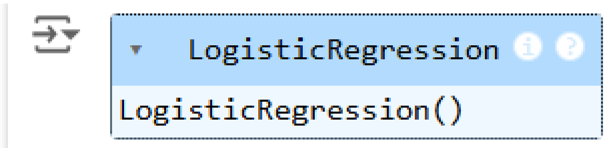
Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
# Step 3: Prepare the Data

# Split the dataset 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)
```

Step 4 : Train a Model

```
# Step 4: Create and Train the Model

model = LogisticRegression()

model.fit(X_train, y_train)
```

OUTPUT :



Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
# Step 5: Make Predictions

y_pred = model.predict(X_test)
```

Step 6 : Evaluate the Model

Evaluate the model performance.

```
# Step 6: Evaluate the Model
```

```
accuracy = accuracy_score(y_test, y_pred)

conf_matrix = confusion_matrix(y_test, y_pred)

class_report = classification_report(y_test, y_pred)

# Print evaluation metrics

print(f"Accuracy: {accuracy}")

print("Confusion Matrix:")

print(conf_matrix)

print("Classification Report:")

print(class_report)
```

OUTPUT :

```
Accuracy: 1.0
Confusion Matrix:
[[20  0]
 [ 0 10]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        20
           1       1.00      1.00      1.00        10

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

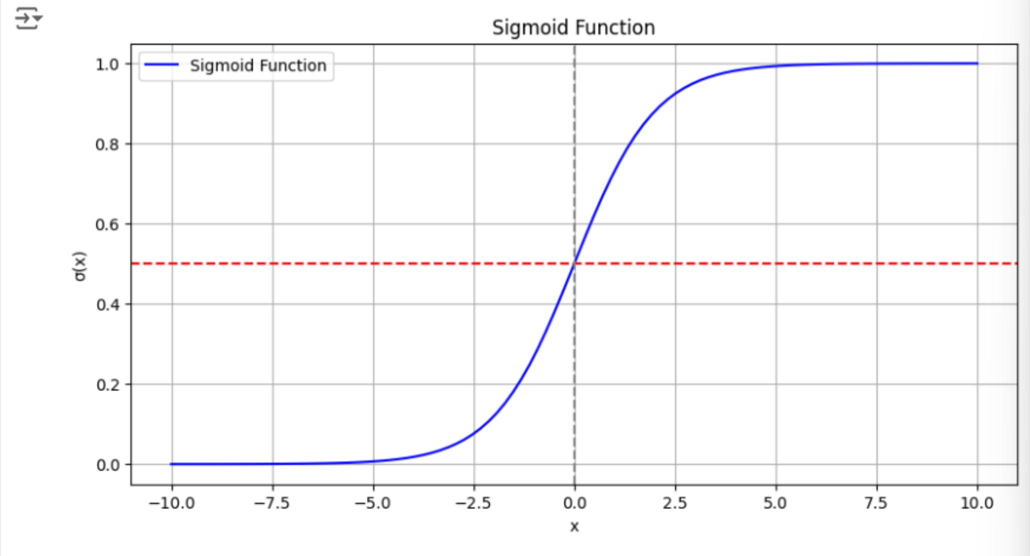
Step 7 :Visualize the Results

Plot the original data points and the fitted regression line.

```
# Step 7: Visualize Results (Optional)

x_values = np.linspace(-10, 10, 100)

sigmoid_values = 1 / (1 + np.exp(-x_values))



# Plot the sigmoid function

plt.figure(figsize=(10, 5))

plt.plot(x_values, sigmoid_values, label='Sigmoid Function', color='blue')
```

```
plt.title('Sigmoid Function')

plt.xlabel('x')

plt.ylabel('σ(x)')

plt.grid()

plt.axhline(0.5, color='red', linestyle='--')  # Line at y=0.5

plt.axvline(0, color='gray', linestyle='--')  # Line at x=0

plt.legend()

plt.show()
```

OUTPUT :



RESULT:

This step-by-step process will help us to implement Logistic models using the Iris dataset and analyze their performance.

EXPT NO : 4 **A python program to implement Single Layer**

DATE: 13.09.2024 **Perceptron**

AIM:

To write a python program to implement Single layer perceptron.

PROCEDURE:

Implementing Single layer perceptron method using the Keras dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

from tensorflow import keras

import matplotlib.pyplot as plt
```

Step 2: Load the Keras Dataset

The Keras dataset can be loaded.

```
(X_train,y_train),(X_test,y_test)=keras.datasets.mnist.load_data()
```

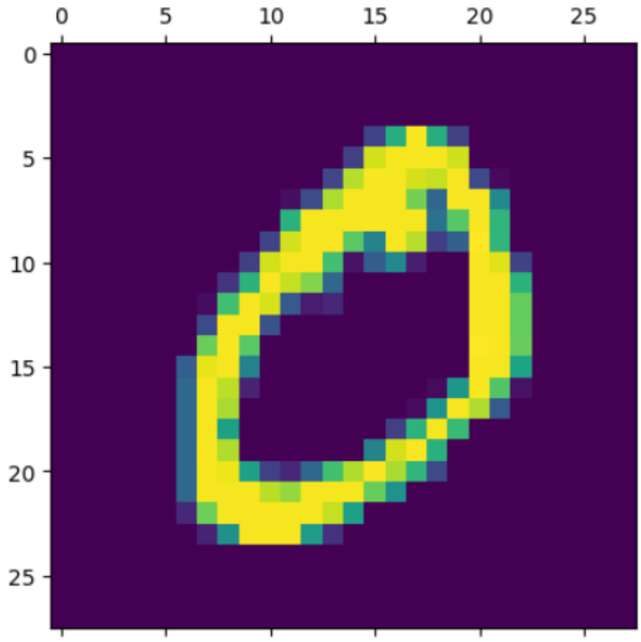
Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
print(f"Training set:{X_train.shape}")

print(f"Testing set:{X_test.shape}")



print(X_train[1].shape)

plt.matshow(X_train[1])
```

OUTPUT :

```
Training set:(60000, 28, 28)
Testing set:(10000, 28, 28)
(28, 28)
<matplotlib.image.AxesImage at 0x7b83d00435e0>
```



Step 4 : Train a Model

```
#Normalizing the dataset

x_train=X_train/255

x_test=X_test/255


#Flatting the dataset in order to compute for model building

x_train_flatten=x_train.reshape(len(x_train),28*28)

x_test_flatten=x_test.reshape(len(x_test),28*28)

x_train_flatten.shape
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
model=keras.Sequential([
```

```
    keras.layers.Dense(10,input_shape=(784,),

                      activation='sigmoid')

])


model.compile(

     optimizer='adam',

     loss='sparse_categorical_crossentropy',

     metrics=['accuracy'])


model.fit(x_train_flatten,y_train,epochs=5


          )
```

OUTPUT :

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8180 - loss: 0.7118
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.9148 - loss: 0.3101
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 956us/step - accuracy: 0.9238 - loss: 0.2769
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 2s 940us/step - accuracy: 0.9250 - loss: 0.2744
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 990us/step - accuracy: 0.9239 - loss: 0.2706
<keras.src.callbacks.history.History at 0x7b83d00c6a70>
```

Step 6 : Evaluate the Model

Evaluate the model performance.

```
model.evaluate(x_test_flatten,y_test)
```

OUTPUT :

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9138 - loss: 0.3021
[0.2668659687042236З, 0.9257000088691711]
```

RESULT:

This step-by-step process will help us to implement Single Layer Perceptron models using the Keras dataset and analyze their performance.

EXPT NO : 5 **A python program to implement Multi Layer**

DATE: 20.09.2024 **Perceptron With Backpropagation**

AIM:

To write a python program to implement Multilayer perceptron with backpropagation .

PROCEDURE:

Implementing Multilayer perceptron with backpropagation using the Keras dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
# importing modules

import tensorflow as tf

import numpy as np

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Activation

import matplotlib.pyplot as plt
```

Step 2: Load the Keras Dataset

The Keras dataset can be loaded.

```
 (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data()
```

OUTPUT :

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Step 3: Data Preprocessing

Ensure the data is clean and ready for modeling. Since the Iris dataset is clean, minimal preprocessing is needed.

```
# Cast the records into float values

x_train = x_train.astype('float32')

x_test = x_test.astype('float32')



# normalize image pixel values by dividing

# by 255

gray_scale = 255

x_train /= gray_scale

x_test /= gray_scale



print("Feature matrix:", x_train.shape)

print("Target matrix:", x_test.shape)

print("Feature matrix:", y_train.shape)

print("Target matrix:", y_test.shape)
```

OUTPUT :

```
Feature matrix: (60000, 28, 28)
Target matrix: (10000, 28, 28)
Feature matrix: (60000,)
Target matrix: (10000,)
```

Step 4 : Train a Model

```
model = Sequential([
```

```
  # reshape 28 row * 28 column data to 28*28 rows

  Flatten(input_shape=(28, 28)),



  # dense layer 1

  Dense(256, activation='sigmoid'),



  # dense layer 2

  Dense(128, activation='sigmoid'),



  # output layer

  Dense(10, activation='sigmoid'),

])
```

OUTPUT:

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning:
  super().__init__(**kwargs)
```

Step 5 : Make Predictions

Use the model to make predictions based on the independent variable.

```
model.compile(optimizer='adam',

      loss='sparse_categorical_crossentropy',

      metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10,

    batch_size=2000,

    validation_split=0.2)
```

OUTPUT:

```
Epoch 1/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 5s 115ms/step - accuracy: 0.3546 - loss: 2.1596 - val_accuracy: 0.68
Epoch 2/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 4s 53ms/step - accuracy: 0.7116 - loss: 1.3743 - val_accuracy: 0.826
Epoch 3/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 53ms/step - accuracy: 0.8221 - loss: 0.8221 - val_accuracy: 0.872
Epoch 4/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 65ms/step - accuracy: 0.8720 - loss: 0.5676 - val_accuracy: 0.892
Epoch 5/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 99ms/step - accuracy: 0.8907 - loss: 0.4444 - val_accuracy: 0.902
Epoch 6/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 102ms/step - accuracy: 0.8993 - loss: 0.3852 - val_accuracy: 0.91
Epoch 7/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 104ms/step - accuracy: 0.9088 - loss: 0.3416 - val_accuracy: 0.91
Epoch 8/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - accuracy: 0.9119 - loss: 0.3188 - val_accuracy: 0.922
Epoch 9/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step - accuracy: 0.9191 - loss: 0.2911 - val_accuracy: 0.926
Epoch 10/10
24/24 ━━━━━━━━━━━━━━━━━━━━ 3s 99ms/step - accuracy: 0.9245 - loss: 0.2704 - val_accuracy: 0.929
<keras.src.callbacks.history.History at 0x7d9ca1406a40>
```

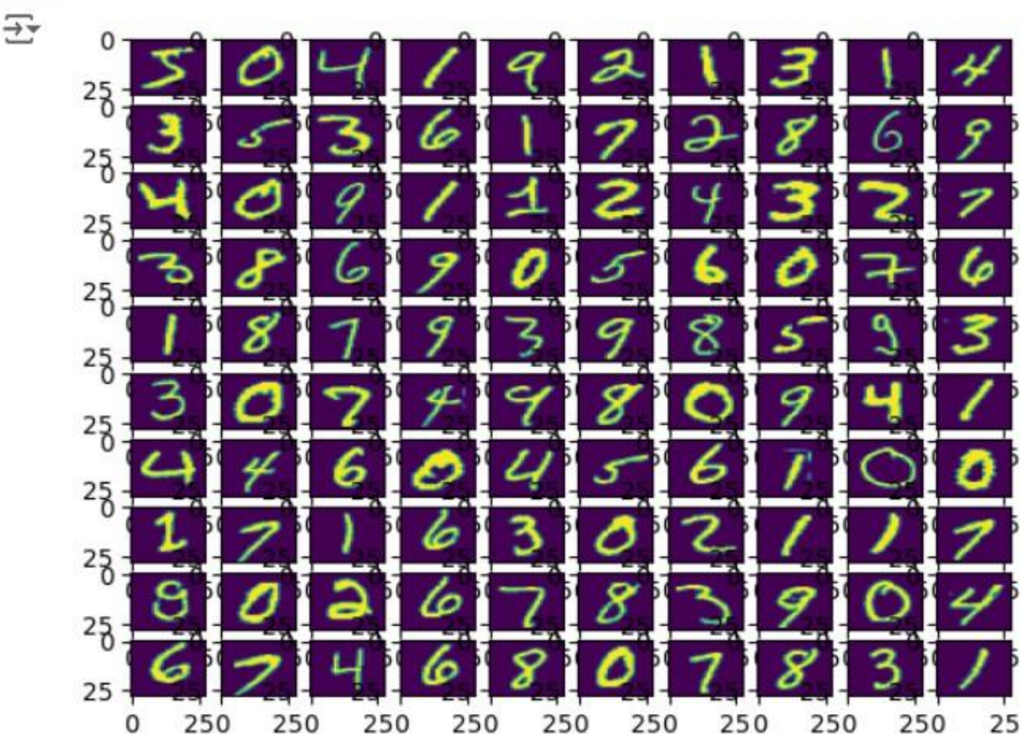
Step 6 : Evaluate the Model

Evaluate the model performance.

```
results = model.evaluate(x_test, y_test, verbose = 0)

print('test loss, test acc:', results)

fig, ax = plt.subplots(10, 10)

k = 0

for i in range(10):

  for j in range(10):

    ax[i][j].imshow(x_train[k].reshape(28, 28),

            aspect='auto')

    k += 1

plt.show()
```

OUTPUT :

```
test loss, test acc: [0.2589016258716583, 0.9277999997138977]
```



RESULT:

This step-by-step process will help us to implement MultiLayer Perceptron with Backpropagation models using the Keras dataset and analyze their performance.

EXPT NO: 6 **A python program to do face recognition using**

DATE: 27.09.2024 **SVM Classifier**

AIM:

To write a python program to implement face recognition using the SVM Classifier

PROCEDURE:

Implementing face recognition using the SVM Classifier using the cat and dog dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import pandas as pd

import imageio

import os

from skimage.transform import resize

from skimage.io import imread

import numpy as np

import matplotlib.pyplot as plt

from sklearn import svm

from sklearn.model_selection import GridSearchCV

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score

from sklearn.metrics import classification_report
```

Step 2: Load theDog and cat Dataset

The dog and cat dataset can be loaded.

```
Categories=['cats','dogs']
```

```
flat_data_arr=[] #input array

target_arr=[] #output array

datadir='/content/images'

#path which contains all the categories of images

for i in Categories:



  print(f'loading... category : {i}')

  path=os.path.join(datadir,i)

  for img in os.listdir(path):

    img_array=imread(os.path.join(path,img))

    img_resized=resize(img_array,(150,150,3))

    flat_data_arr.append(img_resized.flatten())

    target_arr.append(Categories.index(i))

  print(f'loaded category:{i} successfully')

flat_data=np.array(flat_data_arr)

target=np.array(target_arr)

#dataframe

df=pd.DataFrame(flat_data)

df['Target']=target

df.shape
```

OUTPUT :

```
(80, 67501)
```

Step 3: Separate input features and targets.

```
#input data
x=df.iloc[:,:-1]
#output data
y=df.iloc[:,-1]
```

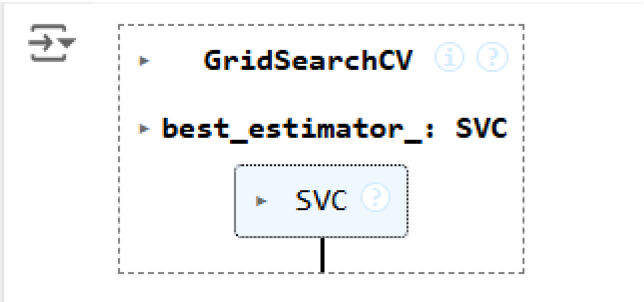
Step 4 : Separate the input features and target

```
# Splitting the data into training and testing sets
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.20,
random_state=77, stratify=y)
```

Step 5 : Build and train the model

```
# Defining the parameters grid for GridSearchCV
param_grid={'C':[0.1,1,10,100],
       'gamma':[0.0001,0.001,0.1,1],
       'kernel':['rbf','poly']}

# Creating a support vector classifier
svc=svm.SVC(probability=True)

# Creating a model using GridSearchCV with the parameters grid
model=GridSearchCV(svc,param_grid)

# Training the model using the training data
model.fit(x_train,y_train)
```

OUTPUT :



Step 6 : Model evaluation

```
# Testing the model using the testing data
y_pred = model.predict(x_test)

# Calculating the accuracy of the model
accuracy = accuracy_score(y_pred, y_test)

# Print the accuracy of the model
print(f"The model is {accuracy*100}% accurate")

print(classification_report(y_test, y_pred, target_names=['cat', 'dog']))
```

OUTPUT :

```
The model is 62.5% accurate
```

```
              precision    recall  f1-score   support

         cat       0.58      0.88      0.70         8
         dog       0.75      0.38      0.50         8

    accuracy                           0.62        16
   macro avg       0.67      0.62      0.60        16
weighted avg       0.67      0.62      0.60        16
```

Step 7 : Prediction

```
path='/content/cat.83.jpg'
img=imread(path)
plt.imshow(img)
plt.show()
img_resize=resize(img,(150,150,3))
l=[img_resize.flatten()]
probability=model.predict_proba(l)
for ind,val in enumerate(Categories):
  print(f'{val} = {probability[0][ind]*100}%')
print("The predicted image is : "+Categories[model.predict(l)[0]])
```

OUTPUT :


```
cats = 52.70216647851706%
dogs = 47.29783352148294%
The predicted image is : cat
```

RESULT :

Thus the process helps us to implement the face recognition using SVM Classifier using python program.

EXPT NO: 7 **A python program to implement Decision tree**

DATE: 04.10.2024

AIM:

To write a python program to implement a Decision tree.

PROCEDURE:

Implementing the decision tree using the Iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn import metrics

import matplotlib.pyplot as plt

from sklearn.tree import plot_tree
```

Step 2: Load the Iris Dataset

The Iris dataset can be loaded and display the first few rows of the dataset .

```
# Load the Iris dataset

iris = datasets.load_iris()

X = iris.data  # Features
```

```
y = iris.target  # Target variable
```

Step 3 : Split the data set into training and testing sets

```
# Split the dataset 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)
```

Step 4 : Create a decision tree classifier

```
# Create a Decision Tree classifier

clf = DecisionTreeClassifier(random_state=42)
```

Step 5 : Train the model :

```
# Train the model

clf.fit(X_train, y_train)
```

OUTPUT :

```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

Step 6 : Make the predictions and evaluate the model

```
# Make predictions

y_pred = clf.predict(X_test)



# Evaluate the model

accuracy = metrics.accuracy_score(y_test, y_pred)

confusion = metrics.confusion_matrix(y_test, y_pred)

classification_report = metrics.classification_report(y_test, y_pred)



print(f"Accuracy: {accuracy:.2f}")

print("Confusion Matrix:")
```

```
print(confusion)

print("Classification Report:")

print(classification_report)
```

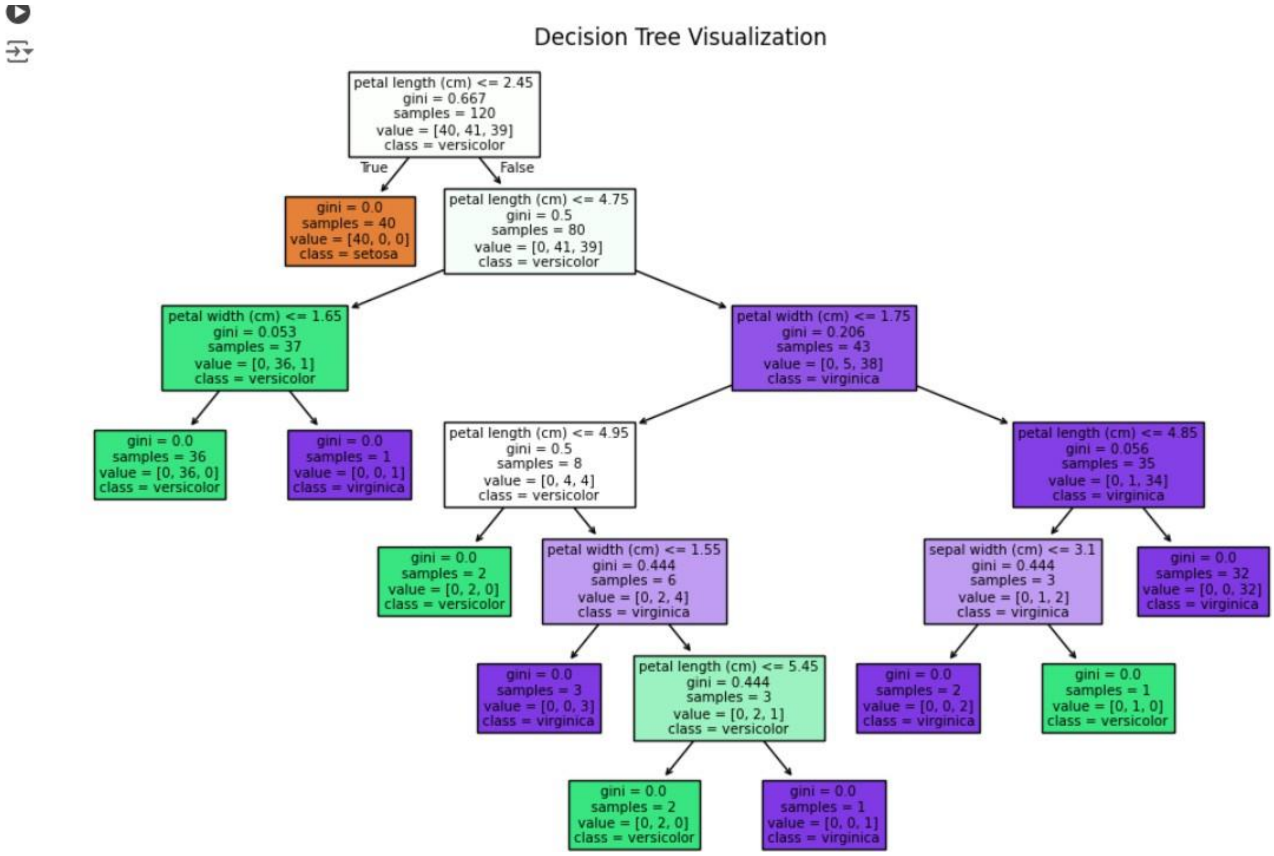
OUTPUT :

```
Accuracy: 1.00
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Step 7 : Visualize the decision tree

```
# Visualize the Decision Tree

plt.figure(figsize=(12,8))

plot_tree(clf, filled=True, feature_names=iris.feature_names,
class_names=iris.target_names)

plt.title("Decision Tree Visualization")

plt.show()
```

OUTPUT :



RESULT :

This process helps us to implement the decision tree using a python program.

EX.NO: 8 **A PYTHON PROGRAM TO IMPLEMENT**

DATE : 18.10.2024 **ADA BOOSTING**

AIM:

To write a python program to implement ADA Boosting.

PROCEDURE:

Implementing ADA Boosting using the dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

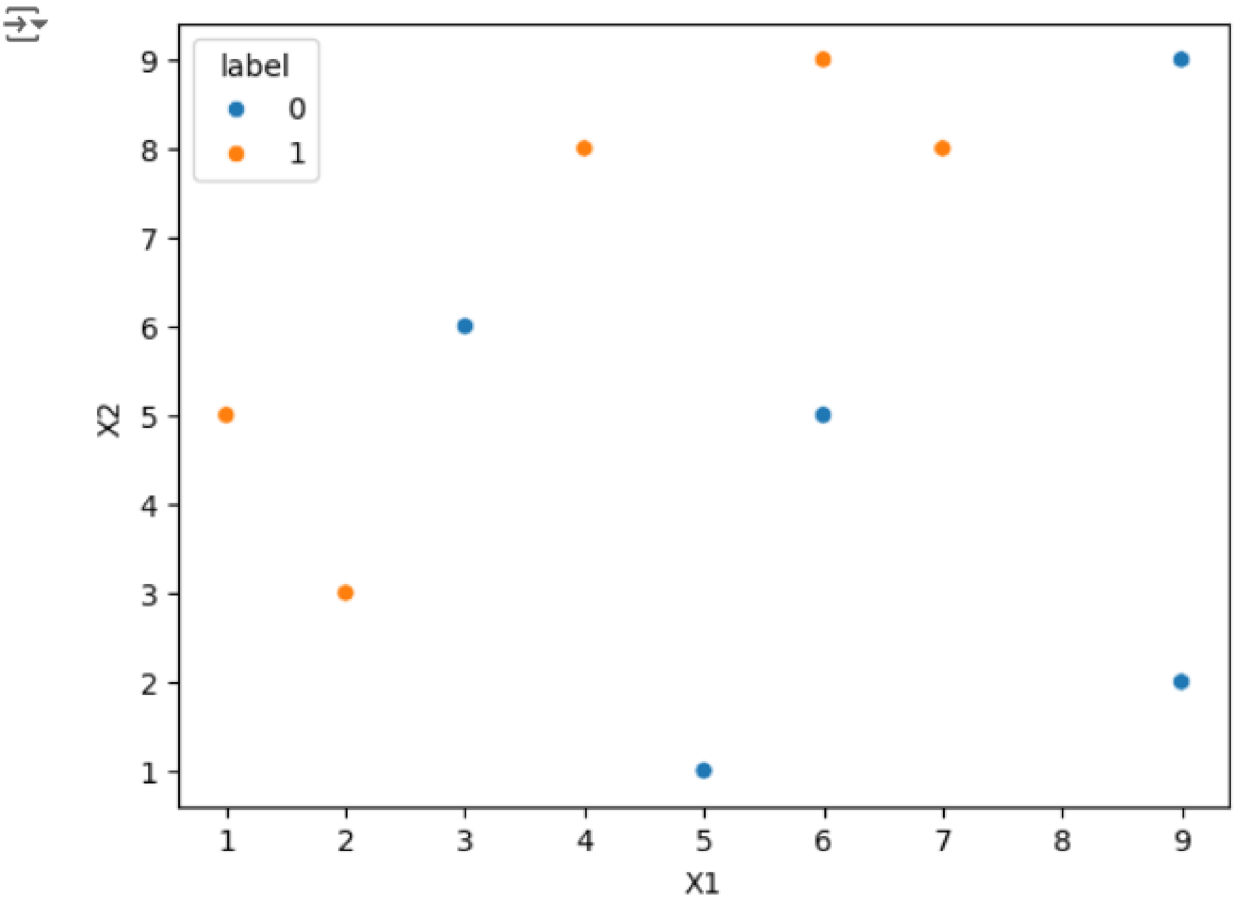
```
import numpy as np

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from mlxtend.plotting import plot_decision_regions

import seaborn as sns

from sklearn.metrics import accuracy_score
```

Step 2 : Load and prepare data

```
df = pd.DataFrame()

df['X1'] = [1, 2, 3, 4, 5, 6, 6, 7, 9, 9]

df['X2'] = [5, 3, 6, 8, 1, 9, 5, 8, 9, 2]

df['label'] = [1, 1, 0, 1, 0, 1, 0, 1, 0, 0]

sns.scatterplot(x=df['X1'], y=df['X2'], hue=df['label'])



df['weights'] = 1 / df.shape[0]

x = df.iloc[:, 0:2].values
```

```
y = df.iloc[:, 2].values
```

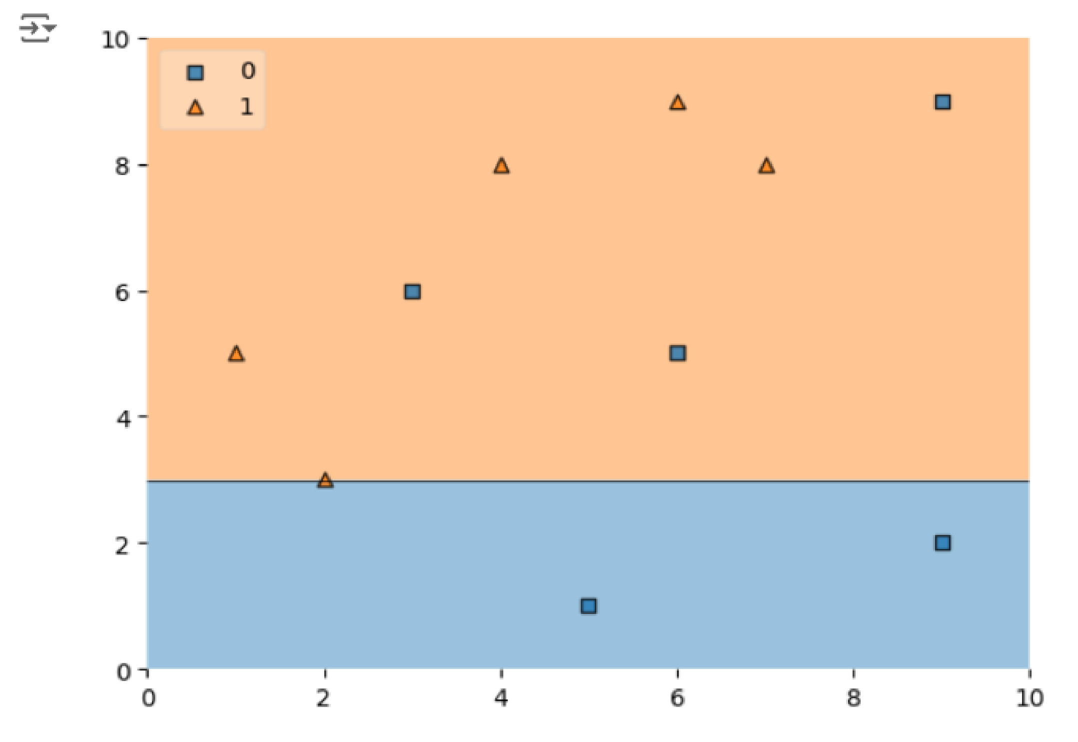
OUTPUT :



Step 3 : Train the 1st model

```
# Step 2: Train 1st Model

dt1 = DecisionTreeClassifier(max_depth=1)

dt1.fit(x, y)

plot_decision_regions(x, y, clf=dt1, legend=2)

df['y_pred'] = dt1.predict(x)
```

OUTPUT :



Step 4 : Calculate model weight

```
# Step 4: Update Weights

def update_row_weights(row, alpha=0.423):

    if row['label'] == row['y_pred']:

        return row['weights'] * np.exp(-alpha)

    else:

        return row['weights'] * np.exp(alpha)



df['updated_weights'] = df.apply(update_row_weights, axis=1)

df['normalized_weights'] = df['updated_weights'] /
df['updated_weights'].sum()

df['cumsum_upper'] = np.cumsum(df['normalized_weights'])

df['cumsum_lower'] = df['cumsum_upper'] - df['normalized_weights']
```

Step 5 : Create new dataset

```python
# Step 5: Create New Dataset
def create_new_dataset(df):
    indices = []
    for i in range(df.shape[0]):
        a = np.random.random()
        for index, row in df.iterrows():
            if row['cumsum_upper'] > a and a > row['cumsum_lower']:
                indices.append(index)
    return indices


index_values = create_new_dataset(df)
second_df = df.iloc[index_values, [0, 1, 2, 3]]
```

Step 6 : Train 2nd model

```python
# Step 6: Train 2nd Model
dt2 = DecisionTreeClassifier(max_depth=1)
x = second_df.iloc[:, 0:2].values
y = second_df.iloc[:, 2].values
dt2.fit(x, y)
```

OUTPUT :

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=1)
```

Step 7 : Plot decision tree and calculate model weights for 2nd model

```python
# Plot the decision tree for the second model
```

```python
plot_decision_regions(x, y, clf=dt2, legend=2)

second_df['y_pred'] = dt2.predict(x)


# Step 7: Calculate Model Weight for 2nd Model

alpha2 = calculate_model_weight(0.1)

print(f"Alpha2: {alpha2}")
```

Step 8 : update weights for 2nd model

```python
# Step 8: Update Weights for 2nd Model

def update_row_weights(row, alpha=1.09):

    if row['label'] == row['y_pred']:

        return row['weights'] * np.exp(-alpha)

    else:

        return row['weights'] * np.exp(alpha)



second_df['updated_weights'] = second_df.apply(update_row_weights, axis=1)

second_df['nomalized_weights'] = second_df['updated_weights'] /
second_df['updated_weights'].sum()

second_df['cumsum_upper'] = np.cumsum(second_df['nomalized_weights'])

second_df['cumsum_lower'] = second_df['cumsum_upper'] -
second_df['nomalized_weights']
```

Step 9 : Calculate alpha for 3rd model

```python
# Step 9: Calculate Alpha for 3rd Model

alpha3 = calculate_model_weight(0.7)

print(f"Alpha3: {alpha3}")
```

```
# Step 10: Accuracy Calculation

y_true = second_df['label'].values

y_pred = second_df['y_pred'].values


# Calculate accuracy for the AdaBoost model

accuracy = accuracy_score(y_true, y_pred)

print(f"Accuracy of the AdaBoost model: {accuracy:.4f}")
```

OUTPUT :

ALPHA 3: -0.4236489301936017

Accuracy of the Ada Boosting model : 0.80000

RESULT :

Thus the python program to implement Adaboosting has been executed successfully and the results have been verified.

EXPT NO: 9A **A python program to implement**

DATE: 25.10.2024 **KNN MODEL .**

AIM:

To write a python program to implement KNN Model.

PROCEDURE:

Implementing KNN Model using the mall_customer dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import classification_report, confusion_matrix

from sklearn.cluster import KMeans
```

Step 2: Load the Dataset

The mall_customer dataset can be loaded and display the first few rows of the dataset.

```
# Load the dataset

dataset = pd.read_csv('/content/Mall_Customers.csv')



# Display the first few rows of the dataset

print(dataset.head())
```

```python
# Display the dimensions of the dataset
print(f"Dataset shape: {dataset.shape}")

# Display descriptive statistics of the dataset
print(dataset.describe())
```

Step 3 : Separate the features (x) and target variable (y)

```python
# Separate the features (X) and the target variable (y)
X = dataset.iloc[:, [3, 4]].values  # We use 'Annual Income' and 'Spending Score'

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

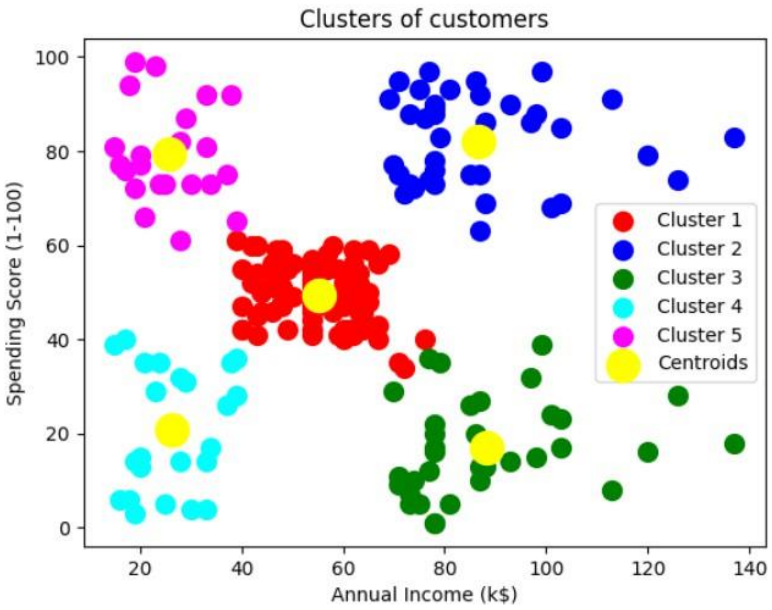
Step 4 : Visualizing the cluster of customer

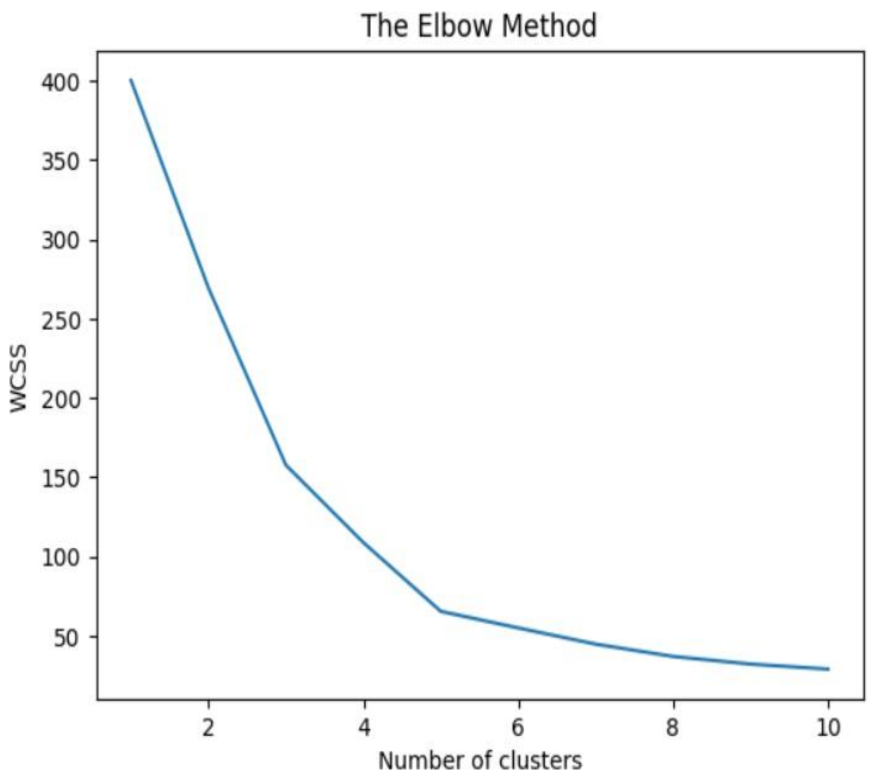
```python
# Apply KMeans clustering using the Elbow Method to find the optimal number of clusters
wcss = []  # Within-cluster sum of squares
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.plot(range(1, 11), wcss)
```

```python
plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()



# From the plot, we can observe that the optimal number of clusters is 5 (elbow point)

kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=300, n_init=10, random_state=0)

y_kmeans = kmeans.fit_predict(X_scaled)



# Visualizing the clusters of customers

plt.scatter(X_scaled[y_kmeans == 0, 0], X_scaled[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')

plt.scatter(X_scaled[y_kmeans == 1, 0], X_scaled[y_kmeans == 1, 1], s=100, c='blue', label='Cluster 2')

plt.scatter(X_scaled[y_kmeans == 2, 0], X_scaled[y_kmeans == 2, 1], s=100, c='green', label='Cluster 3')

plt.scatter(X_scaled[y_kmeans == 3, 0], X_scaled[y_kmeans == 3, 1], s=100, c='cyan', label='Cluster 4')

plt.scatter(X_scaled[y_kmeans == 4, 0], X_scaled[y_kmeans == 4, 1], s=100, c='magenta', label='Cluster 5')



# Plot the centroids

plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=300, c='yellow', label='Centroids')



plt.title('Clusters of customers')

plt.xlabel('Annual Income (k$)')
```

```
plt.ylabel('Spending Score (1-100)')

plt.legend()

plt.show()
```

OUTPUT :





RESULT :

Thus the python program to implement KNN model has been successfully implemented and the results have been verified.

EXPT NO: 9B **A python program to implement**

DATE: 25.10.2024 **K-Means Model**

AIM:

To write a python program to implement the K-means Model.

PROCEDURE:

Implementing K - means Model using the mall_customer dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
import numpy as np

import pandas as pd

from math import sqrt
```

Step 2 : load the Dataset

```
data = pd.read_csv('/content/Mall_Customers.csv')

data.head(5)
```

OUTPUT:

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

Step 3 : Preprocess the data

```
req_data = data[['Age', 'Annual Income (k$)', 'Spending Score (1-100)']]


req_data.head(5)
```

OUTPUT :

	Age	Annual Income (k$)	Spending Score (1-100)
0	19	15	39
1	21	15	81
2	20	16	6
3	23	16	77
4	31	17	40

Step 4 : Assign the data points to clusters

```
shuffle_index = np.random.permutation(req_data.shape[0])  # Shuffle the
dataset rows

req_data = req_data.iloc[shuffle_index]

req_data.head(5)
```

OUTPUT :

	Gender	Age	Annual Income (k$)	Spending Score (1-100)
14	Male	37	20	13
102	Male	67	62	59
89	Female	50	58	46
181	Female	32	97	86
183	Female	29	98	88

Step 5 : Update the clusters centers

```
train_size = int(req_data.shape[0]*0.7)  # Set 70% of the data for
training
```

```
train_df = req_data.iloc[:train_size,:]

test_df = req_data.iloc[train_size:,:]

train = train_df.values  # Convert train data to numpy array

test = test_df.values  # Convert test data to numpy array

y_true = test[:,-1]  # The target values for the test set

print('Train_Shape: ', train_df.shape)

print('Test_Shape: ', test_df.shape)

from math import sqrt


def euclidean_distance(x_test, x_train):

    distance = 0

    for i in range(len(x_test)):  # Loop through all features

        distance += (x_test[i]-x_train[i])**2

    return sqrt(distance)


def get_neighbors(x_test, x_train, num_neighbors):

    distances = []

    data = []

    for i in x_train:

        distances.append(euclidean_distance(x_test, i))

        data.append(i)

    distances = np.array(distances)

    data = np.array(data)

    sort_indexes = distances.argsort()  # Sort distances in ascending
order

    data = data[sort_indexes]  # Sort the data based on sorted distances
```

```
    return data[:num_neighbors]  # Return the closest 'num_neighbors'
neighbors

def prediction(x_test, x_train, num_neighbors):

    classes = []

    neighbors = get_neighbors(x_test, x_train, num_neighbors)

    for i in neighbors:

        classes.append(i[-1])  # The target value is the last column

    predicted = max(classes, key=classes.count)  # Return the most
frequent class (the majority vote)

    return predicted



def predict_classifier(x_test):

    classes = []

    neighbors = get_neighbors(x_test, req_data.values, 5)  # Predict using
the top 5 neighbors

    for i in neighbors:

        classes.append(i[-1])

    predicted = max(classes, key=classes.count)  # Return the majority
vote

    print(predicted)

    return predicted

def accuracy(y_true, y_pred):

    num_correct = 0

    for i in range(len(y_true)):

        if y_true[i] == y_pred[i]:  # Compare true values to predicted
values

            num_correct += 1

    accuracy = num_correct / len(y_true)  # Calculate accuracy as the
```

```
ratio of correct predictions

    return accuracy

def accuracy(y_true, y_pred):

    num_correct = 0

    for i in range(len(y_true)):

        if y_true[i] == y_pred[i]:

            num_correct += 1

    return num_correct / len(y_true)

y_pred = []

for i in test:

    y_pred.append(prediction(i, train, 5))  # Make predictions for each
test instance


# Calculate and print the accuracy

acc = accuracy(y_true, y_pred)

print(f"Accuracy: {acc * 1000:.2f}%")
```

OUTPUT :

```
Accuracy: 66.67%
```

RESULT :

Thus the python program implementing the k-means model is successful.

EXPT NO: 10 **A python program to implement Dimensionality**

DATE: 04.11.2024 **Reduction -PCA.**

AIM:

To write a python program to implement Dimensionality Reduction - PCA .

PROCEDURE:

ImplementingDimensionality reduction -pca using the Iris dataset involve the following steps:

Step 1: Import Necessary Libraries

First, import the libraries that are essential for data manipulation, visualization, and model building.

```
# Importing necessary libraries

from sklearn import datasets

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

import seaborn as sns

import matplotlib.pyplot as plt
```

Step 2: Load the Iris Dataset

The Iris dataset can be loaded and display the first few rows of the dataset

```
# Load the Iris dataset

iris = datasets.load_iris()

df = pd.DataFrame(iris['data'], columns=iris['feature_names'])



# Display the first few rows of the dataset

df.head()
```

OUTPUT :

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Step 3 : Standardize the data

```
# Standardize the features using StandardScaler

scalar = StandardScaler()

scaled_data = pd.DataFrame(scalar.fit_transform(df))   # Scaling the data



# Display the scaled data (optional)

scaled_data.head()
```

OUTPUT :

	0	1	2	3
0	-0.900681	1.019004	-1.340227	-1.315444
1	-1.143017	-0.131979	-1.340227	-1.315444
2	-1.385353	0.328414	-1.397064	-1.315444
3	-1.506521	0.098217	-1.283389	-1.315444
4	-1.021849	1.249201	-1.340227	-1.315444

Step 4 : Apply PCA

```
# Apply PCA to reduce the data to 3 components

pca = PCA(n_components=3)
```

```
pca.fit(scaled_data)  # Fit PCA on scaled data

data_pca = pca.transform(scaled_data)  # Transform the data to principal components



# Convert PCA data to a DataFrame for easier inspection

data_pca = pd.DataFrame(data_pca, columns=['PC1', 'PC2', 'PC3'])

data_pca.head()
```

OUTPUT :

	PC1	PC2	PC3
0	-2.264703	0.480027	0.127706
1	-2.080961	-0.674134	0.234609
2	-2.364229	-0.341908	-0.044201
3	-2.299384	-0.597395	-0.091290
4	-2.389842	0.646835	-0.015738

Step 5 : Explained Variance Ratio

```
# Calculate the explained variance ratio for each principal component

explained_variance = pca.explained_variance_ratio_

print(f"Explained Variance Ratio: {explained_variance}")



# This output shows how much variance each principal component explains.
```

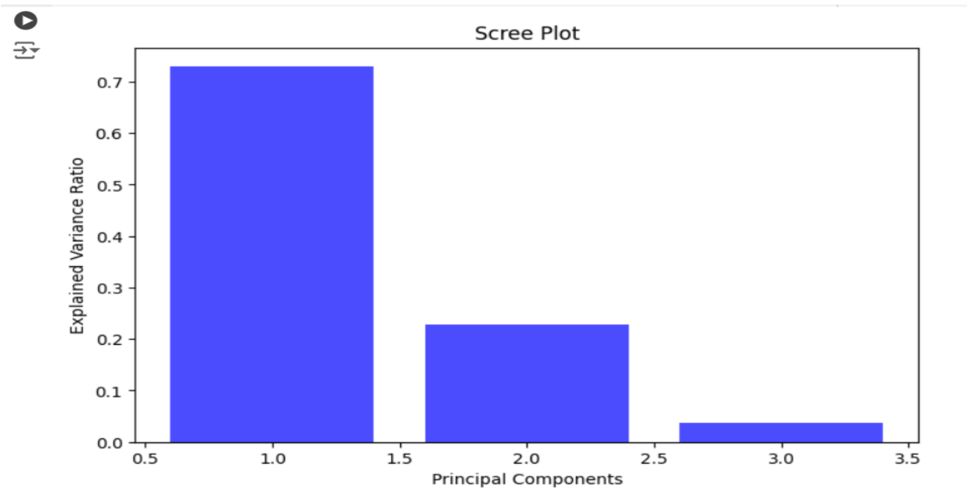
OUTPUT :

```
Explained Variance Ratio: [0.72962445 0.22850762 0.03668922]
```

Step 6 :Visualize the reduced data.

```python
# Plotting the explained variance ratio as a scree plot

plt.figure(figsize=(8, 5))

plt.bar(range(1, len(explained_variance) + 1), explained_variance,
alpha=0.7, color='blue')

plt.ylabel('Explained Variance Ratio')

plt.xlabel('Principal Components')

plt.title('Scree Plot')

plt.show()
```

OUTPUT :



RESULT :

Thus the Dimensionality Reduction has been implemented using PCA in python program Successfully.