**RAJALAKSHMI ENGINEERING COLLEGE**

# **RAJALAKSHMI NAGAR, THANDALAM – 602 105**





Name : AARTHI AH

Year / Branch / Section : 2nd YEAR / AIML A

University Register No. :2116231501001

College Roll No. : 231501001

Semester : 3rd SEMESTER

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 LAYERPERCEPTRON |
| 05 | 20.09.2024 | A PYTHON PROGRAM TO IMPLEMENT MULTI LAYERPERCEPTRON 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**

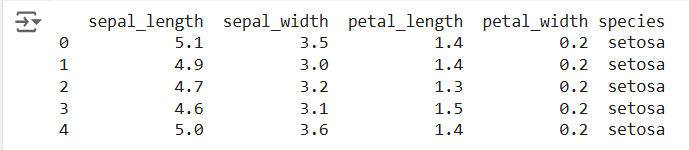
The Iris dataset can be loaded and display the first few rows of the dataset .

# Load the Iris dataset

iris = sns.load\_dataset('iris')

# Display the first few rows of the dataset

print(iris.head())

**OUTPUT :** 

# **Step 3: Data Preprocessing**

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

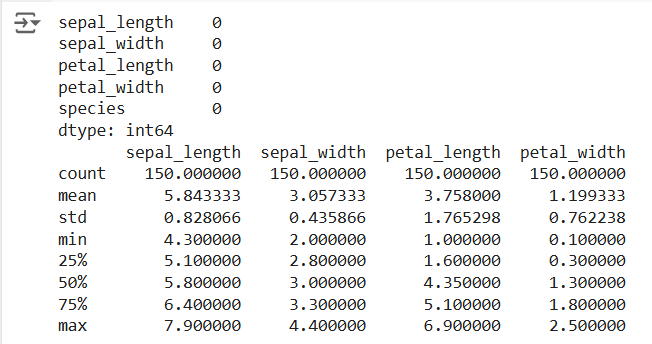
# Check for missing values

print(iris.isnull().sum())

# Display the basic statistical details

print(iris.describe())

**OUTPUT :**



# **Step 4: Univariate Regression**

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

# **: 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']

# **: Split the Data**

Split the data into training and testing sets.

Fit the linear regression model on the training data.

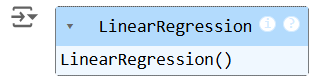
X\_uni\_train, X\_uni\_test, y\_uni\_train, y\_uni\_test = train\_test\_split(X\_uni, y\_uni,

test\_size=0.2, random\_state=42)

**4.3: Train the model**

uni\_model = LinearRegression()

uni\_model.fit(X\_uni\_train, y\_uni\_train)



# **: Make Predictions**

Use the model to make predictions on the test data.

y\_uni\_pred = uni\_model.predict(X\_uni\_test)

# **: 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 :**



# **: Visualize the Results**

Visualize the relationship between the predictor and the target variable.

plt.scatter(X\_uni\_test, y\_uni\_test, color='blue')

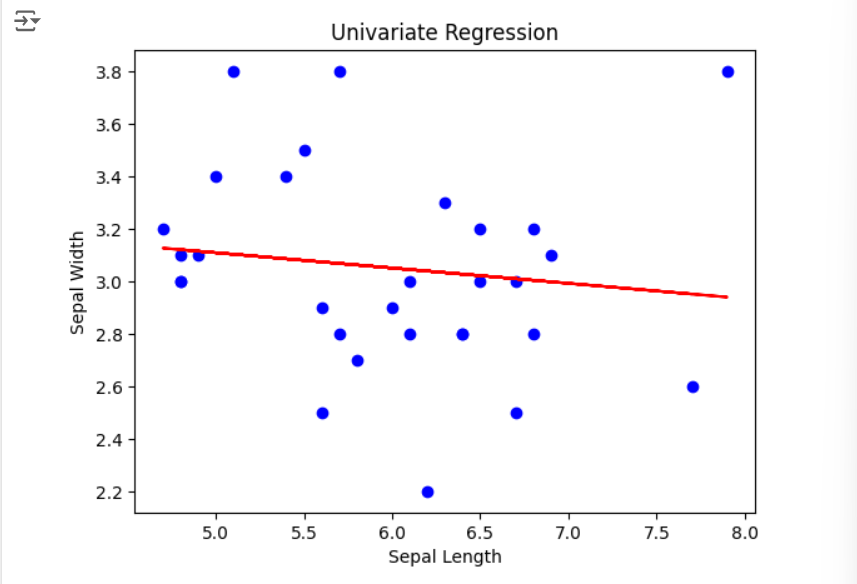
plt.plot(X\_uni\_test, y\_uni\_pred, color='red')

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.

# **: 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']

# **: 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)

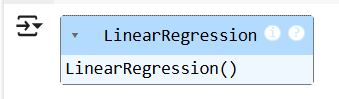
# **: Train the Model**

Fit the linear regression model on the training data.

bi\_model = LinearRegression()

bi\_model.fit(X\_bi\_train, y\_bi\_train)

**OUTPUT :**



# **: Make Predictions**

Use the model to make predictions on the test data.

y\_bi\_pred = bi\_model.predict(X\_bi\_test)

# **: 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 :



# **: 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')

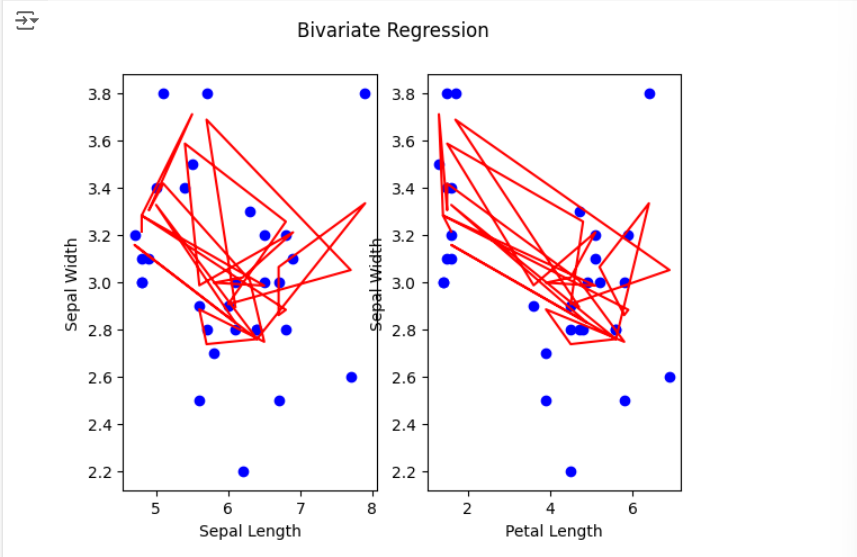
plt.xlabel('Petal Length')

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.

# **: 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']

# **: 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)

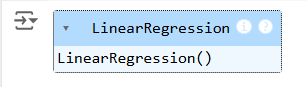
# **: Train the Model**

Fit the linear regression model on the training data.

multi\_model = LinearRegression()

multi\_model.fit(X\_multi\_train, y\_multi\_train)

**OUTPUT :**



# **: Make Predictions**

Use the model to make predictions on the test data.

y\_multi\_pred = multi\_model.predict(X\_multi\_test)

# **: 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 :**



# **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')

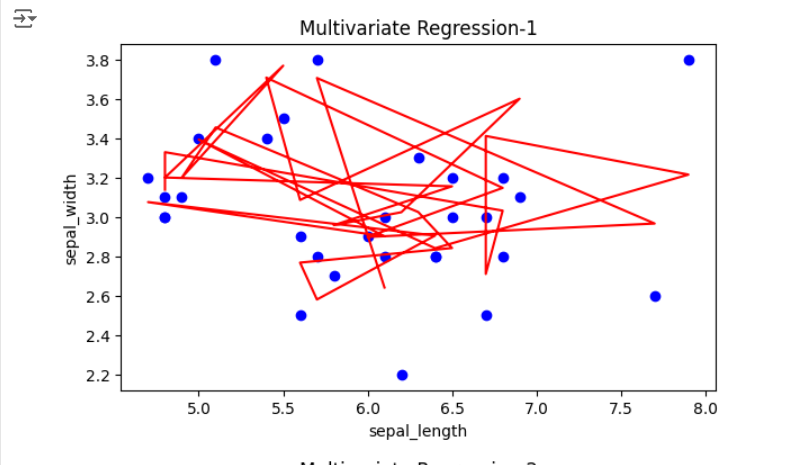
plt.xlabel('petal\_length')

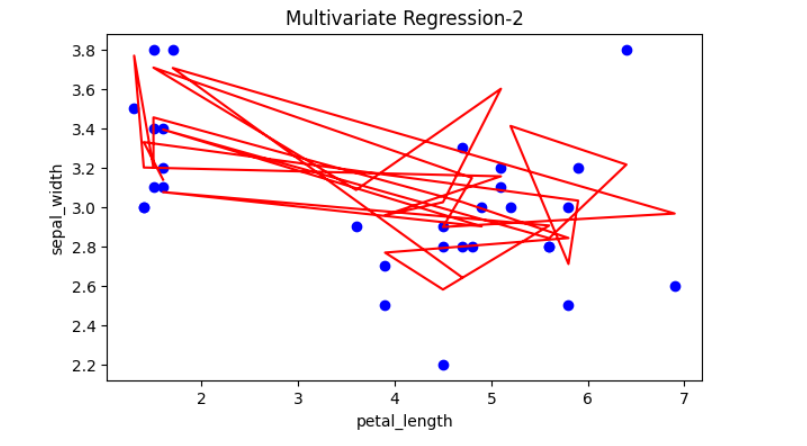
plt.ylabel('sepal\_width')

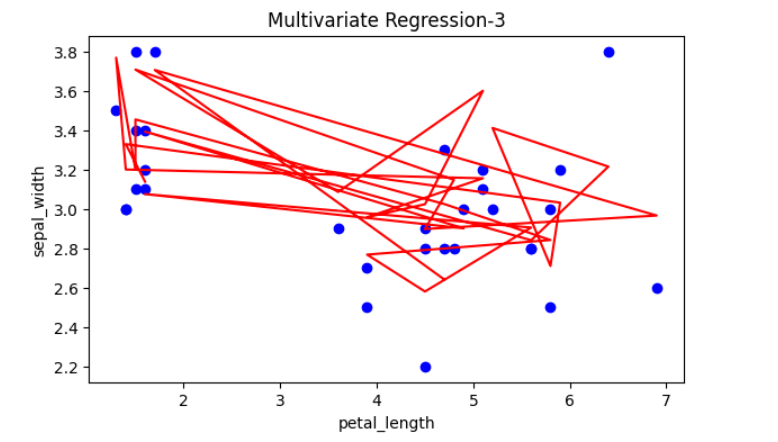
plt.title('Multivariate Regression-3')

plt.show()

**OUTPUT :**







# **Step 8: Interpret the Results**

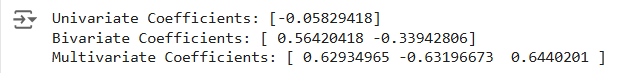
# After implementing and evaluating the models, interpret the coefficients to understand the influence of each predictor on the target variable.

print('Univariate Coefficients:', uni\_model.coef\_)

print('Bivariate Coefficients:', bi\_model.coef\_)

print('Multivariate Coefficients:', multi\_model.coef\_)

**OUTPUT :**



**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 :**



**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 :**



**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 :**

****

****

**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')

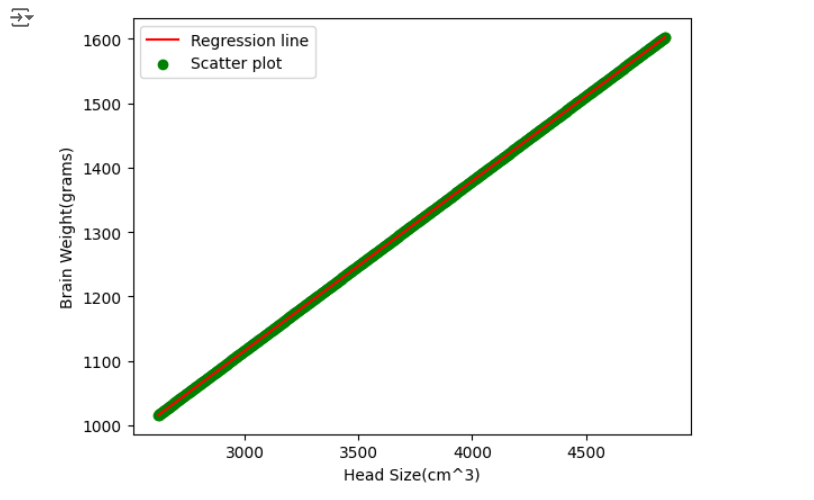
plt.xlabel('Head Size(cm^3)')

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)

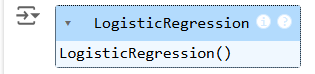
**Step 4 : Train a Model**

**# 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}")

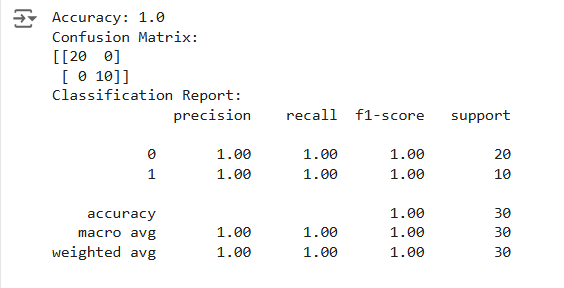
print("Confusion Matrix:")

print(conf\_matrix)

print("Classification Report:")

print(class\_report)

**OUTPUT :**



**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()

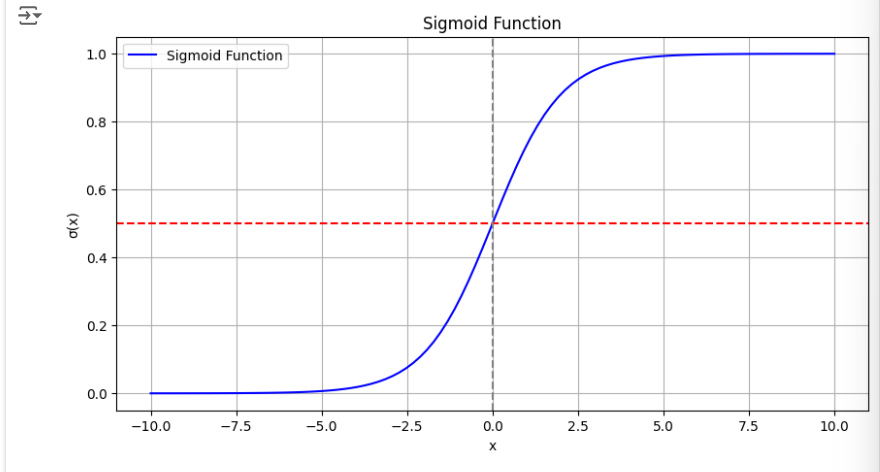
plt.axhline(0.5, color='red', linestyle='--') # Line at y=0.5

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.

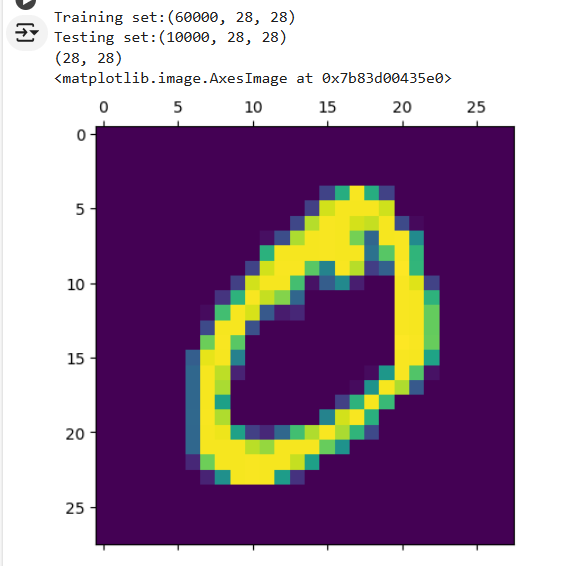
print(f"Training set:{X\_train.shape}")

print(f"Testing set:{X\_test.shape}")

print(X\_train[1].shape)

plt.matshow(X\_train[1])

**OUTPUT :**



**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',

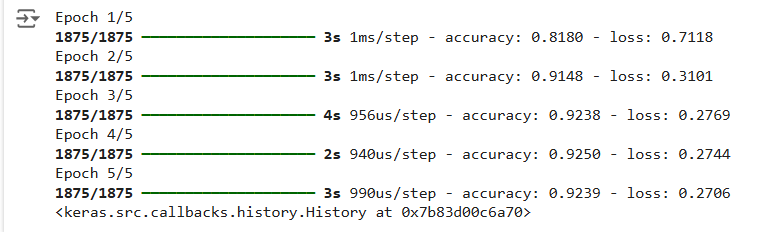
loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

model.fit(x\_train\_flatten,y\_train,epochs=5

)

**OUTPUT :**

****

**Step 6 : Evaluate the Model**

Evaluate the model performance.

model.evaluate(x\_test\_flatten,y\_test)

**OUTPUT :**



**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 :**



# **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

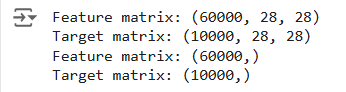
print("Feature matrix:", x\_train.shape)

print("Target matrix:", x\_test.shape)

print("Feature matrix:", y\_train.shape)

print("Target matrix:", y\_test.shape)

**OUTPUT :**



**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**

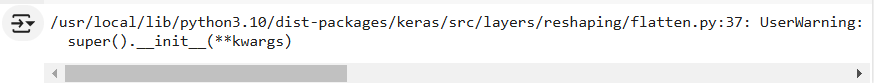
**Dense(128, activation='sigmoid'),**

**# output layer**

**Dense(10, activation='sigmoid'),**

**])**

**OUTPUT:**

****

**Step 5 : Make Predictions**

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

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

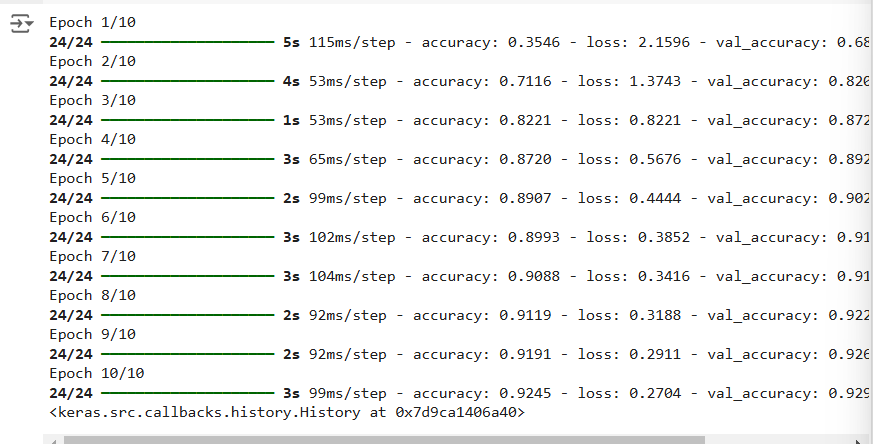
metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=10,

batch\_size=2000,

validation\_split=0.2)

**OUTPUT:**



**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),

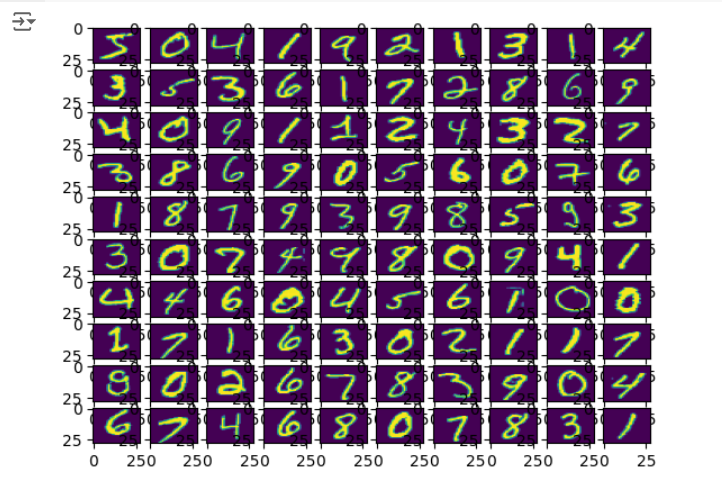
aspect='auto')

k += 1

plt.show()

**OUTPUT :**





**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 :**

# 

# **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)**

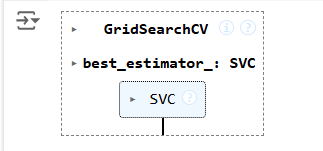
**# Creating a model using GridSearchCV with the parameters grid**

**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)**

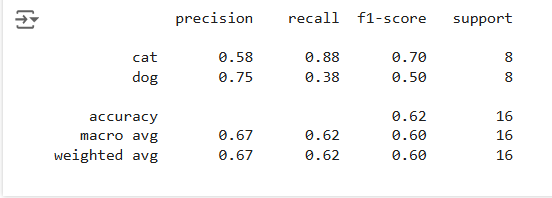
**# Print the accuracy of the model**

**print(f"The model is {accuracy\*100}% accurate")**

**print(classification\_report(y\_test, y\_pred, target\_names=['cat', 'dog']))**

**OUTPUT :**

****

****

**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**

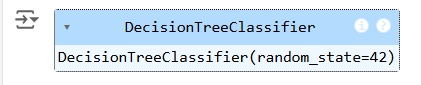
**# Create a Decision Tree classifier**

**clf = DecisionTreeClassifier(random\_state=42)**

**Step 5 : Train the model :   
# Train the model**

**clf.fit(X\_train, y\_train)**

**OUTPUT :**

****

**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}")**

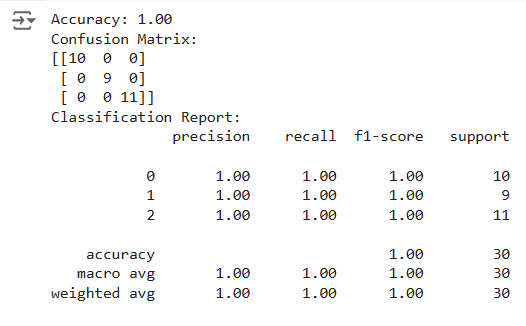
**print("Confusion Matrix:")**

**print(confusion)**

**print("Classification Report:")**

**print(classification\_report)**

**OUTPUT :**

****

**Step 7 : Visualize the decision tree**

**# Visualize the Decision Tree**

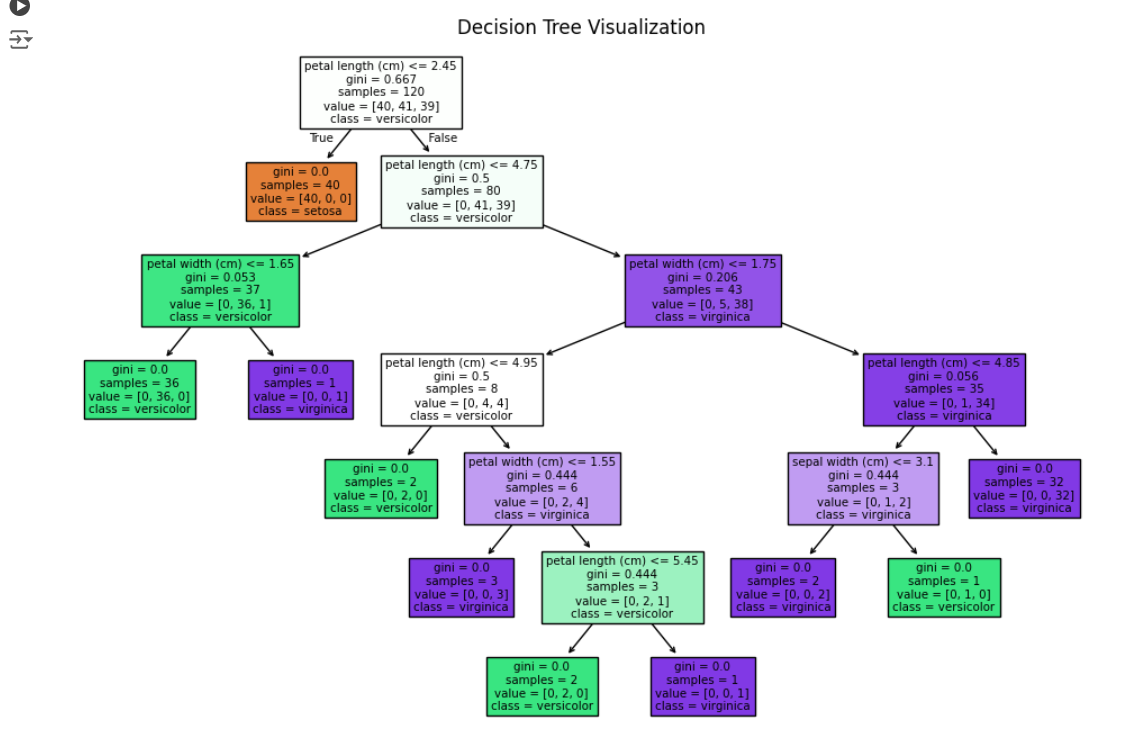
**plt.figure(figsize=(12,8))**

**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]**

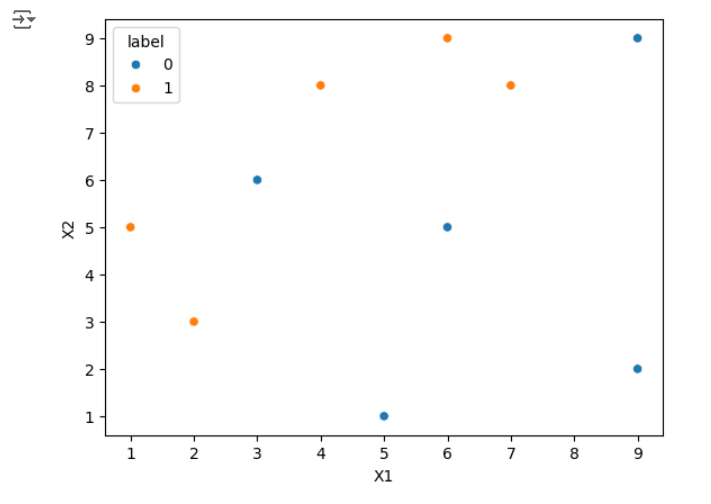
**sns.scatterplot(x=df['X1'], y=df['X2'], hue=df['label'])**

**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**

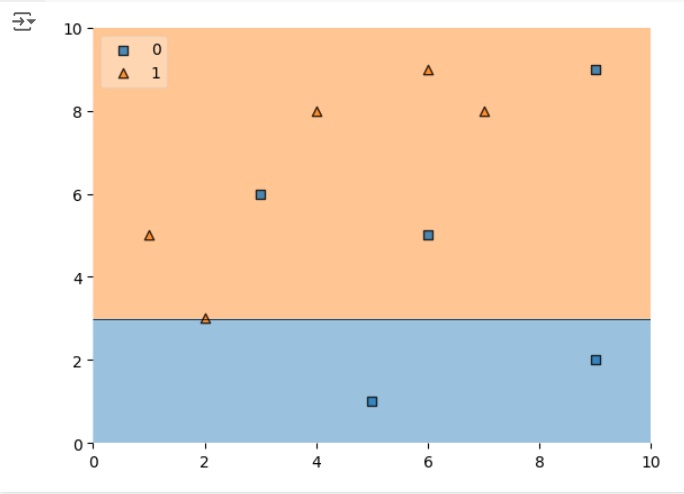
**dt1 = DecisionTreeClassifier(max\_depth=1)**

**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**

**# 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**

**# Step 6: Train 2nd Model**

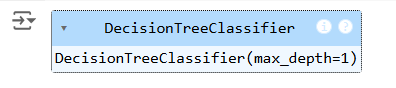
**dt2 = DecisionTreeClassifier(max\_depth=1)**

**x = second\_df.iloc[:, 0:2].values**

**y = second\_df.iloc[:, 2].values**

**dt2.fit(x, y)**

**OUTPUT :**

****

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

**# Plot the decision tree for the second model**

**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**

**# 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**

**# 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())**

**# 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)**

**# 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**

**# 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)**

**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')**

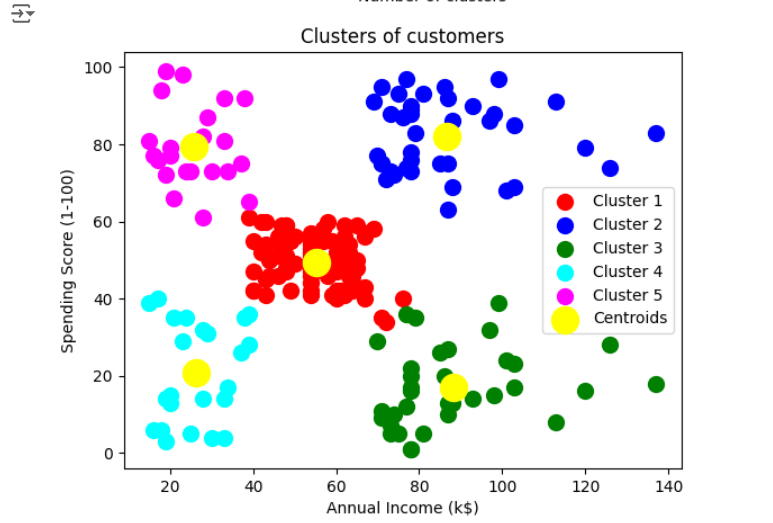
**plt.xlabel('Annual Income (k$)')**

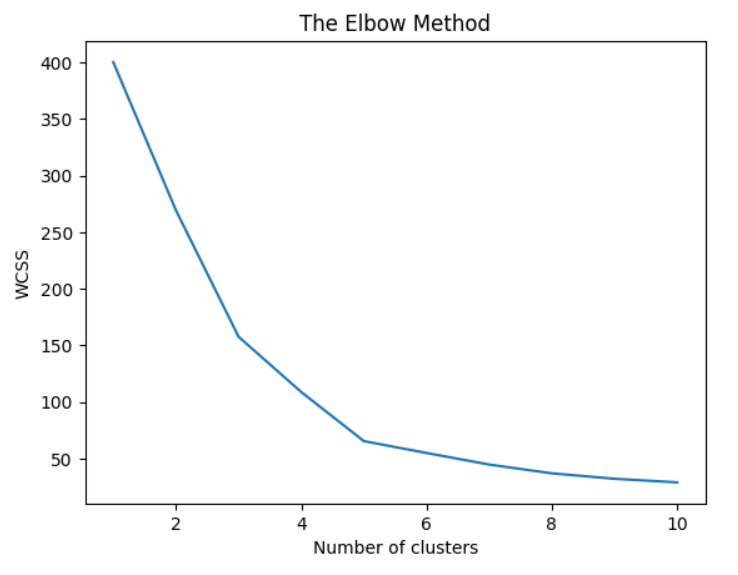
**plt.ylabel('Spending Score (1-100)')**

**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

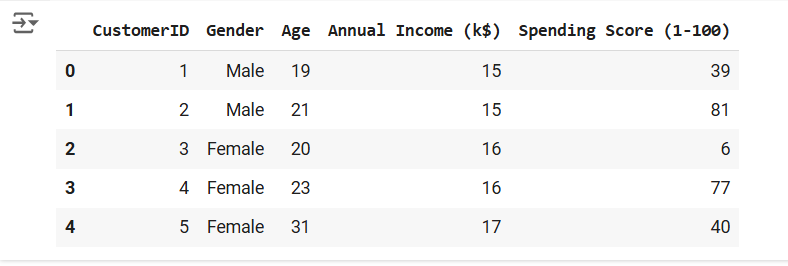
from math import sqrt

**Step 2 : load the Dataset**

**data = pd.read\_csv('/content/Mall\_Customers.csv')**

**data.head(5)**

**OUTPUT:**

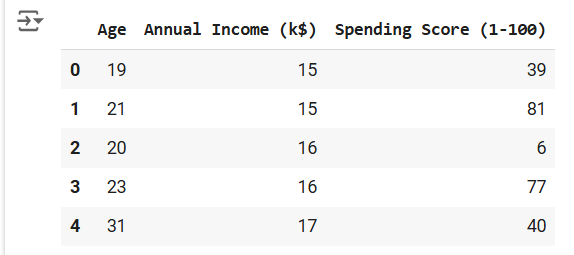
****

**Step 3 : Preprocess the data**

**req\_data = data[['Age', 'Annual Income (k$)', 'Spending Score (1-100)']]**

**req\_data.head(5)**

**OUTPUT :**

****

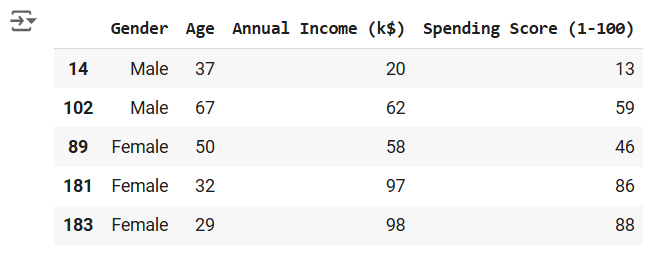
**Step 4 : Assign the data points to clusters**

**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 :**

****

**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 :**

****

**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

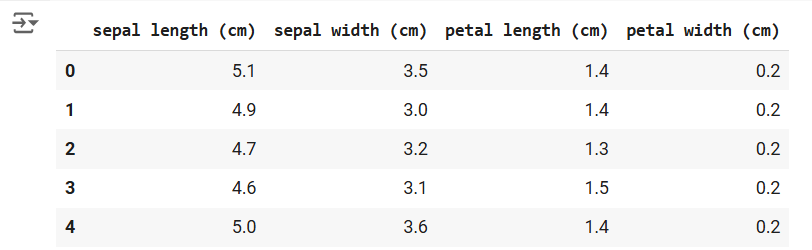
iris = datasets.load\_iris()

df = pd.DataFrame(iris['data'], columns=iris['feature\_names'])

# Display the first few rows of the dataset

df.head()

**OUTPUT :**

****

**Step 3 : Standardize the data**

**# Standardize the features using StandardScaler**

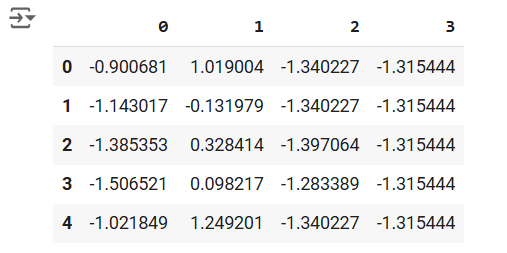
**scalar = StandardScaler()**

**scaled\_data = pd.DataFrame(scalar.fit\_transform(df)) # Scaling the data**

**# Display the scaled data (optional)**

**scaled\_data.head()**

**OUTPUT :**

****

**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**

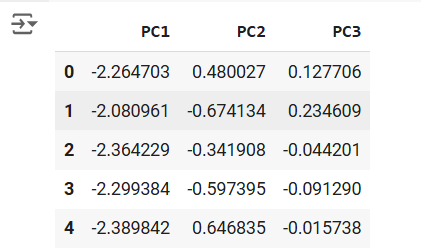
**data\_pca = pca.transform(scaled\_data) # Transform the data to principal components**

**# Convert PCA data to a DataFrame for easier inspection**

**data\_pca = pd.DataFrame(data\_pca, columns=['PC1', 'PC2', 'PC3'])**

**data\_pca.head()**

**OUTPUT :**

****

**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 :**

****

**Step 6 :Visualize the reduced data.**

**# 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')**

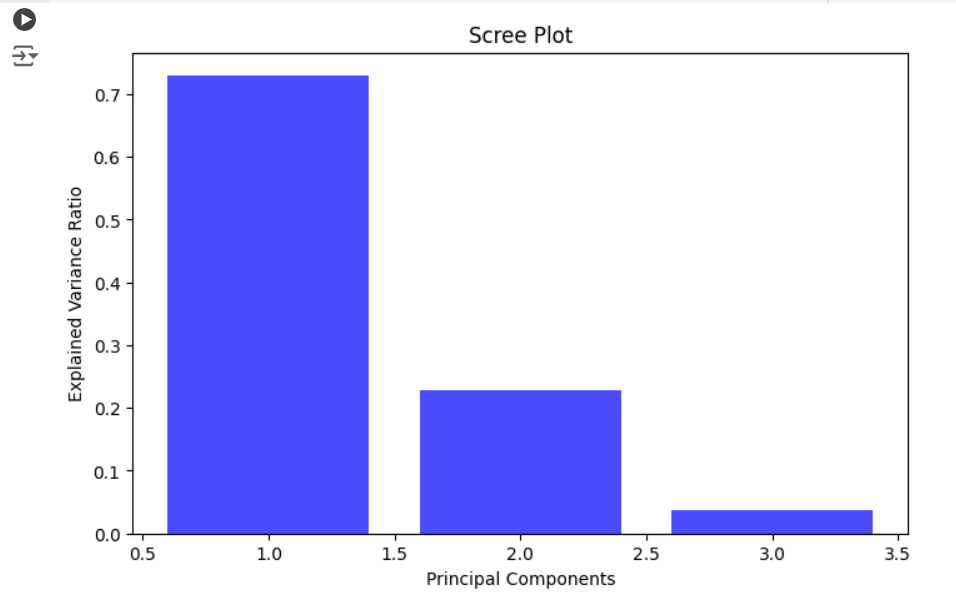
**plt.ylabel('Explained Variance Ratio')**

**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.