**RAJALAKSHMI ENGINEERING COLLEGE**

**RAJALAKSHMI NAGAR, THANDALAM – 602 105**

A logo for a college

Description automatically generated

| **CS23331**  **DESIGN AND ANALYSIS OF ALGORITHM LAB** |
| --- |
| **Laboratory Observation Note Book** |

A white paper with black dots and blue lines

Description automatically generated with medium confidence

| **S.No.** | **Date** | **Title** | **Page**  **No.** | **Teacher’s Signature /**  **Remarks** |
| --- | --- | --- | --- | --- |
| 1. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT UNIVARIATE, BIVARIATE AND MULTIVARIATE REGRESSION** |  |  |
| 2. | / /24 | A PYTHON PROGRAM TO IMPLEMENT SIMPLE LINEAR REGRESSION USING LEAST SQUARE METHOD |  |  |
| 3. | / /24 | A PYTHON PROGRAM TO IMPLEMENT LOGISTIC MODEL |  |  |
| 4. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT SINGLE LAYER PERCEPTRON** |  |  |
| 5. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT MULTI LAYER PERCEPTRON WITH BACK PROPOGATION** |  |  |
| 6. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT SVM CLASSIFIER MODEL** |  |  |
| 7. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT DECISION TREE** |  |  |
| 8. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT ADA BOOSTING** |  |  |
| 9. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT KNN MODEL** |  |  |
| 10. | / /24 | **A PYTHON PROGRAM TO IMPLEMENT DIMENSIONALITY REDUCTION USING**  **PCA** |  |  |

Ex No: 1

Date:

**A PYTHON PROGRAM TO IMPLEMENT UNIVARIATE, BIVARIATE AND**

**MULTIVARIATE REGRESION**

**Aim:**

To implement a python program using univariate, bivariate and multivariate regression features for a given iris dataset.

**Algorithm:**

Step 1: Import necessary libraries:

* pandas for data manipulation, numpy for numerical operations, and matplotlib.pyplot for plotting.

Step 2: Read the dataset:

* Use the pandas `read\_csv` function to read the dataset.
* Store the dataset in a variable (e.g., `data`).

Step 3: Prepare the data:

* Extract the independent variable(s) (X) and dependent variable (y) from the dataset.
* Reshape X and y to be 2D arrays if needed.

Step 4:Univariate Regression:

* For univariate regression, use only one independent variable.
* Fit a linear regression model to the data using numpy’s polyfit function or sklearn’s LinearRegression class.
* Make predictions using the model.
* Calculate the R-squared value to evaluate the model’s performance.

Step 5: Bivariate Regression:

* For bivariate regression, use two independent variables.
* Fit a linear regression model to the data using numpy’s `polyfit` function or sklearn’s `LinearRegression` class.
* Make predictions using the model.
* Calculate the R-squared value to evaluate the model’s performance.

Step 6: Multivariate Regression:

* For multivariate regression, use more than two independent variables.
* Fit a linear regression model to the data using sklearn’s `LinearRegression` class.
* Make predictions using the model.
* Calculate the R-squared value to evaluate the model’s performance.

Step 7: Plot the results:

* For univariate regression, plot the original data points (X, y) as a scatter plot and the regression line as a line plot.
* For bivariate regression, plot the original data points (X1, X2, y) as a 3D scatter plot and the regression plane.
* For multivariate regression, plot the predicted values against the actual values.

Step 8: Display the results:

* Print the coefficients (slope) and intercept for each regression model.
* Print the R-squared value for each regression model.

Step 9: Complete the program:

* Combine all the steps into a Python program.
* Run the program to perform univariate, bivariate, and multivariate regression on the dataset.

**Code:**

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import numpy as np

df=pd.read\_csv('/content/drive/MyDrive/Datasets/iris.csv')

df.head(150)

df.shape

df

df\_Setosa=df.loc[df['species']=='setosa']

df\_Virginica=df.loc[df['species']=='virginica']

df\_Versicolor=df.loc[df['species']=='versicolor']

df\_Setosa

#univariate for sepal width

plt.scatter(df\_Setosa['sepal\_width'],np.zeros\_like(df\_Setosa['sepal\_width']))

plt.scatter(df\_Virginica['sepal\_width'],np.zeros\_like(df\_Virginica['sepal\_width']))

plt.scatter(df\_Versicolor['sepal\_width'],np.zeros\_like(df\_Versicolor['sepal\_width']))

plt.xlabel('sepal\_width')

plt.show()

#univariate for sepal length

plt.scatter(df\_Setosa['sepal\_length'],np.zeros\_like(df\_Setosa['sepal\_length']))

plt.scatter(df\_Virginica['sepal\_length'],np.zeros\_like(df\_Virginica['sepal\_length']))

plt.scatter(df\_Versicolor['sepal\_length'],np.zeros\_like(df\_Versicolor['sepal\_length']))

plt.xlabel('sepal\_length')

plt.show()

#univariate for sepal width

plt.scatter(df\_Setosa['petal\_width'],np.zeros\_like(df\_Setosa['petal\_width']))

plt.scatter(df\_Virginica['petal\_width'],np.zeros\_like(df\_Virginica['petal\_width']))

plt.scatter(df\_Versicolor['petal\_width'],np.zeros\_like(df\_Versicolor['petal\_width']))

plt.xlabel('petal\_width')

plt.show()

#univariate for sepal length

plt.scatter(df\_Setosa['petal\_length'],np.zeros\_like(df\_Setosa['petal\_length']))

plt.scatter(df\_Virginica['petal\_length'],np.zeros\_like(df\_Virginica['petal\_length']))

plt.scatter(df\_Versicolor['petal\_length'],np.zeros\_like(df\_Versicolor['petal\_length']))

plt.xlabel('petal\_length')

plt.show()

#bivariate sepal.width vs petal.width

sns.FacetGrid(df,hue='species',height=5).map(plt.scatter,"sepal\_width","petal\_width").add\_legend();

plt.show()

#bivariate sepal.length vs petal.length

sns.FacetGrid(df,hue='species',height=5).map(plt.scatter,"sepal\_length","petal\_length").add\_legend();

plt.show()

#multivariate all the features

sns.pairplot(df,hue='species',size=2)

**Output:**

A table of numbers with text

Description automatically generated with medium confidence

A table of numbers with text

Description automatically generated with medium confidence

A table with numbers and letters

Description automatically generated

A table of numbers with black text

Description automatically generated

A table with numbers on it

Description automatically generated

A graph with dots and numbers

Description automatically generated

A line of dots with different colors

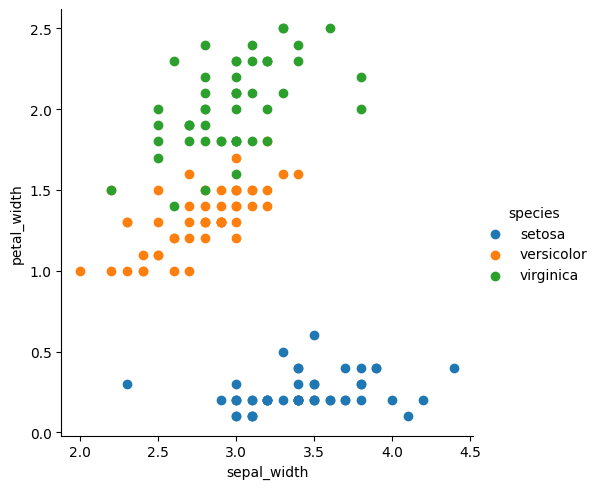
Description automatically generated

A graph with colored dots

Description automatically generated with medium confidence

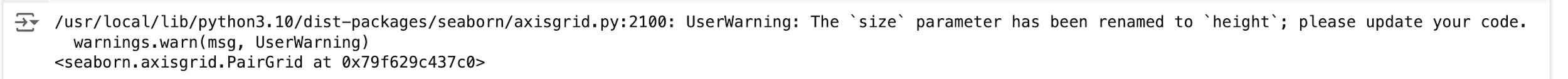
A graph with a number of dots

Description automatically generated with medium confidence



A diagram of different colored dots

Description automatically generated



A group of graphs with different colored dots

Description automatically generated with medium confidence

**Result:**

Thus, the python program to implement univariate, bivariate and multivariate has been successfully implemented and the results have been verified and analysed.

**Ex No: 2**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT SIMPLE LINEAR REGRESSION USING LEAST SQUARE METHOD**

**Aim***:*

To implement a python program for constructing a simple linear regression using least square method.

**Algorithm:**

Step 1: Import necessary libraries:

* pandas for data manipulation and matplotlib.pyplot for plotting.

Step 2: Read the dataset:

* Use the pandas `read\_csv` function to read the dataset (e.g., headbrain.csv).
* Store the dataset in a variable (e.g., `data`).

Step 3: Prepare the data:

* Extract the independent variable (X) and dependent variable (y) from the dataset.
* Reshape X and y to be 2D arrays if needed.

Step 4: Calculate the mean:

* Calculate the mean of X and y.

Step 5: Calculate the coefficients:

* Calculate the slope (m) using the formula:

A black and white math equations

Description automatically generated

* Calculate the intercept (b) using the formula: 

Step 6: Make predictions:

* Use the calculated slope and intercept to make predictions for each X value:



Step 7: Plot the regression line:

* Plot the original data points (X, y) as a scatter plot.
* Plot the regression line (X, predicted\_y) as a line plot.

Step 8: Calculate the R-squared value:

* Calculate the total sum of squares (TSS) using the formula: 
* Calculate the residual sum of squares (RSS) using the formula: 
* Calculate the R-squared value using the formula: 

Step 9: Display the results:

* Print the slope, intercept, and R-squared value.

Step 10: Complete the program:

* Combine all the steps into a Python program.
* Run the program to perform simple linear regression on the dataset.

**Code:**

from google.colab import drive

drive.mount('/content/drive')

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

data = pd.read\_csv('/content/drive/MyDrive/Datasets/headbrain.csv')

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

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

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)

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()

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()

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))

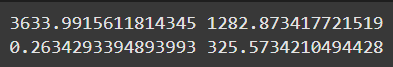
reg=LinearRegression()

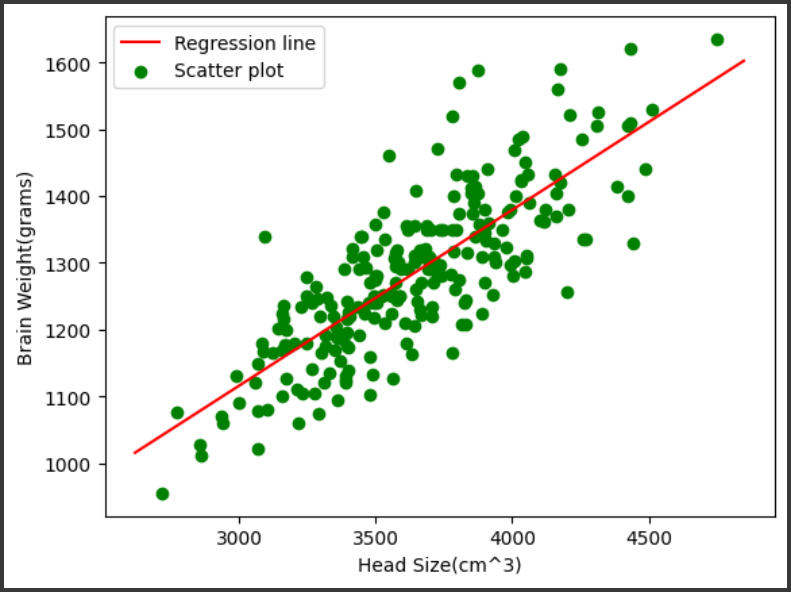
reg=reg.fit(x, y)

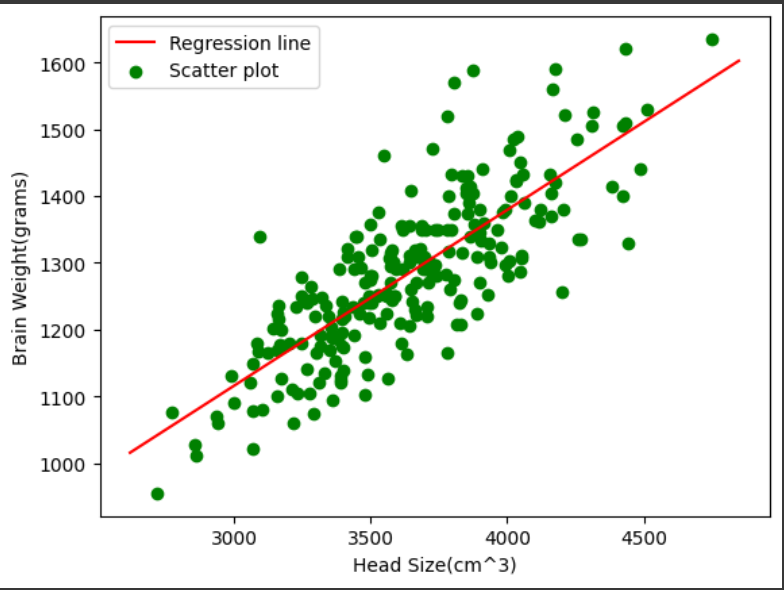
print(reg.score(x, y))

**Output:**

****

****

****

****

****

****

**Result:**

Thus, the python program to Simple Linear Regression using Least Square Method has been successfully implemented and the results have been verified and analysed.

**Ex no: 3**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT LOGISTIC MODEL**

**Aim:**

To implement python program for the logistic model using suv car dataset.

**Algorithm:**

Step 1: Import Necessary Libraries:

* pandas for data manipulation
* sklearn.model\_selection for train-test split
* sklearn.preprocessing for data preprocessing
* sklearn.linear\_model for logistic regression
* matplotlib.pyplot for plotting

Step 2: Read the Dataset:

* Use pandas to read the suv\_cars.csv dataset into a DataFrame.

Step 3: Preprocess the Data:

* Select the relevant columns for the analysis (e.g., 'Age', 'EstimatedSalary', 'Purchased').
* Encode categorical variables if necessary (e.g., using LabelEncoder or OneHotEncoder).
* Split the data into features (X) and target variable (y).

Step 4: Split the Data:

* Split the dataset into training and testing sets using train\_test\_split.

Step 5: Feature Scaling:

* Standardize the features using StandardScaler to ensure they have the same scale.

Step 6: Create and Train the Model:

* Create a logistic regression model using LogisticRegression from sklearn.linear\_model.
* Train the model on the training data using the fit method.
  + Create a function named “Sigmoid ()” which will define the sigmoid values using the
  + formula (1/1+e-z) and return the computed value.
  + Create a function named “initialize()” which will initialize the values with zeroes and assign the value to “weights” variable, initializes with ones and assigns the value to variable “x” and returns both “x” and “weights”.
  + Create a function named “fit” which will be used to plot the graph according to the training data.
  + Create a predict function that will predict values according to the training model created using the fit function.
  + Invoke the standardize() function for “x-train” and “x-test”

Step 7: Make Predictions:

* Use the trained model to make predictions on the test data using the predict method.
  + Use the “predict()” function to predict the values of the testing data and assign the value to “y\_pred” variable.
  + Use the “predict()” function to predict the values of the training data and assign the value to “y\_trainn” variable.
  + Compute f1\_score for both the training and testing data and assign the values to “f1\_score\_tr” and “f1\_score\_te” respectively

Step 8: Evaluate the Model:

* Calculate the accuracy of the model on the test data using the score method.

  (Accuracy = (tp+tn)/(tp+tn+fp+fn)).

* Generate a confusion matrix and classification report to further evaluate the model's performance.

Step 9: Visualize the Results:

* Plot the decision boundary of the logistic regression model (optional).

**Code:**

import pandas as pd

import numpy as np

from numpy import log,dot,exp,shape

from sklearn.metrics import confusion\_matrix

data = pd.read\_csv('/content/drive/MyDrive/suv\_data.csv')

print(data.head())

x = data.iloc[:, [2, 3]].values

y = data.iloc[:, 4].values

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test=train\_test\_split(x,y,test\_size=0.10, random\_state=0)

from sklearn.preprocessing import StandardScaler

sc=StandardScaler()

x\_train=sc.fit\_transform(x\_train)

x\_test=sc.transform(x\_test)

print (x\_train[0:10,:])

from sklearn.linear\_model import LogisticRegression

classifier=LogisticRegression(random\_state=0)

classifier.fit(x\_train,y\_train)

LogisticRegression (random\_state=0)

y\_pred = classifier.predict(x\_test)

print(y\_pred)

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

print ("Confusion Matrix : \n", cm)

from sklearn.metrics import accuracy\_score

print ("Accuracy : ", accuracy\_score(y\_test, y\_pred))

from sklearn.model\_selection import train\_test\_split

x\_train, x\_test, y\_train, y\_test=train\_test\_split(x,y,test\_size=0.10, random\_state=0)

def Std(input\_data):

mean0 = np.mean(input\_data[:, 0])

sd0 = np.std(input\_data[:, 0])

mean1 = np.mean(input\_data[:, 1])

sd1 = np.std(input\_data[:, 1])

return lambda x:((x[0]-mean0)/sd0, (x[1]-mean1)/sd1)

my\_std = Std(x)

my\_std(x\_train[0])

def standardize(X\_tr):

for i in range(shape(X\_tr)[1]):

X\_tr[:,i] = (X\_tr[:,i] - np.mean(X\_tr[:,i]))/np.std(X\_tr[:,i])

def F1\_score(y,y\_hat):

tp,tn,fp,fn = 0,0,0,0

for i in range(len(y)):

if y[i] == 1 and y\_hat[i] == 1:

tp += 1

elif y[i] == 1 and y\_hat[i] == 0:

fn += 1

elif y[i] == 0 and y\_hat[i] == 1:

fp += 1

elif y[i] == 0 and y\_hat[i] == 0:

tn += 1

precision = tp/(tp+fp)

recall = tp/(tp+fn)

f1\_score = 2\*precision\*recall/(precision+recall)

return f1\_score

class LogisticRegression:

def sigmoid(self, z):

sig = 1 / (1 + exp(-z))

return sig

def initialize(self, X):

weights = np.zeros((shape(X)[1] + 1, 1))

X = np.c\_[np.ones((shape(X)[0], 1)), X]

return weights, X

def fit(self, X, y, alpha=0.001, iter=400):

weights, X = self.initialize(X)

def cost(theta):

z = dot(X, theta)

cost0 = y.T.dot(log(self.sigmoid(z)))

cost1 = (1 - y).T.dot(log(1 - self.sigmoid(z)))

cost = -((cost1 + cost0)) / len(y)

return cost

cost\_list = np.zeros(iter,)

for i in range(iter):

weights = weights - alpha \* dot(X.T, self.sigmoid(dot(X, weights)) - np.reshape(y, (len(y), 1)))

cost\_list[i] = cost(weights).item()

self.weights = weights

return cost\_list

def predict(self, X):

z = dot(self.initialize(X)[1], self.weights)

lis = []

for i in self.sigmoid(z):

if i > 0.5:

lis.append(1)

else:

lis.append(0)

return lis

standardize(x\_train)

standardize(x\_test)

obj1 = LogisticRegression()

model = obj1.fit(x\_train, y\_train)

y\_pred = obj1.predict(x\_test)

y\_trainn = obj1.predict(x\_train)

f1\_score\_tr = F1\_score(y\_train, y\_trainn)

f1\_score\_te = F1\_score(y\_test, y\_pred)

print(f1\_score\_tr)

print(f1\_score\_te)

conf\_mat = confusion\_matrix(y\_test, y\_pred)

accuracy = (conf\_mat[0, 0] + conf\_mat[1, 1]) / sum(sum(conf\_mat))

print("Accuracy is : ", accuracy)

**Output:**

A number of numbers on a white background

Description automatically generated

[[-1.05714987 0.53420426]

[ 0.2798728 -0.51764734]

[-1.05714987 0.41733186]

[-0.29313691 -1.45262654]

[ 0.47087604 1.23543867]

[-1.05714987 -0.34233874]

[-0.10213368 0.30045946]

[ 1.33039061 0.59264046]

[-1.15265148 -1.16044554]

[ 1.04388575 0.47576806]]

[0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0

0 0 1]

Confusion Matrix :

[[31 1]

[ 1 7]]

Accuracy : 0.95

(-1.017692393473028, 0.5361288690822568)

0.7583333333333334

0.823529411764706

Accuracy is : 0.925

**Result:**

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

**Ex. No.: 4**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT SINGLE LAYER PERCEPTRON**

**Aim:**

To implement python program for the single layer perceptron.

**Algorithm:**

Step 1: Import Necessary Libraries:

* Import numpy for numerical operations.

Step 2: Initialize the Perceptron:

* Define the number of input features (input\_dim).
* Initialize weights (W) and bias (b) to zero or small random values.

Step 3: Define Activation Function:

* Choose an activation function (e.g., step function, sigmoid, or ReLU).
* User Defined function - sigmoid\_func(x):
  + Compute 1/(1+np.exp(-x)) and return the value.
* User Defined function - der(x):
  + Compute the product of value of sigmoid\_func(x) and (1 - sigmoid\_func(x) ) and return the value.

Step 4; Define Training Data:

* Define input features (X) and corresponding target labels (y).

Step 5: Define Learning Rate and Number of Epochs:

* Choose a learning rate (alpha) and the number of training epochs.

Step 6: Training the Perceptron:

* For each epoch:
  + For each input sample in the training data:
  + Compute the weighted sum of inputs (z) as the dot product of input features and weights plus bias (z = np.dot(X[i], W) + b).
  + Apply the activation function to get the predicted output (y\_pred).
  + Compute the error (error = y[i] - y\_pred).
  + Update the weights and bias using the learning rate and error (W += alpha \* error \* X[i]; b += alpha \* error).

Step 7: Prediction:

* Use the trained perceptron to predict the output for new input data.

Step 8: Evaluate the Model:

* Measure the performance of the model using metrics such as accuracy, precision, recall, etc.

**Code:**

import numpy as np

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

input\_dim=2

W=np.zeros(input\_dim)

b=0.0

def sigmoid\_func(x):

return 1 / (1 + np.exp(-x))

def der(x):

sigmoid = sigmoid\_func(x)

return sigmoid \* (1 - sigmoid)

np.random.seed(42)

x = np.array([[150,8],

[130,7],

[180,6],

[170,5]])

y = np.array([0,0,1,1])

alpha = 0.1

epochs = 10000

for epoch in range(epochs):

for i in range(len(x)):

z = np.dot(x[i], W) + b

y\_pred = sigmoid\_func(z)

error = y[i] - y\_pred

W += alpha \* error \* x[i]

b += alpha \* error

def predict(X):

z = np.dot(X, W) + b

return (sigmoid\_func(z) > 0.5).astype(int)

y\_pred = predict(x)

accuracy = accuracy\_score(y, y\_pred)

precision = precision\_score(y, y\_pred)

recall = recall\_score(y, y\_pred)

F1\_score = f1\_score(y, y\_pred)

print("Prediction:",y\_pred)

print("Accuracy:", accuracy)

print("Precision:", precision)

print("Recall:", recall)

print("F1 Score:", F1\_score)

**Output:**

Prediction: [0 0 1 1]

Accuracy: 1.0

Precision: 1.0

Recall: 1.0

F1 Score: 1.0

**Result:**

Thus, the python program to implement single layer perceptron has been successfully implemented and the results have been verified and analysed.

**Ex. No.: 5**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT MULTI LAYER PERCEPTRON WITH** **BACK PROPOGATION**

**Aim:**

To implement multilayer perceptron with back propagation using python.

**Algorithm:**

Step 1: Import the Necessary Libraries

* Import pandas as pd.
* Import numpy as np.

Step 2: Read and Display the Dataset

* Use `pd.read\_csv("banknotes.csv")` to read the dataset.
* Assign the result to a variable (e.g., `data`).
* Display the first ten rows using `data.head(10)`.

Step 3: Display Dataset Dimensions

* Use the `.shape` attribute on the dataset (e.g., `data.shape`).

Step 4: Display Descriptive Statistics

* Use the `.describe()` function on the dataset (e.g., `data.describe()`).

Step 5: Import Train-Test Split Module

* Import `train\_test\_split` from `sklearn.model\_selection`.

Step 6: Split Dataset with 80-20 Ratio

* Assign the features to a variable (e.g., `X = data.drop(columns='target')`).
* Assign the target variable to another variable (e.g., `y = data['target']`).
* Use `train\_test\_split` to split the dataset into training and testing sets with a ratio of 0.2.
* Assign the results to `x\_train`, `x\_test`, `y\_train`, and `y\_test`.

Step 7: Import MLPClassifier Module

* Import `MLPClassifier` from `sklearn.neural\_network`.

Step 8: Initialize MLPClassifier

* Create an instance of `MLPClassifier` with `max\_iter=500` and `activation='relu'`.
* Assign the instance to a variable (e.g., `clf`).

Step 9: Fit the Classifier

* Fit the model using `clf.fit(x\_train, y\_train)`.

Step 10: Make Predictions

* Use the `.predict()` function on `x\_test` (e.g., `pred = clf.predict(x\_test)`).
* Display the predictions.

Step 11: Import Metrics Modules

* Import `confusion\_matrix` from `sklearn.metrics`.
* Import `classification\_report` from `sklearn.metrics`.

Step 12: Display Confusion Matrix

* Use `confusion\_matrix(y\_test, pred)` to generate the confusion matrix.
* Display the confusion matrix.

Step 13: Display Classification Report

* Use `classification\_report(y\_test, pred)` to generate the classification report.
* Display the classification report.

Step 14: Repeat Steps 9-13 with Different Activation Functions

* Initialize `MLPClassifier` with `activation='logistic'`.
* Fit the model and make predictions.
* Display the confusion matrix and classification report.
* Repeat for `activation='tanh'`.
* Repeat for `activation='identity'`.

Step 15: Repeat Steps 7-14 with 70-30 Ratio

* Use `train\_test\_split` to split the dataset into training and testing sets with a ratio of 0.3.
* Assign the results to `x\_train`, `x\_test`, `y\_train`, and `y\_test`.
* Repeat Steps 7-14 with the new training and testing sets.

**Code:**

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.neural\_network import MLPClassifier

from sklearn.metrics import classification\_report, confusion\_matrix

bnotes = pd.read\_csv('../content/drive/MyDrive/bank\_note\_data.csv')

print(bnotes.head(10))

x = bnotes.drop('Class', axis=1)

y = bnotes['Class']

print(x.head(2))

print(y.head(2))

def train\_and\_evaluate(activation, x\_train, y\_train, x\_test, y\_test):

mlp = MLPClassifier(max\_iter=500, activation=activation)

mlp.fit(x\_train, y\_train)

pred = mlp.predict(x\_test)

print(f"Predictions using activation function '{activation}':\n{pred}\n")

cm = confusion\_matrix(y\_test, pred)

print(f"Confusion Matrix for '{activation}':\n{cm}\n")

report = classification\_report(y\_test, pred)

print(f"Classification Report for '{activation}':\n{report}\n")

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2)

for activation in ['relu', 'logistic', 'tanh', 'identity']:

train\_and\_evaluate(activation, x\_train, y\_train, x\_test, y\_test)

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.3)

for activation in ['relu', 'logistic', 'tanh', 'identity']:

train\_and\_evaluate(activation, x\_train, y\_train, x\_test, y\_test)

**Output:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Result:**

Thus, the python program to implement multi-layer perceptron has been successfully implemented and the results have been verified and analysed.

**Ex no: 6**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT SVM CLASSIFIER MODEL**

**Aim:**

To implement a SVM classifier model using python and determine its accuracy.

**Algorithm:**

Step 1: Import Necessary Libraries

* + Import numpy as np.
  + Import pandas as pd.
  + Import SVM from sklearn.
  + Import matplotlib.pyplot as plt.
  + Import seaborn as sns.
  + Set the font\_scale attribute to 1.2 in seaborn.

Step 2: Load and Display Dataset

* + Read the dataset (muffins.csv) using `pd.read\_csv()`.
  + Display the first five instances using the `head()` function.

Step 3: Plot Initial Data

* + Use the `sns.lmplot()` function.
  + Set the x and y axes to "Sugar" and "Flour".
  + Assign "recipes" to the data parameter.
  + Assign "Type" to the hue parameter.
  + Set the palette to "Set1".
  + Set fit\_reg to False.
  + Set scatter\_kws to {"s": 70}.
  + Plot the graph.

Step 4: Prepare Data for SVM

* + Extract "Sugar" and "Butter" columns from the recipes dataset and assign to variable `sugar\_butter`.
  + Create a new variable `type\_label`.
  + For each value in the "Type" column, assign 0 if it is "Muffin" and 1 otherwise.

Step 5: Train SVM Model

* + Import the SVC module from the svm library.
  + Create an SVC model with kernel type set to linear.
  + Fit the model using `sugar\_butter` and `type\_label` as the parameters.

Step 6: Calculate Decision Boundary

* + Use the `model.coef\_` function to get the coefficients of the linear model.
  + Assign the coefficients to a list named `w`.
  + Calculate the slope `a` as `w[0] / w[1]`.
  + Use `np.linspace()` to generate values from 5 to 30 and assign to variable `xx`.
  + Calculate the intercept using the first value of the model intercept and divide by `w[1]`.
  + Calculate the decision boundary line `y` as `a \* xx - (model.intercept\_[0] / w[1])`.

Step 7: Calculate Support Vector Boundaries

* + Assign the first support vector to variable `b`.
  + Calculate `yy\_down` as `a \* xx + (b[1] - a \* b[0])`.
  + Assign the last support vector to variable `b`.
  + Calculate `yy\_up` using the same method.

Step 8: Plot Decision Boundary

* + Use the `sns.lmplot()` function again with the same parameters as in Step 3.
  + Plot the decision boundary line `xx` and `yy`.

Step 9: Plot Support Vector Boundaries

* + Plot the decision boundary with `xx`, `yy\_down`, and `'k--'`.
  + Plot the support vector boundaries with `xx`, `yy\_up`, and `'k--'`.
  + Scatter plot the first and last support vectors.

Step 10: Import Additional Libraries

* + Import `confusion\_matrix` from `sklearn.metrics`.
  + Import `classification\_report` from `sklearn.metrics`.
  + Import `train\_test\_split` from `sklearn.model\_selection`.

Step 11: Split Dataset

* + Assign `x\_train`, `x\_test`, `y\_train`, and `y\_test` using `train\_test\_split`.
  + Set the test size to 0.2.

Step 12: Train New Model

* + Create a new SVC model named `model1`.
  + Fit the model using the training data (`x\_train` and `y\_train`).

Step 13: Make Predictions

* + Use the `predict()` function on `model1` with `x\_test` as the parameter.
  + Assign the predictions to variable `pred`.

Step 14: Evaluate Model

* + Display the confusion matrix.
  + Display the classification report.

**Code:**

import numpy as np

import pandas as pd

from sklearn import svm

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.model\_selection import train\_test\_split

sns.set(font\_scale=1.2)

recipes = pd.read\_csv('recipes\_muffins\_cupcakes.csv')

print(recipes.head())

print(recipes.shape)

sns.lmplot(x='Sugar', y='Flour', data=recipes, hue='Type', palette='Set1', fit\_reg=False, scatter\_kws={"s": 70})

sugar\_butter = recipes[['Sugar', 'Flour']].values

type\_label = np.where(recipes['Type'] == 'Muffin', 0, 1)

model = svm.SVC(kernel='linear')

model.fit(sugar\_butter, type\_label)

w = model.coef\_[0]

a = -w[0] / w[1]

xx = np.linspace(5, 30)

yy = a \* xx - (model.intercept\_[0] / w[1])

b = model.support\_vectors\_[0]

yy\_down = a \* xx + (b[1] - a \* b[0])

b = model.support\_vectors\_[-1]

yy\_up = a \* xx + (b[1] - a \* b[0])

sns.lmplot(x='Sugar', y='Flour', data=recipes, hue='Type', palette='Set1', fit\_reg=False, scatter\_kws={"s": 70})

plt.plot(xx, yy, linewidth=2, color='black')

plt.plot(xx, yy\_down, 'k--')

plt.plot(xx, yy\_up, 'k--')

plt.scatter(model.support\_vectors\_[:, 0], model.support\_vectors\_[:, 1], s=80, facecolors='none')

x\_train, x\_test, y\_train, y\_test = train\_test\_split(sugar\_butter, type\_label, test\_size=0.2)

model1 = svm.SVC(kernel='linear')

model1.fit(x\_train, y\_train)

pred = model1.predict(x\_test)

print(pred)

print(confusion\_matrix(y\_test, pred))

print(classification\_report(y\_test, pred, zero\_division=1))

plt.show()

**Output:**

**A screenshot of a computer

Description automatically generated**

A graph of different types of sugar

Description automatically generated

A graph with red and blue dots

Description automatically generated

**Result:**

Thus, the python program to implement SVM classifier model has been successfully implemented and the results have been verified and analysed.

**Ex. No.: 7**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT DECISION TREE**

**Aim:**

To implement a decision tree using a python program for the given dataset and plot the trained decision tree.

**Algorithm:**

Step 1: Import the Iris Dataset

1. Import `load\_iris` from `sklearn.datasets`.

Step 2: Import Necessary Libraries

1. Import numpy as np.

2. Import matplotlib.pyplot as plt.

3. Import `DecisionTreeClassifier` from `sklearn.tree`.

Step 3: Declare and Initialize Parameters

1. Declare and initialize `n\_classes = 3`.

2. Declare and initialize `plot\_colors = "ryb"`.

3. Declare and initialize `plot\_step = 0.02`.

Step 4: Prepare Data for Model Training

1. Load the iris dataset using `load\_iris()`.

2. Assign the dataset's data to variable `X`.

3. Assign the dataset's target to variable `Y`.

Step 5: Train the Model

1. Create an instance of `DecisionTreeClassifier`.

2. Fit the classifier using `clf.fit(X, Y)`.

Step 6: Initialize Pair Index and Plot Graph

1. Loop through each pair of features using `for pairidx, pair in enumerate(combinations (range(X.shape[1]), 2)):`

2. Inside the loop, assign `X` with the selected pair of features (e.g., `X = iris.data[:, pair]`).

3. Assign `Y` with the target list (e.g., `Y = iris.target`).

Step 7: Assign Axis Limits

1. Inside the loop, assign `x\_min` with the minimum value of the selected feature minus 1 (e.g., `x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1`).

2. Assign `x\_max` with the maximum value of the selected feature plus 1.

3. Assign `y\_min` with the minimum value of the second selected feature minus 1 (e.g., `y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1`).

4. Assign `y\_max` with the maximum value of the second selected feature plus 1.

Step 8: Create Meshgrid

1. Use `np.meshgrid` to create a grid of values from `x\_min` to `x\_max` and `y\_min` to `y\_max` with steps of `plot\_step`.

2. Assign the results to variables `xx` and `yy`.

Step 9: Plot Graph with Tight Layout

1. Use `plt.tight\_layout()` to adjust the layout of the plots.

2. Set `h\_pad=0.5`, `w\_pad=0.5`, and `pad=2.5`.

Step 10: Predict and Reshape

1. Use the classifier to predict on the meshgrid (e.g., `Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])`).

2. Reshape `Z` to the shape of `xx`.

Step 11: Plot Decision Boundary

1. Use `plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)` to plot the decision boundary with the "RdYlBu" color scheme.

Step 12: Plot Feature Pairs

1. Inside the loop, label the x-axis and y-axis with the feature names (e.g., `plt.xlabel(iris.feature\_names[pair[0]])` and `plt.ylabel(iris.feature\_names[pair[1]])`).

Step 13: Plot Training Points

1. Use `plt.scatter(X[:, 0], X[:, 1], c=Y, cmap=plt.cm.RdYlBu, edgecolor='k', s=15)` to plot the training points with the "RdYlBu" color scheme, black edge color, and size 15.

Step 14: Plot Final Decision Tree

1. Set the title of the plot to "Decision tree trained on all the iris features" (e.g., `plt.title("Decision tree trained on all the iris features")`).

2. Display the plot using `plt.show()`.

**Code:**

from sklearn.datasets import load\_iris

iris = load\_iris()

import numpy as np

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier

# Parameters

n\_classes = 3

plot\_colors = "ryb"

plot\_step = 0.02

for pairidx, pair in enumerate([[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]):

  # We only take the two corresponding features

  X = iris.data[:, pair]

  y = iris.target

# Train

clf = DecisionTreeClassifier().fit(X, y)

# Plot the decision boundary

plt.subplot(2, 3, pairidx + 1)

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, plot\_step), np.arange(y\_min, y\_max, plot\_step))

plt.tight\_layout(h\_pad=0.5, w\_pad=0.5, pad=2.5)

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

cs = plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu)

plt.xlabel(iris.feature\_names[pair[0]])

plt.ylabel(iris.feature\_names[pair[1]])

# Plot the training points

for i, color in zip(range(n\_classes), plot\_colors):

  idx = np.where(y == i)

  plt.scatter(X[idx, 0],X[idx, 1],c=color,label=iris.target\_names[i],cmap=plt.cm.RdYlBu,edgecolor="black",s=15)

plt.suptitle("Decision surface of decision trees trained on pairs of features")

plt.legend(loc="lower right", borderpad=0, handletextpad=0)

plt.axis("tight")

plt.show()

from sklearn.tree import plot\_tree

plt.figure()

clf = DecisionTreeClassifier().fit(iris.data,iris.target)

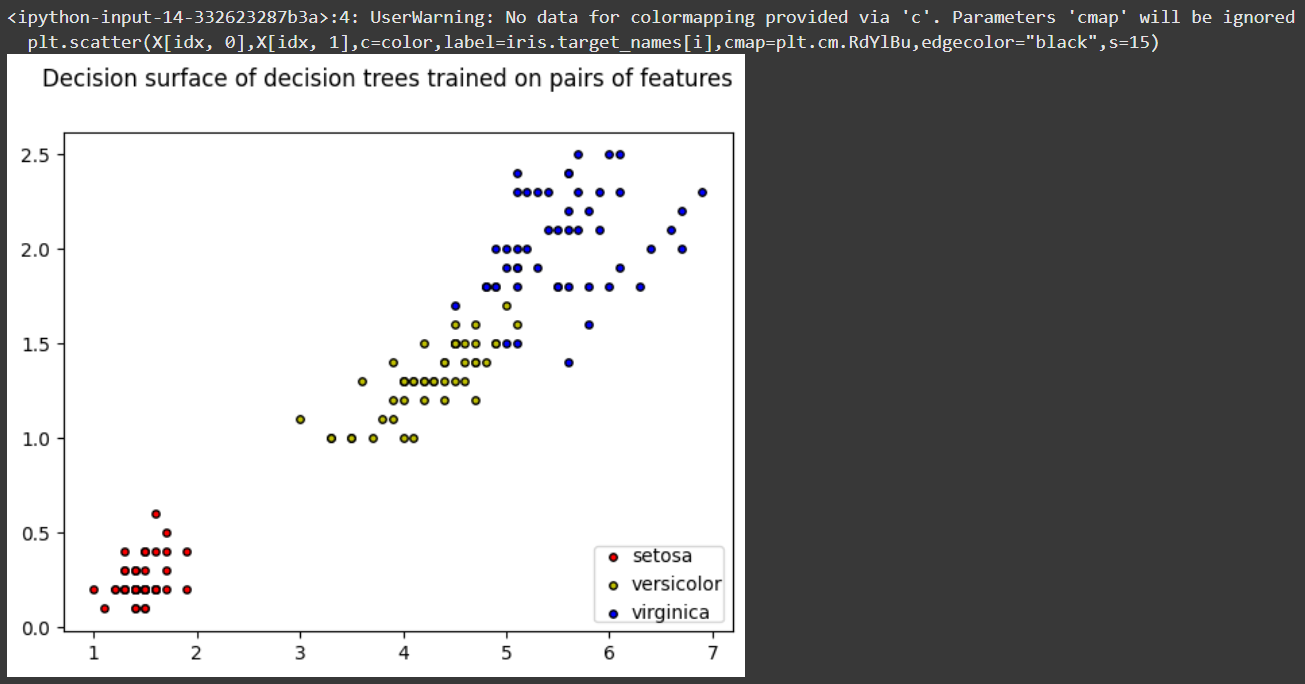
plot\_tree(clf, filled=True)

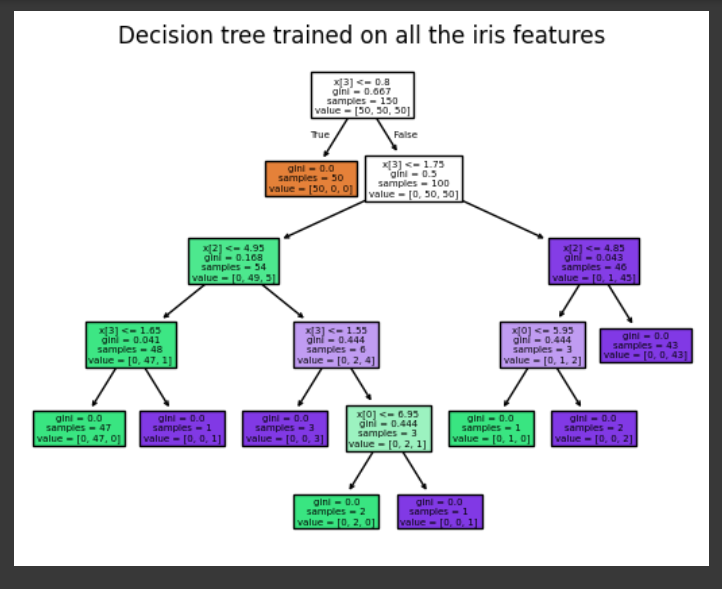
plt.title("Decision tree trained on all the iris features")

plt.show()

**OUTPUT:**







**Result:**

Thus, the python program to implement Decision Tree has been successfully implemented and the results have been verified and analysed.

**Ex. No.: 8**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT ADA BOOSTING**

**Aim:**

**To implement a python program for Ada Boosting.**

**Algorithm:**

**Step 1: Import Necessary Libraries**

Import numpy as np.

Import pandas as pd.

Import DecisionTreeClassifier from sklearn.tree.

Import train\_test\_split from sklearn.model\_selection.

Import accuracy\_score from sklearn.metrics.

**Step 2: Load and Prepare Data**

Load your dataset using pd.read\_csv() (e.g., df = pd.read\_csv('data.csv')).

Separate features (X) and target (y).

Split the dataset into training and testing sets using train\_test\_split().

**Step 3: Initialize Parameters**

Set the number of weak classifiers n\_estimators.

Initialize an array weights for instance weights, setting each weight to 1 / number\_of\_samples.

**Step 4: Train Weak Classifiers**

Loop for n\_estimators iterations:

Train a weak classifier using DecisionTreeClassifier(max\_depth=1) on the training data weighted by weights.

Predict the target values using the trained weak classifier.

Calculate the error rate err as the sum of weights of misclassified samples divided by the sum of all weights.

Compute the classifier's weight alpha using 0.5 \* np.log((1 - err) / err).

Update the weights: multiply the weights of misclassified samples by np.exp(alpha) and the weights of correctly classified samples by np.exp(-alpha).

Normalize the weights so that they sum to 1.

Append the trained classifier and its weight to lists classifiers and alphas.

**Step 5: Make Predictions**

For each sample in the testing set:

Initialize a prediction score to 0.

For each trained classifier and its weight:

Add the classifier's prediction (multiplied by its weight) to the prediction score.

Take the sign of the prediction score as the final prediction.

**Step 6: Evaluate the Model**

Compute the accuracy of the AdaBoost model on the testing set using accuracy\_score().

**Step 7: Output Results**

Print or plot the final accuracy and possibly other evaluation metrics.

**Code:**

import pandas as pd

import numpy as np

from mlxtend.plotting import plot\_decision\_regions

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]

import seaborn as sns

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

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

from sklearn.tree import DecisionTreeClassifier

dt1 = DecisionTreeClassifier(max\_depth=1)

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

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

# Step 2 - Train 1st Model

dt1.fit(x,y)

from sklearn.tree import plot\_tree

plot\_decision\_regions (x,y,clf=dt1, legend=2)

df['y pred'] = dt1.predict(x)

def calculate\_model\_weight(error):

  return 0.5\*np.log((1-error)/(error))

# Step - 3 Calculate model weight

alpha1 = calculate\_model\_weight(0.3)

alpha1

# 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() # Calculating normalized weights by dividing updated weights by sum of all updated weights

df['normalized\_weights'].sum()

df['cumsum\_upper'] = np.cumsum(df['normalized\_weights'])

df['cumsum\_lower']=df['cumsum\_upper'] - df['normalized\_weights']

df[['X1','X2','label','weights','y\_pred','updated\_weights','cumsum\_lower','cumsum\_upper']]

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)

index\_values

second\_df = df.iloc[index\_values,[0,1,2,3]]

second\_df

dt2 = DecisionTreeClassifier(max\_depth=1)

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

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

dt2.fit(x,y)

plot\_tree(dt2)

plot\_decision\_regions(x, y, clf=dt2, legend=2)

second\_df['y\_pred'] = dt2.predict(x)

second\_df

alpha2 = calculate\_model\_weight(0.1)

alpha2

# Step 4 - Update weights

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

second\_df['normalized\_weights'] = second\_df['updated\_weights'] / second\_df['updated\_weights'].sum()

second\_df['normalized\_weights'].sum()

second\_df['cumsum\_upper'] = np.cumsum(second\_df['normalized\_weights'])

second\_df['cumsum\_lower'] = second\_df['cumsum\_upper'] - second\_df['normalized\_weights']

second\_df[['X1','X2','label','weights','y\_pred','normalized\_weights','cumsum\_lower','cumsum\_upper']]

alpha3 = calculate\_model\_weight(0.7)

alpha3

from sklearn.tree import DecisionTreeClassifier

print(alpha1,alpha2,alpha3)

dt3 = DecisionTreeClassifier(max\_depth=2)

# Fit dt3 before making predictions

dt3.fit(x, y) # Assuming 'x' and 'y' are your training data from previous cells.

query = np.array([1,5]).reshape(1,2)

dt1.predict(query)

dt2.predict(query)

dt3.predict(query)

alpha1\*1 + alpha2\*(1) + alpha3\*(1)

np.sign(1.09)

query = np.array([9,9]).reshape(1,2)

dt1.predict(query)

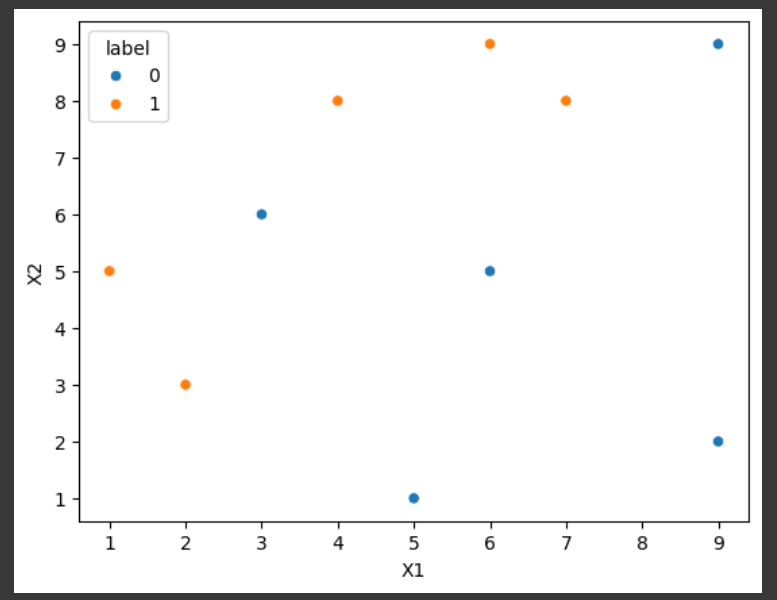
dt2.predict(query)

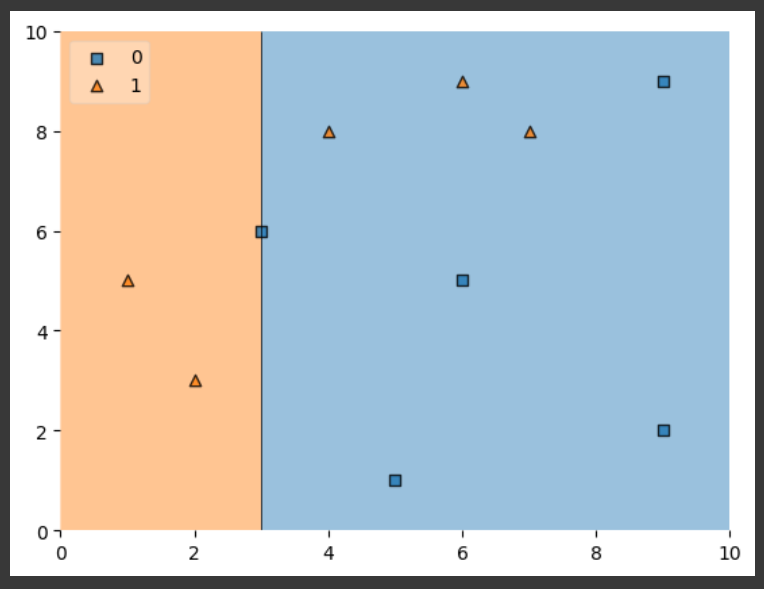
dt3.predict(query)

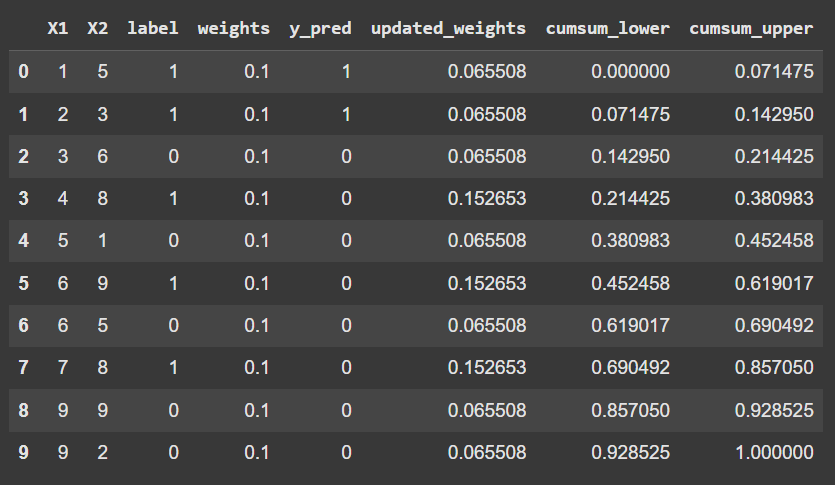
alpha1\*(1) + alpha2\*(-1) + alpha3\*(-1)

np.sign(-0.25)

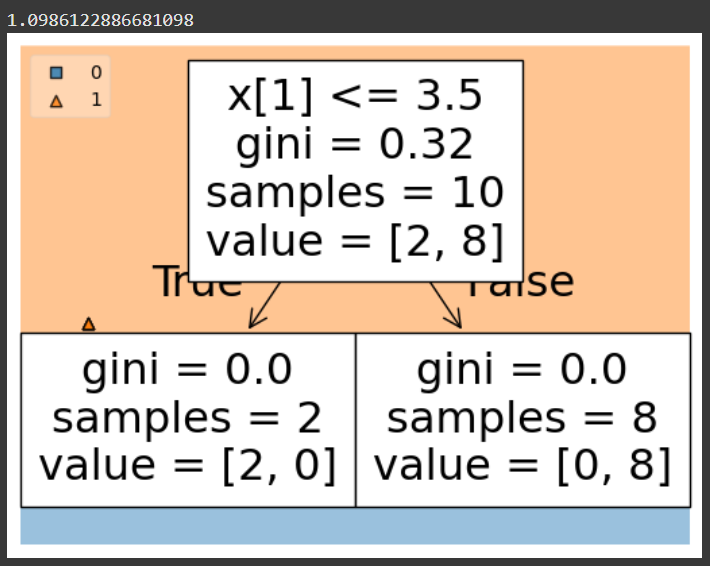
**Output:**



****

****

****

****

****

****

****

**Result:**

Thus the python program to implement ADA Boosting has been successfully implemented and the results have been verified and analyzed.

**Ex. No.: 9**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT KNN MODEL**

**Aim:**

To implement a python program using a KNN Algorithm in a model.

**Algorithm:**

* 1. **Import Necessary Libraries**
* Import necessary libraries: pandas, numpy, train\_test\_split from sklearn.model\_selection, StandardScaler from sklearn.preprocessing, KNeighborsClassifier from sklearn.neighbors, and classification\_report and confusion\_matrix from sklearn.metrics.
  1. Load and Explore the Dataset
* Load the dataset using pandas.
* Display the first few rows of the dataset using df.head().
* Display the dimensions of the dataset using df.shape().
* Display the descriptive statistics of the dataset using df.describe().

1. Preprocess the Data

* Separate the features (X) and the target variable (y).
* Split the data into training and testing sets using train\_test\_split.
* Standardize the features using StandardScaler.

1. Train the KNN Model

* Create an instance of KNeighborsClassifier with a specified number of neighbors (k).
* For each data point, calculate the Euclidean distance to all other data points.
* Select the K nearest neighbors based on the calculated Euclidean distances.
* Among the K nearest neighbors, count the number of data points in each category.
* Assign the new data point to the category for which the number of neighbors is maximum.

1. Make Predictions

* Use the trained model to make predictions on the test data.
* Evaluate the Model
* Generate the confusion matrix and classification report using the actual and predicted values.
* Print the confusion matrix and classification report.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

dataset = pd.read\_csv('../input/mall-customers/Mall\_Customers.csv')

X = dataset.iloc[:,[3,4]].values

print(dataset)

from sklearn.cluster import KMeans

wcss =[]

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)

    wcss.append(kmeans.inertia\_)

# Plot the graph to visualize the Elbow Method to find the optimal number of cluster

plt.plot(range(1,11),wcss)

plt.title('The Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('WCSS')

plt.show()

kmeans=KMeans(n\_clusters= 5, init = 'k-means++', max\_iter = 300, n\_init = 10, random\_state = 0)

y\_kmeans = kmeans.fit\_predict(X)

y\_kmeans

type(y\_kmeans)

y\_kmeans

plt.scatter(X[y\_kmeans == 0, 0], X[y\_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')

plt.scatter(X[y\_kmeans == 1, 0], X[y\_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')

plt.scatter(X[y\_kmeans == 2, 0], X[y\_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')

plt.scatter(X[y\_kmeans == 3, 0], X[y\_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')

plt.scatter(X[y\_kmeans == 4, 0], X[y\_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')

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()

plt.scatter(X[y\_kmeans == 0, 0], X[y\_kmeans == 0, 1], s = 100, c = 'red', label = 'Cluster 1')

plt.scatter(X[y\_kmeans == 1, 0], X[y\_kmeans == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')

plt.scatter(X[y\_kmeans == 2, 0], X[y\_kmeans == 2, 1], s = 100, c = 'green', label = 'Cluster 3')

plt.scatter(X[y\_kmeans == 3, 0], X[y\_kmeans == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')

plt.scatter(X[y\_kmeans == 4, 0], X[y\_kmeans == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5')

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 and analyzed.

**Ex. No.: 10**

**Date:**

**A PYTHON PROGRAM TO IMPLEMENT DIMENSIONALITY REDUCTION USING PCA**

**Aim:**

To implement Dimensionality Reduction using PCA in a python program.

**Algorithm:**

Step 1: Import Libraries

Import necessary libraries, including pandas, numpy, matplotlib.pyplot, and sklearn.decomposition.PCA.

Step 2: Load the Dataset (iris dataset)

Load your dataset into a pandas DataFrame.

Step 3: Standardize the Data

Standardize the features of the dataset using StandardScaler from sklearn.preprocessing.

Step 4: Apply PCA

* Create an instance of PCA with the desired number of components.
* Fit PCA to the standardized data.
* Transform the data to its principal components using transform.

Step 5: Explained Variance Ratio

* Calculate the explained variance ratio for each principal component.
* Plot a scree plot to visualize the explained variance ratio.

Step 6: Choose the Number of Components

Based on the scree plot, choose the number of principal components that explain a significant amount of variance.

Step 7: Apply PCA with Chosen Components

Apply PCA again with the chosen number of components.

Step 8: Visualize the Reduced Data

* Transform the original data to the reduced dimension using the fitted PCA.
* Visualize the reduced data using a scatter plot.

Step 9: Interpretation

Interpret the results, considering the trade-offs between dimensionality reduction and information loss.

**Code:**

from sklearn import datasets

import pandas as pd

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

import seaborn as sns

iris = datasets.load\_iris()

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

df.head()

scalar = StandardScaler()

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

scaled\_data

sns.heatmap(scaled\_data.corr())

pca = PCA(n\_components = 3)

pca.fit(scaled\_data)

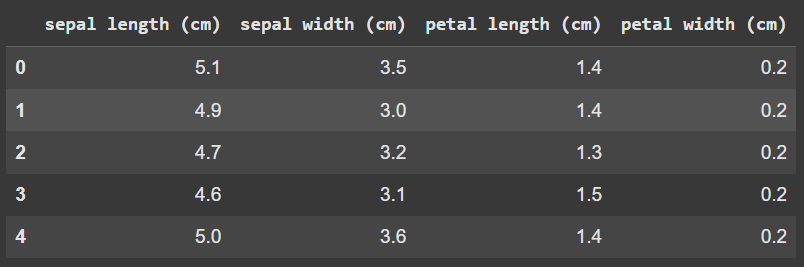
data\_pca = pca.transform(scaled\_data)

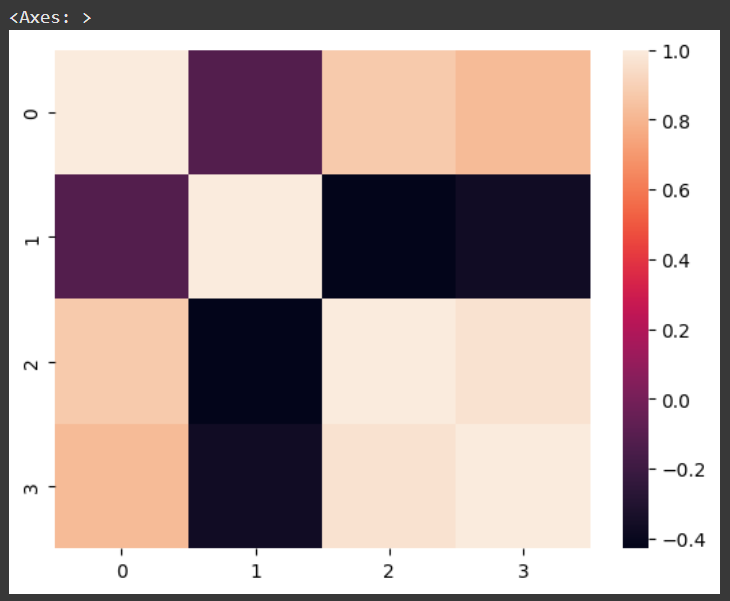
data\_pca = pd.DataFrame(data\_pca,columns=['PC1','PC2','PC3'])

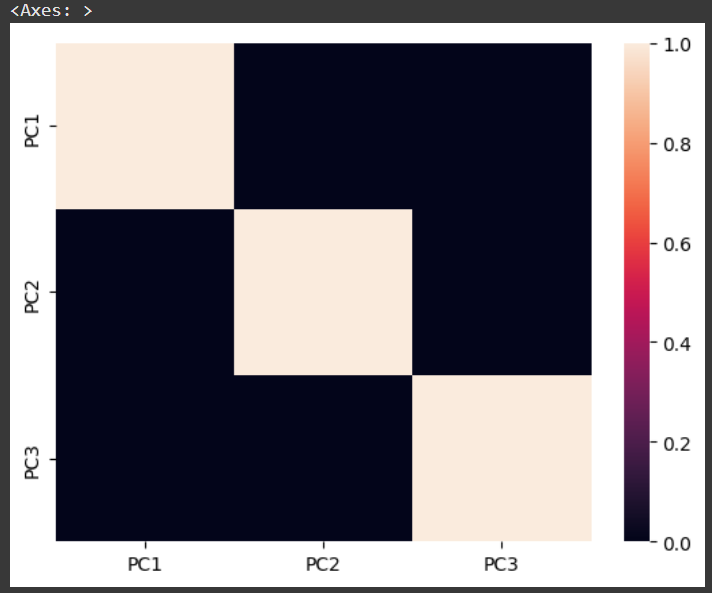
data\_pca.head()

sns.heatmap(data\_pca.corr())

**Output:**

****

****



**Result:**

Thus the python program to implement Dimensionality Reduction using PCA has been successfully implemented and the results have been verified and analyzed.