**Practical-1(A)**

**Aim**: Design a simple machine learning model to train the training instances and test the same.

**Description:-**

**Simple Machine learning model**

Linear regression is the simplest machine learning model in which we try to predict one output variable using one or more input variables. The representation of linear regression is a linear equation, which combines a set of input values(x) and predicted output(y) for the set of those input values.

**Code:-**

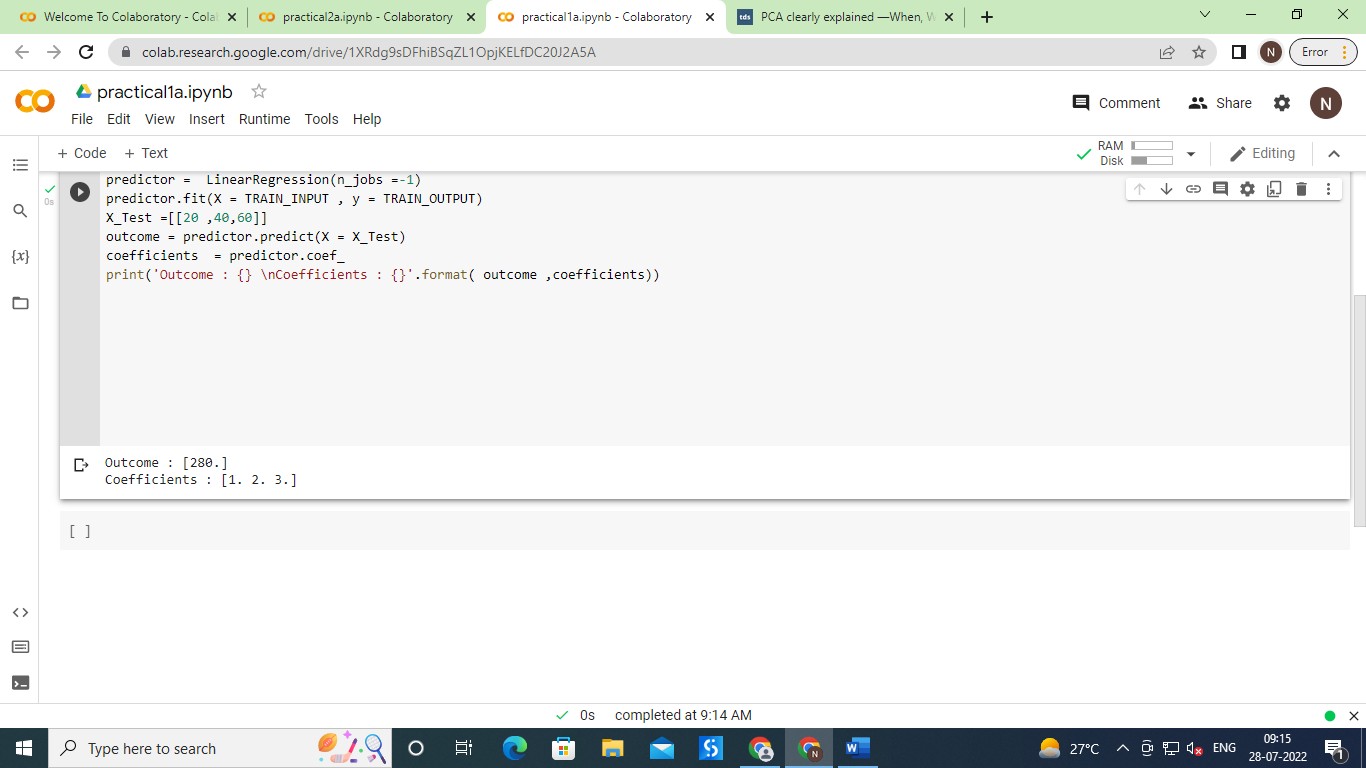
from random import randint TRAIN\_SET\_LIMIT=1000 TRAIN\_SET\_COUNT=100 TRAIN\_INPUT = list() TRAIN\_OUTPUT = list()

for i in range (TRAIN\_SET\_COUNT) : a = randint(0, TRAIN\_SET\_LIMIT ) b = randint(0, TRAIN\_SET\_LIMIT ) c = randint(0, TRAIN\_SET\_LIMIT ) op = a + (2 \* b) + (3 \* c) TRAIN\_INPUT.append([a ,b, c]) TRAIN\_OUTPUT.append(op)

from sklearn.linear\_model import LinearRegression predictor = LinearRegression(n\_jobs =-1) predictor.fit(X = TRAIN\_INPUT , y = TRAIN\_OUTPUT) X\_Test =[[20 ,40,60]]

outcome = predictor.predict(X = X\_Test) coefficients = predictor.coef\_

print('Outcome : {} \nCoefficients : {}'.format( outcome ,coefficients))

**Output:-**

**Practical-1(B)**

**Aim**: Implement and demonstrate the FIND-S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a .CSV file

**Description:-**

**Find-S algorithm**

The find-S algorithm is a basic concept learning algorithm in machine learning. The find-S algorithm finds the most specific hypothesis that fits all the positive examples. We have to note here that the algorithm considers only those positive training example.

**Code:-**

import csv num\_attributes = 6 a =[]

print("\n The Given Training Data Set \n") with open('enjoysports.csv' ,'r') as csvfile :

reader = csv.reader(csvfile) for row in reader:

a.append (row) print(row)

print("\n the initial value of hypothesis :") hypothesis = ['0'] \* num\_attributes print(hypothesis)

for j in range (0,num\_attributes) :

hypothesis[j] = a[0][j];

print("\n Find S: Finding a Maximally Specific Hypothesis\n") for i in range( 0, len(a)) :

if a[i] [num\_attributes] =='yes' :

for j in range (0, num\_attributes) :

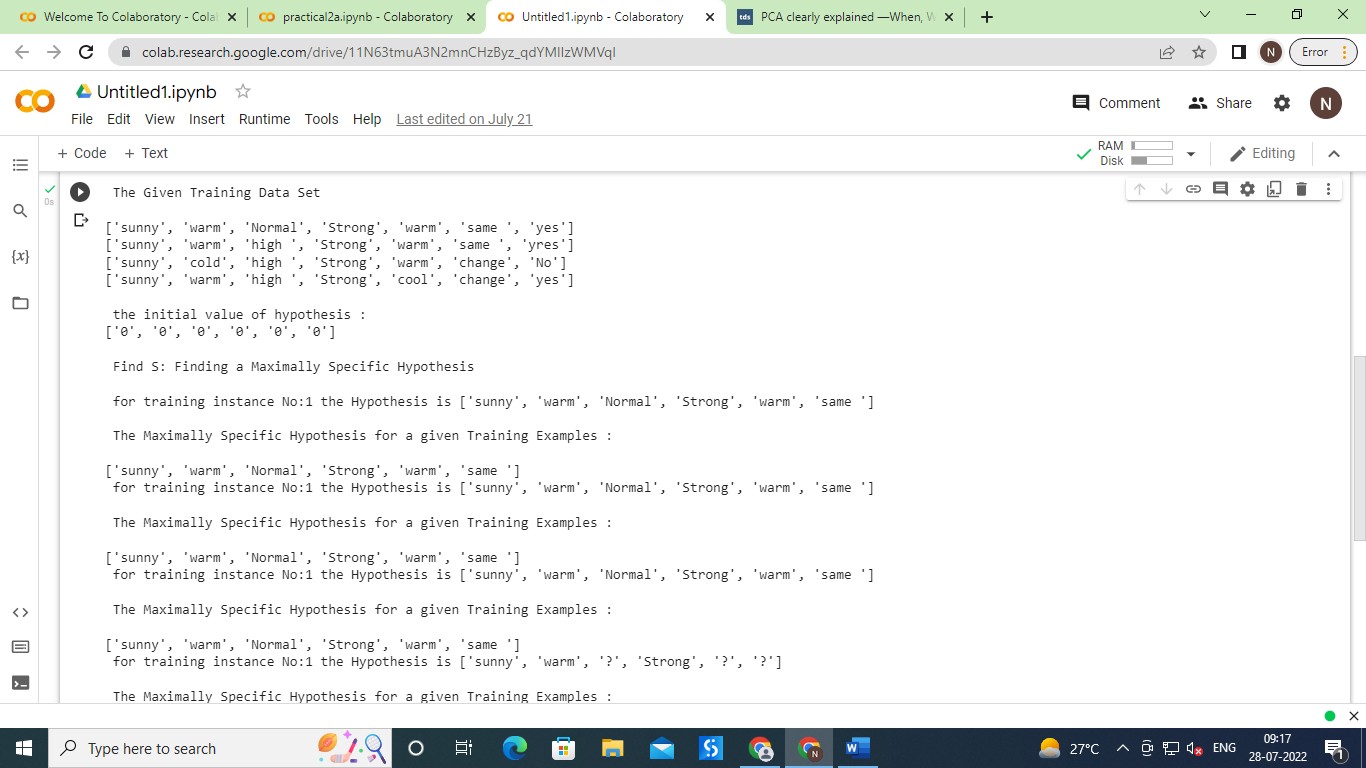
if a[i][j]!= hypothesis[j] :

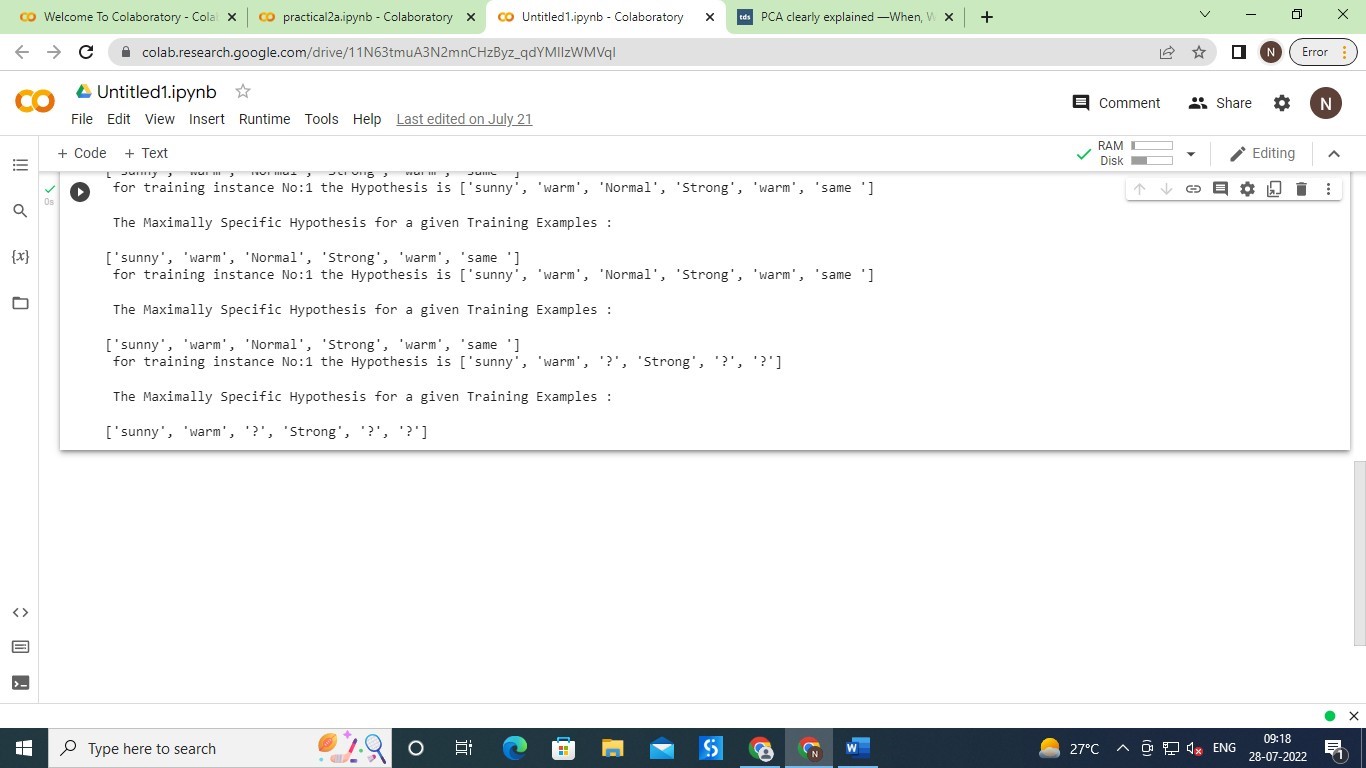
hypothesis[j] ='?'

else : hypothesis[j] = a[i][j]

print(" for training instance No:{0} the Hypothesis is".format(1) ,hypothesis) print("\n The Maximally Specific Hypothesis for a given Training Examples :\n") print(hypothesis)

**Output:-**





**Practical 2(A)**

**Aim:** Perform Data Loading, Feature selection (Principal Component analysis) and Feature Scoring and Ranking.

**Description:-**

**Principal component analysis**

Principal component analysis, or PCA, is a statistical technique to convert high dimensional data to low dimensional data by selecting the most important features that capture maximum information about the dataset. The features are selected on the basis of variance that they cause in the output. The feature that causes highest variance is the first principal component will be ignored. The feature that is responsible for second highest variance is considered the second principal component, and so on. It is important to mention that principal components do not have any correlation with each other.

**Code:-**

import numpy as np

import matplotlib.pyplot as plt from sklearn import datasets

from sklearn.decomposition import PCA import pandas as pd

from sklearn.preprocessing import StandardScaler plt.style.use('ggplot')

# Load the data

iris = datasets.load\_iris()

X = iris.data y = iris.target

# Z-score the features scaler = StandardScaler() scaler.fit(X)

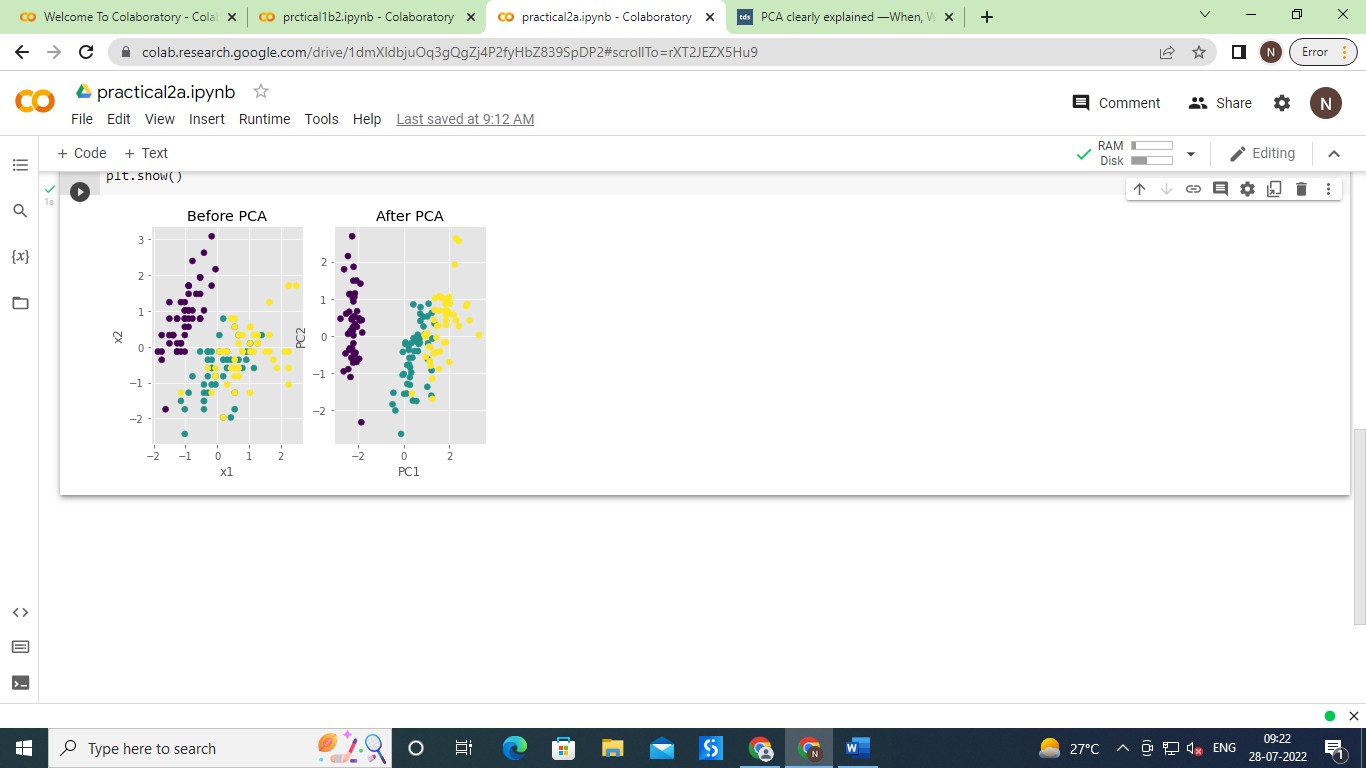
X = scaler.transform(X)# The PCA model

pca = PCA(n\_components=2) # estimate only 2 PCs X\_new = pca.fit\_transform(X)

fig, axes = plt.subplots(1,2) axes[0].scatter(X[:,0], X[:,1], c=y) axes[0].set\_xlabel('x1') axes[0].set\_ylabel('x2') axes[0].set\_title('Before PCA') axes[1].scatter(X\_new[:,0], X\_new[:,1], c=y) axes[1].set\_xlabel('PC1') axes[1].set\_ylabel('PC2') axes[1].set\_title('After PCA')

plt.show()

**Output:-**



**Practical 2(B)**

**Aim:** For a given set of training data examples stored in a .CSV file, implement and demonstrate the Candidate-Elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

**Description:-**

**Hypothesis consistent**.

The candidate elimination algorithm incrementally builds the version space given a hypothesis space H and a set E of examples. The examples are added one by one; each example possibly shrinks the version space by removing the hypotheses that are inconsistent with the example. The candidate elimination algorithm does this by updating the general and specific boundary for each new example. You can consider this as an extended form of Find-S algorithm.

Consider both positive and negative examples.

Actually, positive examples are used here as Find-S algorithm (Basically they are generalizing from the specification).

While the negative example is specified from generalize form.

**Terms :-**

* **General Hypothesis**: Not Specifying features to learn the machine. G = {‘?’, ‘?’,’?’,’?’…}: Number of attributes.
* **Specific Hypothesis:** Specifying features to learn machine (Specific feature). S= {‘pi’,’pi’,’pi’…}: Number of pi depends on number of attributes.
* **Version Space:** It is intermediate of general hypothesis and Specific hypothesis. It not only just written one hypothesis but a set of all possible hypothesis based on training data-set.

**Code:-**

import csv

with open("sample\_data/enjoysports.csv") as f: csv\_file=csv.reader(f)

data=list(csv\_file) s=data[1][:-1]

g=[['?' for i in range(len(s))] for j in range(len(s))] for i in data:

if i[-1]=="Yes":

for j in range(len(s)): if i[j]!=s[j]:

s[j]='?'

g[j][j]='?'

elif i[-1]=="No":

for j in range(len(s)): if i[j]!=s[j]:

g[j][j]=s[j] else:

g[j][j]="?"

print("\nSteps of Candidate Elimination Algorithm",data.index(i)+1) print(s)

print(g) gh=[]

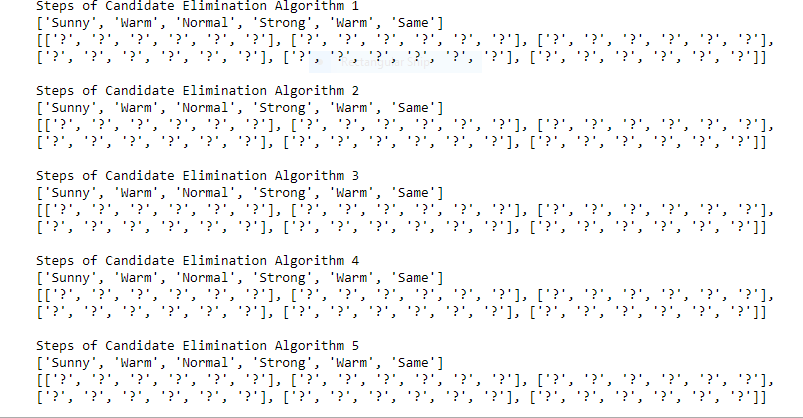
for i in g:

for j in i: if j!='?':

gh.append(i) break

print("\nFinal specific hypothesis:\n",s) print("\nFinal general hypothesis:\n",gh)

**Output:-**



**Practical 3(A)**

**Aim:** Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets.

**Description:-**

**Naïve Bayes Classifier**

Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions. It is a probabilistic classifier, which means it predicts on the basis of the probability of an object

**Code:-**

import numpy as np import pandas as pd

from sklearn import datasets wine= datasets.load\_wine() print(wine)

print("Feature:",wine.feature\_names) print("Labels:",wine.target\_names) X=pd.DataFrame(wine['data']) print(X.head(0)) print(wine.data.shape) y=print(wine.target)

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

X\_train, X\_test, y\_train,y\_test = train\_test\_split(wine.data,wine.target,test\_size=0.30,rando m\_state=109)

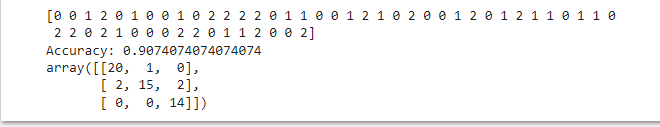
from sklearn.naive\_bayes import GaussianNB gnb=GaussianNB()

gnb.fit(X\_train ,y\_train) y\_pred =gnb.predict(X\_test) print(y\_pred)

from sklearn import metrics print("Accuracy:",metrics.accuracy\_score(y\_test,y\_pred)) from sklearn.metrics import confusion\_matrix cm=np.array(confusion\_matrix(y\_test ,y\_pred))

print(cm)

**Output:-**



Practical 3(B)

**Aim:** Write a program to implement Decision Tree and Random forest with Prediction, Test Score and Confusion Matrix.

**Description:-**

**Decision Tree Classifier :-**

* Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome.
* In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
* The decisions or the test are performed on the basis of features of the given dataset.

**Random Forest Classifier :-**

* Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model.
* As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."

**Code:-**

import pandas as pd import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

%matplotlib inline sns.set\_style("whitegrid") plt.style.use("fivethirtyeight")

df = pd.read\_csv("sample\_data/WA\_Fn-UseC\_-HR-Employee-Attrition.csv") df.head()

sns.countplot(x='Attrition', data=df)

df.drop(['EmployeeCount', 'EmployeeNumber', 'Over18', 'StandardHours'], axis="columns", i nplace=True)

categorical\_col = []

for column in df.columns:

if df[column].dtype == object and len(df[column].unique()) <= 50: categorical\_col.append(column)

df['Attrition'] = df.Attrition.astype("category").cat.codes categorical\_col.remove('Attrition')

from sklearn.preprocessing import LabelEncoder label = LabelEncoder()

for column in categorical\_col:

df[column] = label.fit\_transform(df[column])



from sklearn.model\_selection import train\_test\_split X = df.drop('Attrition', axis=1)

y = df.Attrition

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42) from sklearn.metrics import accuracy\_score,confusion\_matrix,classification\_report def print\_score(clf, X\_train, y\_train, X\_test, y\_test, train=True):

if train:

pred = clf.predict(X\_train)

clf\_report = pd.DataFrame(classification\_report(y\_train, pred, output\_dict=True)) print("Train Result:\n================================================")

print(f"Accuracy Score: {accuracy\_score(y\_train, pred) \* 100:.2f}%") print(" ") print(f"CLASSIFICATION REPORT:\n{clf\_report}")

print(" ") print(f"Confusion Matrix: \n {confusion\_matrix(y\_train, pred)}\n")

elif train==False:

pred = clf.predict(X\_test)

clf\_report = pd.DataFrame(classification\_report(y\_test, pred, output\_dict=True)) print("Test Result:\n================================================")

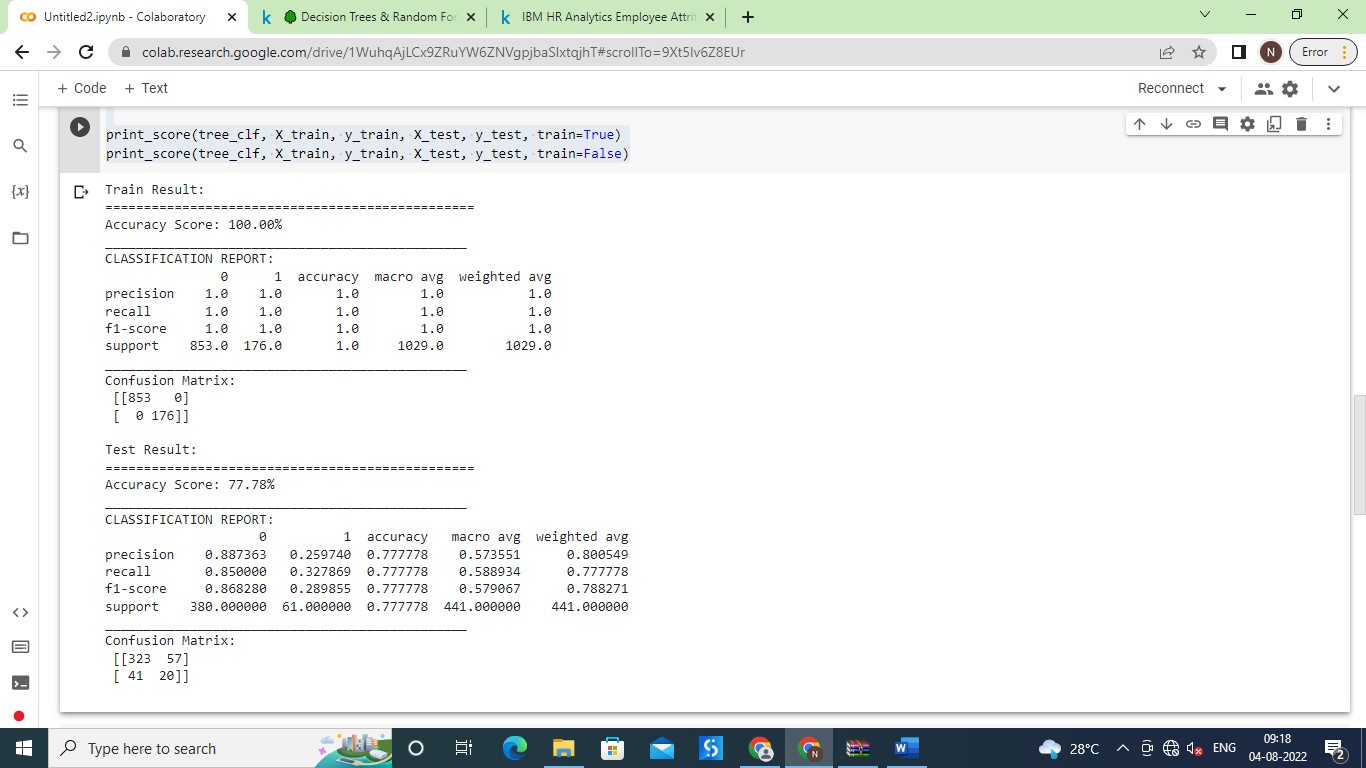
print(f"Accuracy Score: {accuracy\_score(y\_test, pred) \* 100:.2f}%") print(" ") print(f"CLASSIFICATION REPORT:\n{clf\_report}")

print(" ") print(f"Confusion Matrix: \n {confusion\_matrix(y\_test, pred)}\n")

from sklearn.tree import DecisionTreeClassifier

tree\_clf = DecisionTreeClassifier(random\_state=42) tree\_clf.fit(X\_train, y\_train)

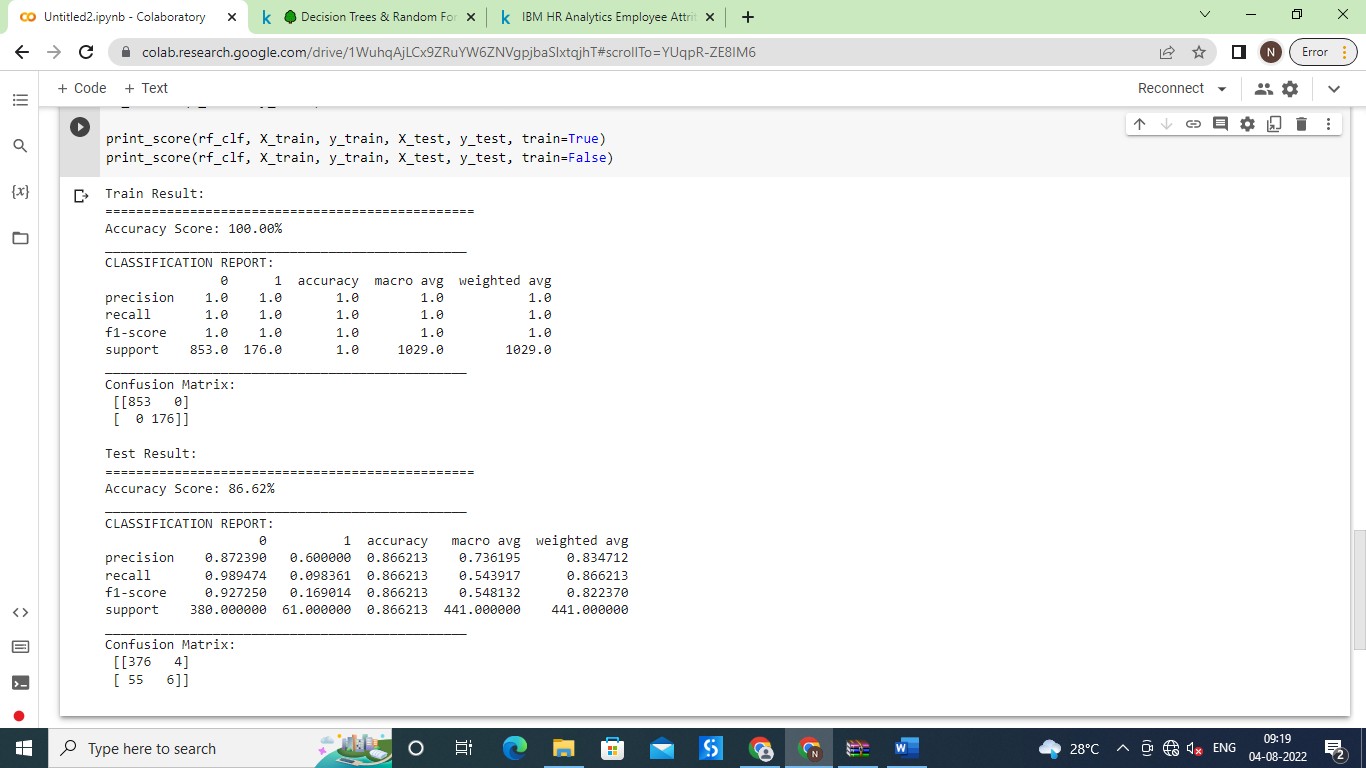
print\_score(tree\_clf, X\_train, y\_train, X\_test, y\_test, train=True) print\_score(tree\_clf, X\_train, y\_train, X\_test, y\_test, train=False)



from sklearn.ensemble import RandomForestClassifier rf\_clf = RandomForestClassifier(n\_estimators=100) rf\_clf.fit(X\_train, y\_train)

print\_score(rf\_clf, X\_train, y\_train, X\_test, y\_test, train=True) print\_score(rf\_clf, X\_train, y\_train, X\_test, y\_test, train=False)

**Output**:-



**Practical 4(A)**

**Aim:** For a given set of training data examples stored in a .CSV file implement Least Square Regression algorithm.

**Description:-**

**Least Squares method:-**

The least squares method is a form of mathematical regression analysis used to determine the line of best fit for a set of data, providing a visual demonstration of the relationship between the data points. Each point of data represents the relationship between a known independent variable and an unknown dependent variable.

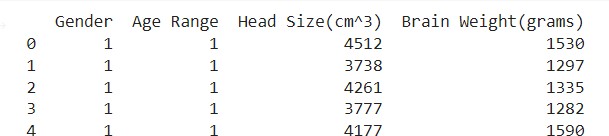
**Code:-**

import numpy as np import pandas as pd

import matplotlib.pyplot as plt data=pd.read\_csv("/content/sample\_data/headbrain.csv") print(data.shape)

### (237, 4)

print(data.head())



X=data['Head Size(cm^3)'].values Y=data['Brain Weight(grams)'].values mean\_x=np.mean(X) mean\_y=np.mean(Y)

n=len(X) numer = 0

denom = 0

for i in range(n):

numer+= (X[i] - mean\_x) \* (Y[i]- mean\_y) denom +=(X[i] - mean\_x) \*\* 2

m = numer/denom

c= mean\_y -( m \* mean\_x) print("Coefficients") print(m,c)

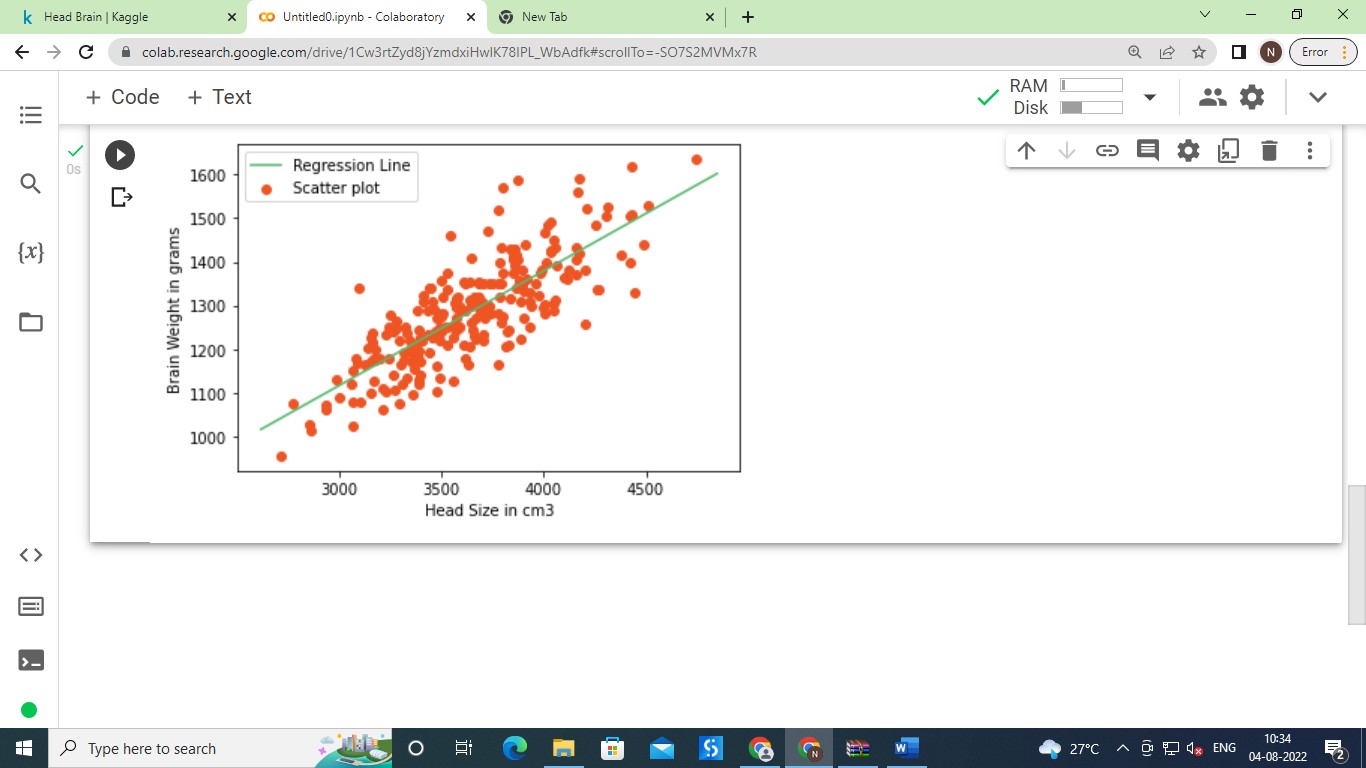
max\_x = np.max(X) + 100 min\_x = np.min(X) - 100

x= np.linspace(min\_x ,max\_x , 1000) y=c + m \* x

plt.plot(x, y , color='#58b970', label='Regression Line') plt.scatter(X,Y ,c ='#ef5423',label='Scatter plot')

plt.xlabel('Head Size in cm3') plt.ylabel('Brain Weight in grams') plt.legend()

plt.show()



Practical-4**(B)**

**Aim:** For a given set of training data examples stored in a .CSV file implement Logistic Regression algorithm

**Description:-**

**Logistic Regression algorithm:-**

Logistic regression is a statistical method that is used for building machine learning models where the dependent variable is dichotomous: i.e. binary. Logistic regression is used to describe data and the relationship between one dependent variable and one or more independent variables

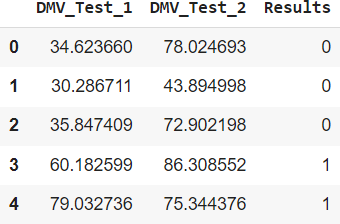
**Code:-**

import numpy as np

import matplotlib.pyplot as plt import pandas as pd

dataset = pd.read\_csv('https://raw.githubusercontent.com/mk- gurucharan/Classification/master/DMVWrittenTests.csv')

X = dataset.iloc[:, [0, 1]].values y = dataset.iloc[:, 2].values dataset.head(5)



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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.25, random\_state = 0) from sklearn.linear\_model import LogisticRegression

classifier = LogisticRegression() classifier.fit(X\_train, y\_train)

### LogisticRegression()

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



from sklearn.metrics import confusion\_matrix cm = confusion\_matrix(y\_test, y\_pred)

from sklearn.metrics import accuracy\_score

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

**Output:-**



**Practical-5(A)**

**Aim**: Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

**Description:-**

**ID3 algorithm**

ID3 algorithm stands for Iterative Dichotomiser 3, is a classification algorithm that follows a greedy approach of building a decision tree by selecting a best attribute that yields maximum Information Gain (IG) or minimum Entropy (H)

**Decision Trees**

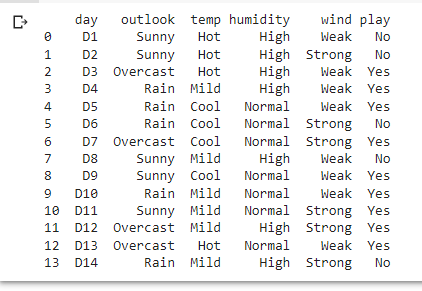
* In simple words, a decision tree is a structure that contains nodes (rectangular boxes) and edges(arrows) and is built from a dataset (table of columns representing features/attributes and rows corresponds to records). Each node is either used to make a decision (known as decision node) or represent an outcome (known as leaf node).
* The initial node is called the root node (colored in blue), the final nodes are called the leaf nodes (colored in green) and the rest of the nodes are called intermediate or internal nodes.

**Code:-**

import numpy as np import pandas as pd eps = np.finfo(float).eps

from numpy import log2 as log

df\_tennis = pd.read\_csv('sample\_data/play\_tennis.csv') print( df\_tennis)



df = pd.DataFrame(df\_tennis,columns=['outlook','temp','humidity','wind','play']) ##1. claculate entropy o the whole dataset

entropy\_node = 0 #Initialize Entropy

values = df.play.unique() #Unique objects - 'Yes', 'No' for value in values:

fraction = df.play.value\_counts()[value]/len(df.play) entropy\_node += -fraction\*np.log2(fraction)

def ent(df,attribute): target\_variables = df.play.unique() variables = df[attribute].unique() entropy\_attribute = 0

for variable in variables: entropy\_each\_feature = 0

for target\_variable in target\_variables:

num = len(df[attribute][df[attribute]==variable][df.play ==target\_variable]) #numerat

or

den = len(df[attribute][df[attribute]==variable])

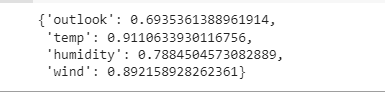
#denominator

fraction = num/(den+eps) #pi entropy\_each\_feature += -fraction\*log(fraction+eps)

#This calculates entropy for one feature like 'Sweet' fraction2 = den/len(df)

entropy\_attribute += -fraction2\*entropy\_each\_feature #Sums up all the entropy ETaste return(abs(entropy\_attribute))

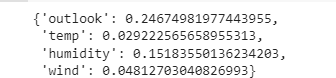
a\_entropy = {k:ent(df,k) for k in df.keys()[:-1]} a\_entropy



def ig(e\_dataset,e\_attr): return(e\_dataset-e\_attr)

#entropy\_node = entropy of dataset #a\_entropy[k] = entropy of k(th) attr

IG = {k:ig(entropy\_node,a\_entropy[k]) for k in a\_entropy} IG



def find\_entropy(df):

Class = df.keys()[-1] #To make the code generic, changing target variable class name entropy = 0

values = df[Class].unique() for value in values:

fraction = df[Class].value\_counts()[value]/len(df[Class]) entropy += -fraction\*np.log2(fraction)

return entropy

def find\_entropy\_attribute(df,attribute):

Class = df.keys()[-1]

target\_variables = df[Class].unique() #This gives all 'Yes' and 'No' variables = df[attribute].unique()

entropy2 = 0

for variable in variables: entropy = 0

for target\_variable in target\_variables:

num = len(df[attribute][df[attribute]==variable][df[Class] ==target\_variable]) den = len(df[attribute][df[attribute]==variable])

fraction = num/(den+eps)

entropy += -fraction\*log(fraction+eps) fraction2 = den/len(df)

entropy2 += -fraction2\*entropy return abs(entropy2)

def find\_winner(df): Entropy\_att = [] IG = []

for key in df.keys()[:-1]: #Entropy\_att.append(find\_entropy\_attribute(df,key))

IG.append(find\_entropy(df)-find\_entropy\_attribute(df,key)) return df.keys()[:-1][np.argmax(IG)]

def get\_subtable(df, node,value):

return df[df[node] == value].reset\_index(drop=True) def buildTree(df,tree=None):

Class = df.keys()[-1]

#Get attribute with maximum information gain node = find\_winner(df)

#Get distinct value of that attribute e.g Salary is node and Low,Med and High are values attValue = np.unique(df[node])

#Create an empty dictionary to create tree if tree is None:

tree={} tree[node] = {}

#We make loop to construct a tree by calling this function recursively. #In this we check if the subset is pure and stops if it is pure.

for value in attValue:

subtable = get\_subtable(df,node,value)

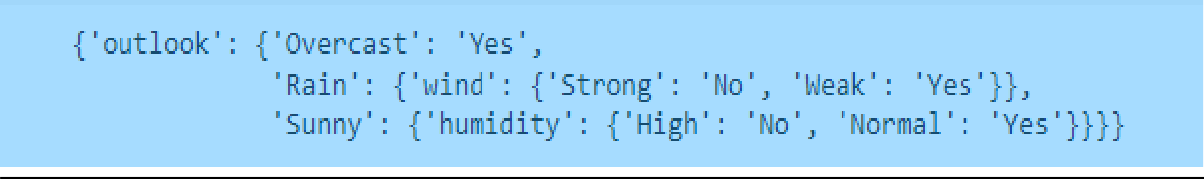
clValue,counts = np.unique(subtable['play'],return\_counts=True) if len(counts)==1:#Checking purity of subset

tree[node][value] = clValue[0] else:

tree[node][value] = buildTree(subtable) return tree

t=buildTree(df) import pprint pprint.pprint(t)

**Output:-**



**Practical-5(B)**

**Aim**: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set.

**Description:-**

**k-Nearest Neighbour:-**

The k-nearest neighbours algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another. K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories. K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm. K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.

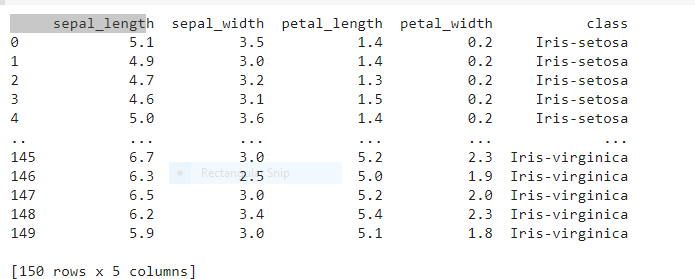
**Code:-**

import pandas as pd import numpy as np import operator

import matplotlib.pyplot as plt

data = pd.read\_csv('sample\_data/Iris.csv', header=None, names=['sepal\_l ength', 'sepal\_width', 'petal\_length', 'petal\_width', 'class'])

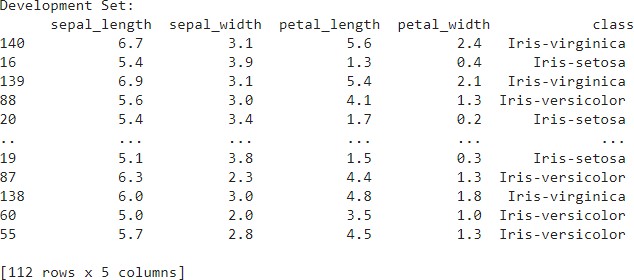
print(data)

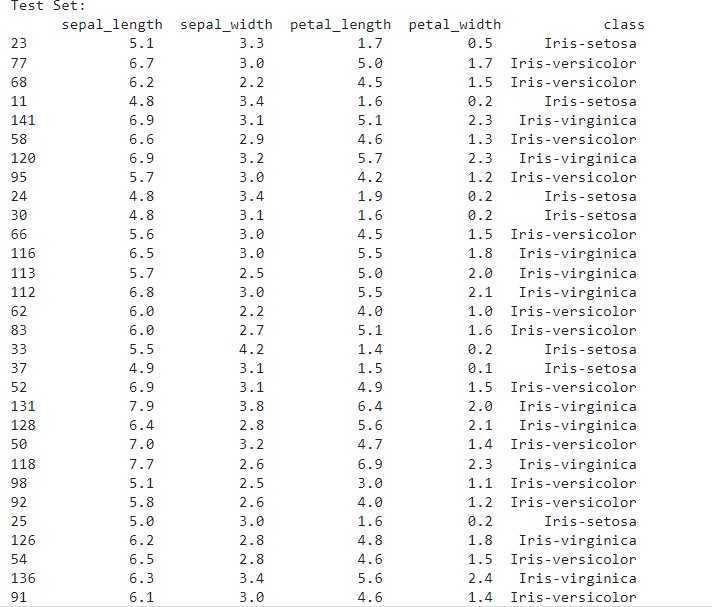


indices = np.random.permutation(data.shape[0]) div = int(0.75 \* len(indices))

development\_id, test\_id = indices[:div], indices[div:]

development\_set, test\_set = data.loc[development\_id,:], data.loc[test\_id,:] print("Development Set:\n", development\_set, "\n\nTest Set:\n", test\_set)





mean\_development\_set = development\_set.mean() mean\_test\_set = test\_set.mean() std\_development\_set = development\_set.std() std\_test\_set = test\_set.std()

test\_class = list(test\_set.iloc[:,-1]) dev\_class = list(development\_set.iloc[:,-1])

def euclideanDistance(data\_1, data\_2, data\_len): dist = 0

for i in range(data\_len):

dist = dist + np.square(data\_1[i] - data\_2[i]) return np.sqrt(dist)

def normalizedEuclideanDistance(data\_1, data\_2, data\_len, data\_mean, data\_std): n\_dist = 0

for i in range(data\_len):

n\_dist = n\_dist + (np.square(((data\_1[i] - data\_mean[i])/data\_std[i]) - ((data\_2[i] - data\_ mean[i])/data\_std[i])))

return np.sqrt(n\_dist)

def cosineSimilarity(data\_1, data\_2): dot = np.dot(data\_1, data\_2[:-1])

norm\_data\_1 = np.linalg.norm(data\_1) norm\_data\_2 = np.linalg.norm(data\_2[:-1]) cos = dot / (norm\_data\_1 \* norm\_data\_2) return (1-cos)

def knn(dataset, testInstance, k, dist\_method, dataset\_mean, dataset\_std): distances = {}

length = testInstance.shape[1]

if dist\_method == 'euclidean':

for x in range(len(dataset)):

dist\_up = euclideanDistance(testInstance, dataset.iloc[x], length) distances[x] = dist\_up[0]

elif dist\_method == 'normalized\_euclidean': for x in range(len(dataset)):

dist\_up = normalizedEuclideanDistance(testInstance, dataset.iloc[x], length, dataset\_ mean, dataset\_std)

distances[x] = dist\_up[0] elif dist\_method == 'cosine':

for x in range(len(dataset)):

dist\_up = cosineSimilarity(testInstance, dataset.iloc[x]) distances[x] = dist\_up[0]

# Sort values based on distance

sort\_distances = sorted(distances.items(), key=operator.itemgetter(1)) neighbors = []

# Extracting nearest k neighbors for x in range(k):

neighbors.append(sort\_distances[x][0])

# Initializing counts for 'class' labels counts as 0

counts = {"Iris-setosa" : 0, "Iris-versicolor" : 0, "Iris-virginica" : 0} # Computing the most frequent class

for x in range(len(neighbors)):

response = dataset.iloc[neighbors[x]][-1] if response in counts:

counts[response] += 1 else:

counts[response] = 1

# Sorting the class in reverse order to get the most frequest class

sort\_counts = sorted(counts.items(), key=operator.itemgetter(1), reverse=True) return(sort\_counts[0][0])

# Creating a list of list of all columns except 'class' by iterating through the development set row\_list = []

for index, rows in development\_set.iterrows():

my\_list =[rows.sepal\_length, rows.sepal\_width, rows.petal\_length, rows.petal\_width] row\_list.append([my\_list])

# k values for the number of neighbors that need to be considered k\_n = [1, 3, 5, 7]

# Distance metrics

distance\_methods = ['euclidean', 'normalized\_euclidean', 'cosine']

# Performing kNN on the development set by iterating all of the development set data point s and for each k and each distance metric

obs\_k = {}

for dist\_method in distance\_methods: development\_set\_obs\_k = {}

for k in k\_n: development\_set\_obs = [] for i in range(len(row\_list)):

development\_set\_obs.append(knn(development\_set, pd.DataFrame(row\_list[i]), k, di st\_method, mean\_development\_set, std\_development\_set))

development\_set\_obs\_k[k] = development\_set\_obs

# Nested Dictionary containing the observed class for each k and each distance metric (ob s\_k of the form obs\_k[dist\_method][k])

obs\_k[dist\_method] = development\_set\_obs\_k print(obs\_k)



accuracy = {}

for key in obs\_k.keys(): accuracy[key] = {}

for k\_value in obs\_k[key].keys(): #print('k = ', key)

count = 0

for i,j in zip(dev\_class, obs\_k[key][k\_value]): if i == j:

count = count + 1 else:

pass

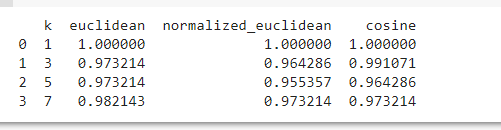
accuracy[key][k\_value] = count/(len(dev\_class))

# Storing the accuracy for each k and each distance metric into a dataframe df\_res = pd.DataFrame({'k': k\_n})

for key in accuracy.keys():

value = list(accuracy[key].values()) df\_res[key] = value

print(df\_res)

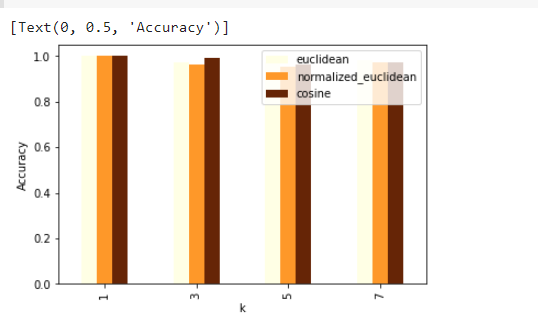


# Plotting a Bar Chart for accuracy

draw = df\_res.plot(x='k', y=['euclidean', 'normalized\_euclidean', 'cosine'], kind="bar", colorm ap='YlOrBr')

draw.set(ylabel='Accuracy')

**Output:-**



Practical-6**(A)**

**Aim:** Implement the different Distance methods (Euclidean) with Prediction, Test Score and Confusion Matrix.

**Description:-**

**Euclidean distance**

Euclidean distance is considered the traditional metric for problems with geometry. It can be simply explained as the ordinary distance between two points. It is one of the most used algorithms in the cluster analysis. One of the algorithms that use this formula would be K-mean. Mathematically it computes the root of squared differences between the coordinates between two objects.

**The confusion matrix**

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing.

**Code:-**

import numpy as np

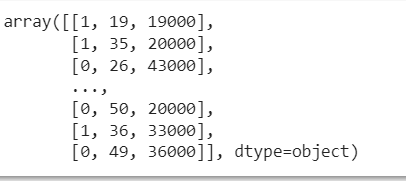
import matplotlib.pyplot as plt import pandas as pd

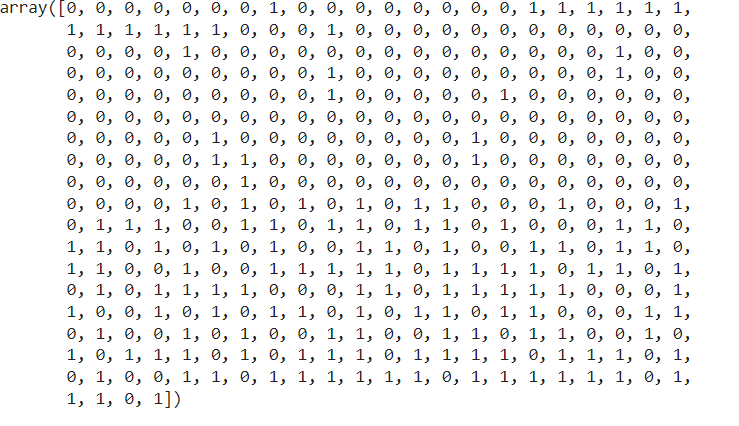
import sklearn

dataset = pd.read\_csv('sample\_data/Social\_Network\_Ads.csv')

X = dataset.iloc[:, [1, 2, 3]].values y = dataset.iloc[:, -1].values

X



y

from sklearn.preprocessing import LabelEncoder le = LabelEncoder()

X[:,0] = le.fit\_transform(X[:,0])

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

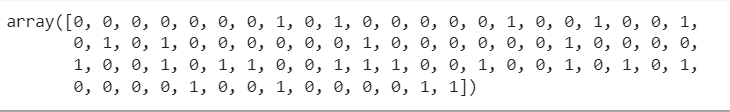
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y,test\_size = 0.20, random\_state = 0) from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

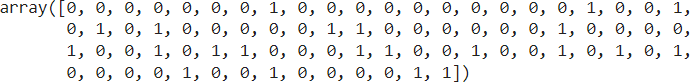
X\_train = sc.fit\_transform(X\_train) X\_test = sc.transform(X\_test)

from sklearn.neighbors import KNeighborsClassifier classifier=KNeighborsClassifier(n\_neighbors=5,metric='minkowski',p = 2) classifier.fit(X\_train, y\_train)

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

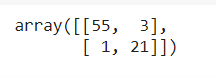


y\_test



from sklearn.metrics import confusion\_matrix,accuracy\_score cm = confusion\_matrix(y\_test, y\_pred)

ac = accuracy\_score(y\_test,y\_pred) print(cm)



Accuracy=

Practical-6**(B)**

**Aim:** Implement the classification model using clustering for the following techniques with K means clustering with Prediction, Test Score and Confusion Matrix.

**Description:-**

**K means clustering:-**

K-Means Clustering is an unsupervised learning algorithm that is used to solve the clustering problems in machine learning or data science. In this topic, we will learn what is K-means clustering algorithm, how the algorithm works, along with the Python implementation of k-means clustering

**The confusion matrix**

The confusion matrix is a matrix used to determine the performance of the classification models for a given set of test data. It can only be determined if the true values for test data are known. The matrix itself can be easily understood, but the related terminologies may be confusing.

**Code:-**

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv) import matplotlib.pyplot as plt # for data visualization

import seaborn as sns # for statistical data visualization

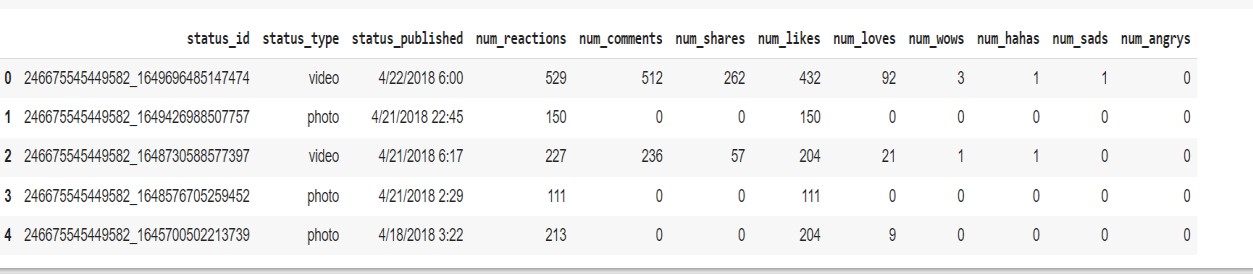
%matplotlib inline

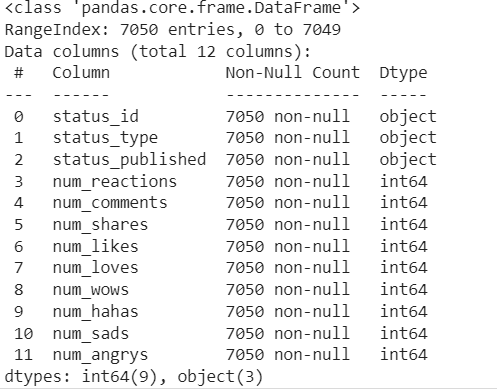
data = '/content/sample\_data/Live.csv' df = pd.read\_csv(data)

df.shape

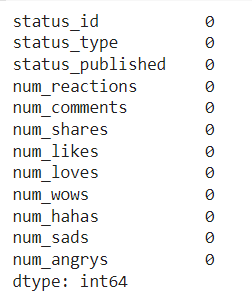


df.head()

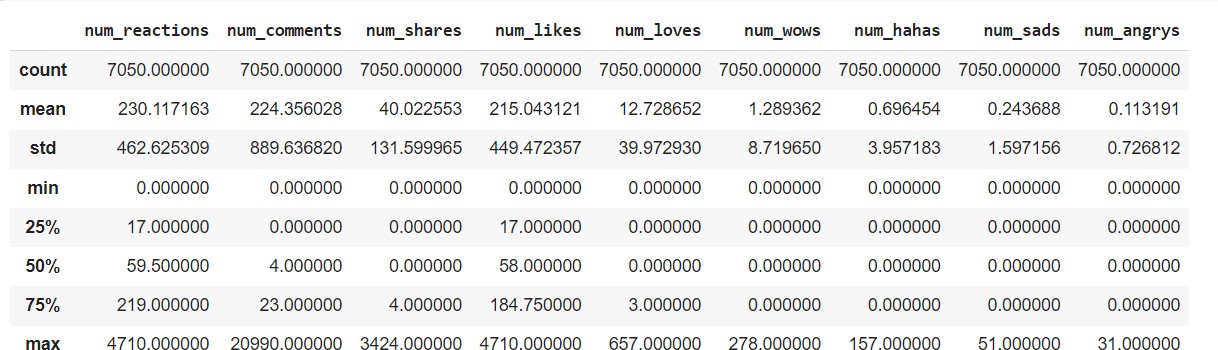


df.info()

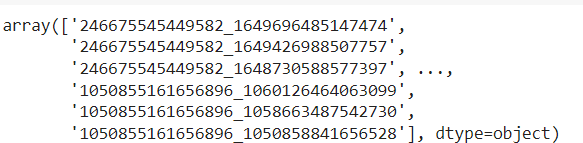
df.isnull().sum()



df.describe()

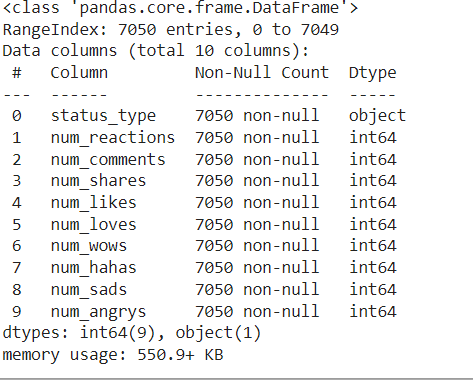


df['status\_id'].unique()

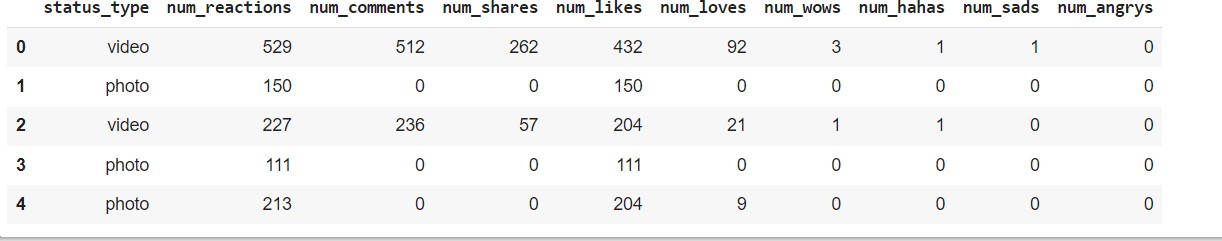


len(df['status\_id'].unique()) df['status\_published'].unique() len(df['status\_published'].unique()) df['status\_type'].unique() len(df['status\_type'].unique())

df.drop(['status\_id', 'status\_published'], axis=1, inplace=True) df.info()



df.head()



X = df

y = df['status\_type']

from sklearn.preprocessing import LabelEncoder le = LabelEncoder()

X['status\_type'] = le.fit\_transform(X['status\_type']) y = le.transform(y)

X.info()

X.head()

cols = X.columns

from sklearn.preprocessing import MinMaxScaler ms = MinMaxScaler()

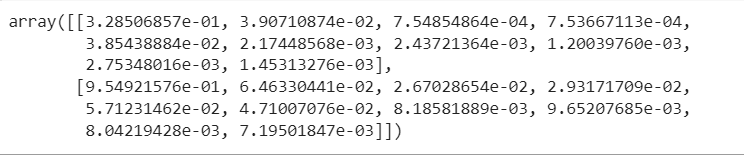
X = ms.fit\_transform(X)

X = pd.DataFrame(X, columns=[cols]) X.head()

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=2, random\_state=0) kmeans.fit(X)

kmeans.cluster\_centers\_



labels = kmeans.labels\_

# check how many of the samples were correctly labeled correct\_labels = sum(y == labels)

print("Result: %d out of %d samples were correctly labeled." % (correct\_labels, y.size))



print('Accuracy score: {0:0.2f}'. format(correct\_labels/float(y.size)))



from sklearn.cluster import KMeans cs = []

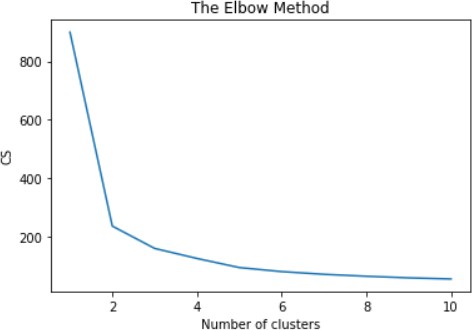
for i in range(1, 11):

kmeans = KMeans(n\_clusters = i, init = 'k-

means++', max\_iter = 300, n\_init = 10, random\_state = 0) kmeans.fit(X)

cs.append(kmeans.inertia\_) plt.plot(range(1, 11), cs) plt.title('The Elbow Method') plt.xlabel('Number of clusters') plt.ylabel('CS')

plt.show()



from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters=4,random\_state=0) kmeans.fit(X)

labels = kmeans.labels\_

# check how many of the samples were correctly labeled correct\_labels = sum(y == labels)

print("Result: %d out of %d samples were correctly labeled." % (correct\_labels, y.size)) print('Accuracy score: {0:0.2f}'. format(correct\_labels/float(y.size)))



Practical-7

**Aim:** Implement the classification model using clustering for the following techniques with hierarchical clustering with Prediction, Test Score and Confusion Matrix

**Description:-**

**Classification model using clustering**

A classification model using clustering involves using a clustering algorithm to group similar data points together, and then using the clusters as labels for a classification model. The clustering algorithm groups the data into clusters based on similarity, and the classification model assigns new data points to one of the clusters based on the features of the data point. This approach can be useful when there is no clear boundary between classes, or when the number of classes is not known in advance.

**Hierarchical clustering**

* Hierarchical clustering is a popular method for grouping objects. It creates groups so that objects within a group are similar to each other and different from objects in other groups.
* Hierarchical clustering is another unsupervised machine learning algorithm, which is used to group the unlabeled datasets into a cluster and also known as hierarchical cluster analysis or HCA.
* In this algorithm, we develop the hierarchy of clusters in the form of a tree, and this tree-shaped structure is known as the dendrogram

**Code:-**

# Importing the libraries import numpy as nm

import matplotlib.pyplot as mtp import pandas as pd

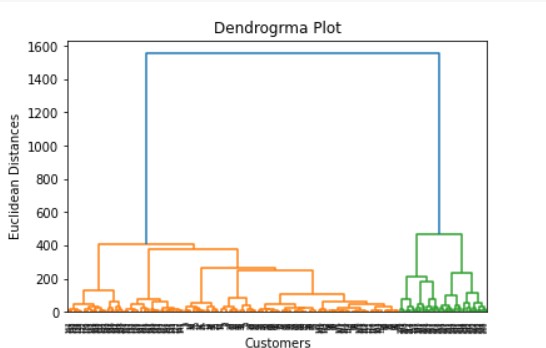
# Importing the dataset

dataset = pd.read\_csv('/content/sample\_data/Mall\_Customers.csv') x = dataset.iloc[:, [3, 4]].values

#Finding the optimal number of clusters using the dendrogram import scipy.cluster.hierarchy as shc

dendro = shc.dendrogram(shc.linkage(x, method="ward")) mtp.title("Dendrogrma Plot")

mtp.ylabel("Euclidean Distances") mtp.xlabel("Customers") mtp.show()



#training the hierarchical model on dataset

from sklearn.cluster import AgglomerativeClustering

hc= AgglomerativeClustering(n\_clusters=5, affinity='euclidean', linkage='ward') y\_pred= hc.fit\_predict(x)

#visulaizing the clusters

mtp.scatter(x[y\_pred == 0, 0], x[y\_pred == 0, 1], s = 100, c = 'purple', label = 'Cluster 1')

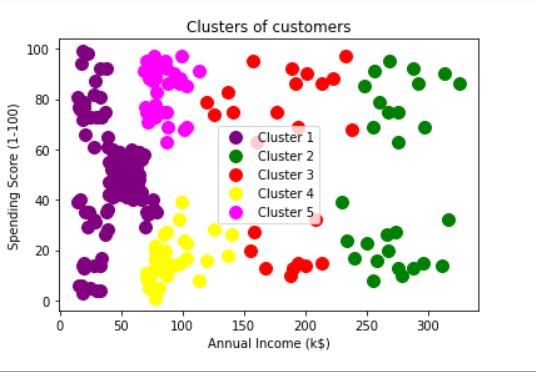
mtp.scatter(x[y\_pred == 1, 0], x[y\_pred == 1, 1], s = 100, c = 'green', label = 'Cluster 2')

mtp.scatter(x[y\_pred== 2, 0], x[y\_pred == 2, 1], s = 100, c = 'red', label = 'Cluster 3')

mtp.scatter(x[y\_pred == 3, 0], x[y\_pred == 3, 1], s = 100, c = 'yellow', label = 'Cluster 4')

mtp.scatter(x[y\_pred == 4, 0], x[y\_pred == 4, 1], s = 100, c = 'magenta', label = 'Cluster 5') mtp.title('Clusters of customers')

mtp.xlabel('Annual Income (k$)') mtp.ylabel('Spending Score (1-100)') mtp.legend()

mtp.show()

Practical-8**(A)**

**Aim:** Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard Heart Disease Data Set.

**Description:-**

**Bayesian networks**

* Bayesian networks are a type of Probabilistic Graphical Model that can be used to build models from data and/or expert opinion.
* They can be used for a wide range of tasks including diagnostics, reasoning, causal modeling, decision making under uncertainty, anomaly detection, automated insight and prediction.

**Bayesian network constructed**

There are two ways to build a Bayesian network: a manual construction or automatic construction (so called "learning") from databases. Both methods have advantages and dis- advantages. Manual construction of a Bayesian network assumes prior expert knowledge of the un- derlying domain.

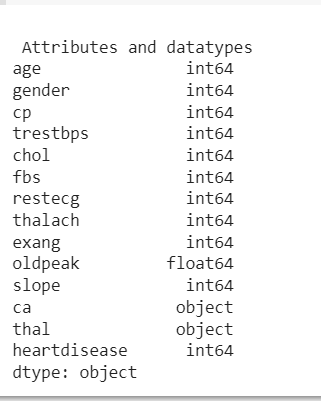
**Code:-**

import numpy as np import pandas as pd import csv

!pip install pgmpy

from pgmpy.estimators import MaximumLikelihoodEstimator from pgmpy.models import BayesianModel

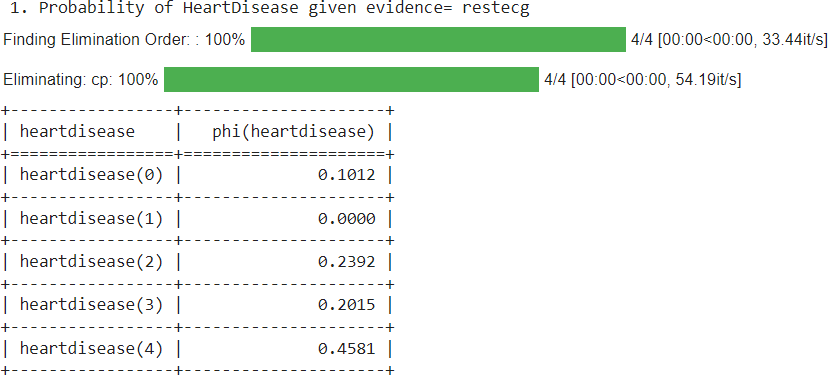
from pgmpy.inference import VariableElimination heartDisease = pd.read\_csv('sample\_data/heart.csv') heartDisease = heartDisease.replace('?',np.nan) print('\n Attributes and datatypes') print(heartDisease.dtypes)



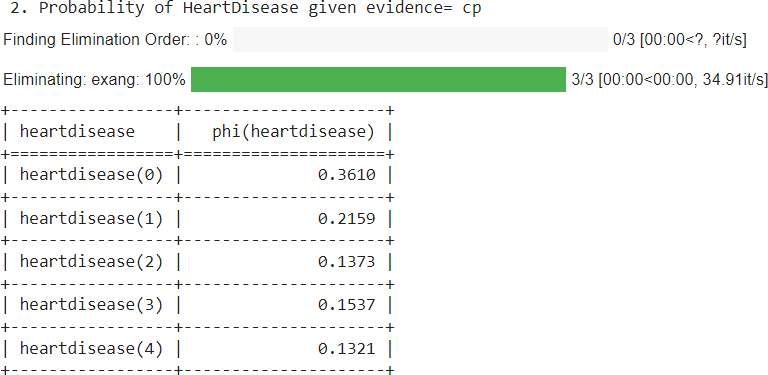
model= BayesianModel([('age','heartdisease'),('gender','heartdisease'),('exang','heartdiseas e'),('cp','heartdisease'),('heartdisease','restecg'),('heartdisease','chol')])

print('\nLearning CPD using Maximum likelihood estimators') model.fit(heartDisease,estimator=MaximumLikelihoodEstimator) print('\n Inferencing with Bayesian Network:') HeartDiseasetest\_infer = VariableElimination(model)

print('\n 1. Probability of HeartDisease given evidence= restecg') q1=HeartDiseasetest\_infer.query(variables=['heartdisease'],evidence={'restecg':1}) print(q1)



print('\n 2. Probability of HeartDisease given evidence= cp ') q2=HeartDiseasetest\_infer.query(variables=['heartdisease'],evidence={'cp':2}) print(q2)



Practical-8**(B)**

**Aim:** Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.

**Description:-**

**non-parametric Locally Weighted Regression algorithm**

Locally weighted linear regression is a non-parametric algorithm, that is, the model does not learn a fixed set of parameters as is done in ordinary linear regression. Rather parameters theta are computed individually for each query point x

**Weighted linear regression**

Weighted linear regression is a generalization of linear regression where the covariance matrix of errors is incorporated in the model. Hence, it can be beneficial when we are dealing with a heteroscedastic data.

**Code:-**

import matplotlib.pyplot as plt import pandas as pd

import numpy as np

def kernel(point, xmat, k): m,n = np.shape(xmat)

weights = np.mat(np.eye((m))) for j in range(m):

diff = point - X[j]

weights[j,j] = np.exp(diff\*diff.T/(-2.0\*k\*\*2)) return weights

def localWeight(point, xmat, ymat, k): wei = kernel(point,xmat,k)

W = (X.T\*(wei\*X)).I\*(X.T\*(wei\*ymat.T)) return W

def localWeightRegression(xmat, ymat, k): m,n = np.shape(xmat)

ypred = np.zeros(m) for i in range(m):

ypred[i] = xmat[i]\*localWeight(xmat[i],xmat,ymat,k) return ypred

# load data points

data = pd.read\_csv('sample\_data/10-dataset.csv') bill = np.array(data.total\_bill)

tip = np.array(data.tip)

#preparing and add 1 in bill mbill = np.mat(bill)

mtip = np.mat(tip)

m= np.shape(mbill)[1] one = np.mat(np.ones(m))

X = np.hstack((one.T,mbill.T))

#set k here

ypred = localWeightRegression(X,mtip,0.5) SortIndex = X[:,1].argsort(0)

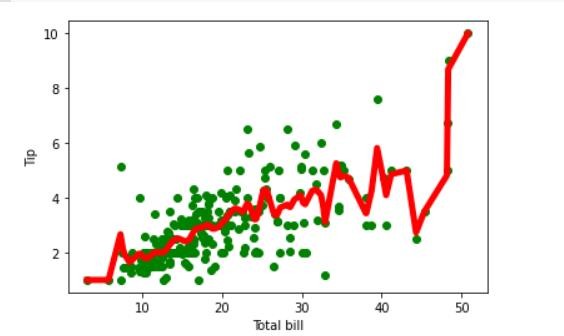
xsort = X[SortIndex][:,0] fig = plt.figure()

ax = fig.add\_subplot(1,1,1) ax.scatter(bill,tip, color='green')

ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5) plt.xlabel('Total bill')

plt.ylabel('Tip') plt.show();

**Output:-**



Practical-9**(A)**

**Aim:** Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

**Description:-**

**Artificial Neural Network**

The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes

**Artificial Neural Network by implementing the Backpropagation algorithm**

Backpropagation is a supervised learning algorithm used to train artificial neural networks. It is based on the concept of gradient descent, which is used to minimize the error between the predicted output of the network and the true output.

**The backpropagation algorithm has two main steps:**

* Forward propagation: In this step, the input is passed through the network, and the output is calculated. The error between the predicted output and the true output is also calculated.
* Backward propagation: In this step, the error is propagated back through the network, and the weights are adjusted in order to minimize the error. The weights are adjusted using the gradient descent algorithm, which updates the weights in the opposite direction of the gradient of the error function

import numpy as np import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split import matplotlib.pyplot as plt

# Load dataset data = load\_iris()

# Get features and target X=data.data y=data.target

# Get dummy variable

y = pd.get\_dummies(y).values y[:3]



#Split data into train and test data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=20, random\_state=4) # Initialize variables

learning\_rate = 0.1

iterations = 5000 N = y\_train.size

# number of input features input\_size = 4

# number of hidden layers neurons hidden\_size = 2

# number of neurons at the output layer output\_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])

# Initialize weights np.random.seed(10)

# initializing weight for the hidden layer

W1 = np.random.normal(scale=0.5, size=(input\_size, hidden\_size))

# initializing weight for the output layer

W2 = np.random.normal(scale=0.5, size=(hidden\_size , output\_size)) def sigmoid(x):

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

def mean\_squared\_error(y\_pred, y\_true):

return ((y\_pred - y\_true)\*\*2).sum() / (2\*y\_pred.size) def accuracy(y\_pred, y\_true):

acc = y\_pred.argmax(axis=1) == y\_true.argmax(axis=1) return acc.mean()

for itr in range(iterations):

# feedforward propagation # on hidden layer

Z1 = np.dot(X\_train, W1) A1 = sigmoid(Z1)

# on output layer Z2 = np.dot(A1, W2) A2 = sigmoid(Z2)

# Calculating error

mse = mean\_squared\_error(A2, y\_train) acc = accuracy(A2, y\_train)

results=results.append({"mse":mse,"accuracy":acc},ignore\_index=True)

# backpropagation E1 = A2 - y\_train

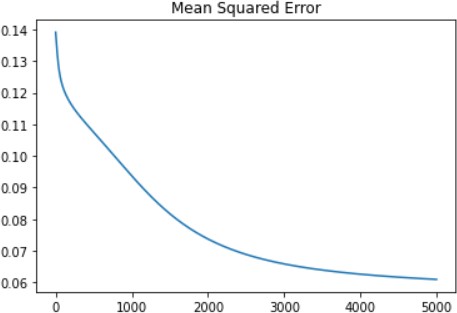
dW1 = E1 \* A2 \* (1 - A2)

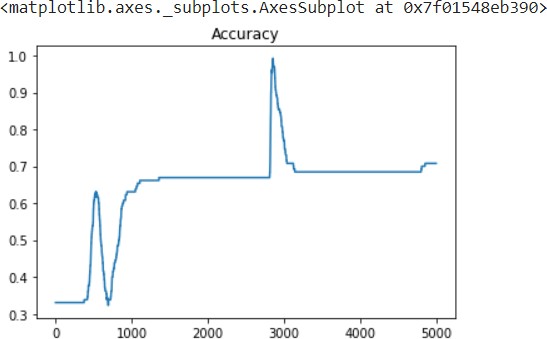
E2 = np.dot(dW1, W2.T) dW2 = E2 \* A1 \* (1 - A1)

# weight updates

W2\_update = np.dot(A1.T, dW1) / N W1\_update = np.dot(X\_train.T, dW2) / N W2 = W2 - learning\_rate \* W2\_update W1 = W1 - learning\_rate \* W1\_update

results.mse.plot(title="Mean Squared Error")



results.accuracy.plot(title="Accuracy")

# feedforward

Z1 = np.dot(X\_test, W1) A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2) A2 = sigmoid(Z2)

acc = accuracy(A2, y\_test) print("Accuracy: {}".format(acc))



Practical-9**(B)**

**Aim:** Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task.

**Description:-**

**The naïve Bayesian Classifier model**

* A Naive Bayes classifier is a probabilistic machine learning model that is based on the Bayes' theorem. It is called "naive" because it makes the assumption that all features are independent of each other, given the class label.
* The Naive Bayes classifier can be used for both binary and multiclass classification problems. There are different types of Naive Bayes classifiers such as Gaussian Naive Bayes, Multinomial Naive Bayes, and Bernoulli Naive Bayes.
* To implement a Naive Bayes classifier in Python, you can use the scikit-learn library, which provides several implementations of Naive Bayes classifiers.

**Code:-**

import pandas as pd

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

from sklearn.feature\_extraction.text import CountVectorizer from sklearn.naive\_bayes import MultinomialNB

from sklearn import metrics msg=pd.read\_csv('sample\_data/data.csv',names=['message','label']) print('The dimensions of the dataset',msg.shape) msg['labelnum']=msg.label.map({'pos':1,'neg':0})

X=msg.message y=msg.labelnum



xtrain,xtest,ytrain,ytest=train\_test\_split(X,y)

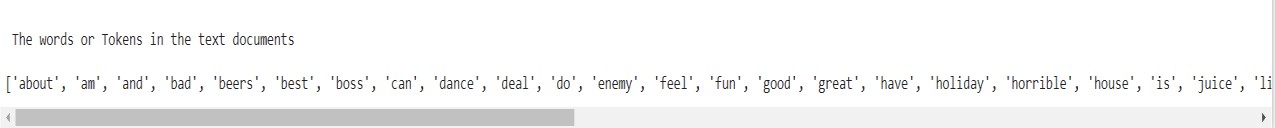
print ('\n the total number of Training Data :',ytrain.shape) print ('\n the total number of Test Data :',ytest.shape)



cv = CountVectorizer()

xtrain\_dtm = cv.fit\_transform(xtrain) xtest\_dtm=cv.transform(xtest)

print('\n The words or Tokens in the text documents \n') print(cv.get\_feature\_names()) df=pd.DataFrame(xtrain\_dtm.toarray(),columns=cv.get\_feature\_names())



clf = MultinomialNB().fit(xtrain\_dtm,ytrain) predicted = clf.predict(xtest\_dtm)

#printing accuracy, Confusion matrix, Precision and Recall

print('\n Accuracy of the classifier is',metrics.accuracy\_score(ytest,predicted)) print('\n Confusion matrix')

print(metrics.confusion\_matrix(ytest,predicted))

print('\n The value of Precision',metrics.precision\_score(ytest,predicted)) print('\n The value of Recall', metrics.recall\_score(ytest,predicted))

**Output:-**

