1st

**import** numpy **as** np

x **=** np**.**array([['Sunny','Warm','Normal','Strong','Warm','Same','Yes'],

['Sunny','Warm','High','Strong','Warm','Same','Yes'],

['Rainy','Cold','High','Strong','Warm','Change','No'],

['Sunny','Warm','High','Strong','Cool','Change','Yes']])

r,c **=** x**.**shape

print("The number of rows and columns are: ", r,"and", c**-**1)

print("The dataset is: \n",x)

s **=** []

s **=** np**.**empty(c**-**1, dtype**=**object)

**for** i **in** range(c**-**1):

s[i] **=** "\u03A6"

print("\n Initial specific hypotheis: \n",s)

**for** i **in** range(c**-**1):

s[i] **=** x[0,i]

print("after processing first sample, specific hypotheis")

print(s)

**for** i **in** range(1,r):

**if** x[i,c**-**1] **==** 'No':

**continue**

**else**:

**for** j **in** range(c**-**1):

**if** x[i,j] **!=** s[j]:

s[j] **=** '?'

print("\nIntermediate hypotheis: \n",s)

print("\nFinal hypotheis: \n",s)

print("enter new instance value")

o/p:-

The number of rows and columns are: 4 and 6

The dataset is:

[['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same' 'Yes']

['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same' 'Yes']

['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change' 'No']

['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change' 'Yes']]

Initial specific hypotheis:

['Φ' 'Φ' 'Φ' 'Φ' 'Φ' 'Φ']

after processing first sample, specific hypotheis

['Sunny' 'Warm' 'Normal' 'Strong' 'Warm' 'Same']

Intermediate hypotheis:

['Sunny' 'Warm' '?' 'Strong' 'Warm' 'Same']

Intermediate hypotheis:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final hypotheis:

['Sunny' 'Warm' '?' 'Strong' '?' '?']

enter new instance value

**2nd**

**import** numpy **as** np

x **=** np**.**array([['sunny','warm','normal','strong','warm','same','yes'],

['sunny','warm','high','strong','warm','same','yes'],

['rainy','cold','high','strong','warm','change','no'],

['sunny','warm','high','strong','cool','change','yes']])

r,c**=**x**.**shape

print("The number of rows and columns are: ", r,"and", c**-**1)

print("The dataset is: \n",x)

s**=**np**.**empty(c**-**1, dtype**=**object)

**for** i **in** range(c**-**1): s[i] **=** "\u03A6"

print("\n Most specific hypothies:\n",s)

g**=**np**.**empty(c**-**1,dtype**=**object)

**for** i **in** range(c**-**1):g[i]**=**"?"

print("\n Most general hypothies:\n",g)

**for** i **in** range(c**-**1):s[i]**=**x[0,i]

Flag**=**0

**for** i **in** range(1,r):

*#print("\n specific hypothies:\n",s)*

**if** x[i,c**-**1]**==**'yes':

**for** j **in** range(c**-**1):

**if** x[i,j]**!=**s[j]:

s[j]**=**'?'

print("\nIntermediate specific hypothies:\n",s)

**if** Flag**==**0:

Flag**=**1

**continue**

**else**:

r1**=**np**.**shape(g)[0]

**for** i1 **in** range(r1):

**for** j1 **in** range(c**-**1):

**if** g[i1,j1]**!=**x[i,j1] **and** g[i1,j1]**!=**"?":

g**=**np**.**delete(g,i1,axis**=**0)

**else**:

**for** j **in** range(c**-**1):

g1**=**np**.**empty(c**-**1,dtype**=**object)

**if** s[j]**==**'?':

**continue**

**elif** x[i,j]**!=**s[j]:

**for** l **in** range(c**-**1):g1[1]**=**"?"

g1[j]**=**s[i]

g**=**np**.**vstack([g,g1])

g**=**np**.**delete(g,(0),axis**=**0)

print("\nIntermediate general hypotheis\n",g)

print("\n Specific hypotheis:\n",s)

print("\n General hypotheis:\n",g)

print("Enter new instance value")

new**=**[]

new**=**np**.**empty(c**-**1,dtype**=**object)

nr,nc**=**g**.**shape

print("ENTER ATTRIBUTE VALUES for SKY,AIRTEMP,HUMIDITY,WIND,WATER, and FORECAST")

**for** i **in** range(c**-**1):

new[i]**=**input("enter attribute values")

Flag**=**1

**for** i **in** range(c**-**1):

**if** s[i]**!=**'?' **and** g[i][j]**!=**new[j]:

Flag1**=**0

**break**

**if** Flag1**==**1:

**break**

print("\n The assigned label for ENJOYING WATER SPORT is")

**if** Flag**==**1 **or** Flag1**==**1:

print("YES")

**else**:

print("NO")

The number of rows and columns are: 4 and 6

The dataset is:

[['sunny' 'warm' 'normal' 'strong' 'warm' 'same' 'yes']

['sunny' 'warm' 'high' 'strong' 'warm' 'same' 'yes']

['rainy' 'cold' 'high' 'strong' 'warm' 'change' 'no']

['sunny' 'warm' 'high' 'strong' 'cool' 'change' 'yes']]

Most specific hypothies:

['Φ' 'Φ' 'Φ' 'Φ' 'Φ' 'Φ']

Most general hypothies:

['?' '?' '?' '?' '?' '?']

Intermediate specific hypothies:

['sunny' 'warm' '?' 'strong' 'warm' 'same']

Intermediate general hypotheis

[['?' '?' None None None None]

[None '?' None None None None]

[None '?' None None None '?']]

Intermediate specific hypothies:

['sunny' 'warm' '?' 'strong' '?' '?']

**---------------------------------------------------------------------------**

**IndexError** Traceback (most recent call last)

Input **In [2]**, in <cell line: 17>**()**

29 **for** i1 **in** range(r1):

30 **for** j1 **in** range(c-1):

**---> 31** **if** g[i1,j1]!=x[i,j1] **and** g[i1,j1]!="?":

32 g=np.delete(g,i1,axis=0)

33 **else**:

**IndexError**: index 0 is out of bounds for axis 0 with size 0

In [ ]:

3rd-

**import** pandas **as** pd

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

**from** sklearn.metrics **import** accuracy\_score

**from** sklearn.naive\_bayes **import** GaussianNB

data**=**pd**.**read\_csv("C:\\Users\\mdtah\\Desktop\\rvr moodle\\4th Year\\ML Lab\\IT\\play\_tennis.csv")

print("the given data set is\n")

print(data)

predictors**=**data**.**iloc[:,0:4]

target**=**data**.**iloc[:,4]

predictors\_train,predictors\_test,target\_train,target\_test**=**train\_test\_split(predictors,target,test\_size**=**0.2,random\_state**=**123)

gnb**=**GaussianNB()

model**=**gnb**.**fit(predictors\_train,target\_train)

prediction**=**model**.**predict(predictors\_test)

print("Accuracy of classifier")

accuracy\_score(target\_test,prediction,normalize**=True**)

the given data set is

Outlook { Sunny=1,rain=0,overcast=0.5} \

0 1.0

1 1.0

2 0.5

3 0.0

4 0.0

.. ...

65 0.0

66 1.0

67 0.5

68 0.5

69 0.0

Temperature{hot=1,cool=0,mild=0.5} Humidity{Normal=1,High=0} \

0 0.0 0

1 0.0 0

2 0.0 0

3 0.5 0

4 1.0 1

.. ... ...

65 0.5 1

66 0.5 1

67 0.5 0

68 1.0 1

69 0.5 0

Wind{Strong=1,Weak=0} PlayTennis{Yes=1,No=0}

0 0 0

1 1 0

2 0 1

3 0 1

4 0 1

.. ... ...

65 0 1

66 1 1

67 1 1

68 0 1

69 1 0

[70 rows x 5 columns]

Accuracy of classifier

Out[3]:

0.7857142857142857

In [ ]:

4th

*#To read training data*

**import** pandas **as** pd

**import** numpy **as** np

**from** pprint **import** pprint

*#Import the dataset and define the feature as well as the target datasets / columns#*

dataset **=** pd**.**read\_csv('C:\\Users\\mdtah\\Desktop\\rvr moodle\\4th Year\\ML Lab\\IT\\play\_tennisDT1.csv', names**=**['Outlook','Temperature','humidity','Wind','PlayTennis'])

dataset

*#Construction of decision tree*

**def** entropy(target\_col):

elements,counts **=** np**.**unique(target\_col,return\_counts **=** **True**)

entropy **=** np**.**sum([(**-**counts[i]**/**np**.**sum(counts))**\***np**.**log2(counts[i]**/**np**.**sum(counts)) **for** i **in** range(len(elements))])

**return** entropy

**def** InfoGain(data,split\_attribute\_name,target\_name**=**"PlayTennis"):

*#Calculate the entropy of the total dataset*

total\_entropy **=** entropy(data[target\_name])

*##Calculate the entropy of the dataset*

*#Calculate the values and the corresponding counts for the split attribute*

vals,counts**=** np**.**unique(data[split\_attribute\_name],return\_counts**=True**)

*#Calculate the weighted entropy*

Weighted\_Entropy **=** np**.**sum([(counts[i]**/**np**.**sum(counts))**\***entropy(data**.**where(data[split\_attribute\_name]**==**vals[i])**.**dropna()[target\_name]) **for** i **in** range(len(vals))])

*#Calculate the information gain*

Information\_Gain **=** total\_entropy **-** Weighted\_Entropy

print(split\_attribute\_name,vals,counts,Information\_Gain)

**return** Information\_Gain

**def** ID3(data,originaldata,features,target\_attribute\_name**=**"PlayTennis",parent\_node\_class **=** **None**):

*#Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#*

*#If all target\_values have the same value, return this value*

**if** len(np**.**unique(data[target\_attribute\_name])) **<=** 1:

**return** np**.**unique(data[target\_attribute\_name])[0]

*#If the dataset is empty, return the mode target feature value in the original dataset*

**elif** len(data)**==**0:

**return** np**.**unique(originaldata[target\_attribute\_name])[np**.**argmax(np**.**unique(originaldata[target\_attribute\_name],return\_counts**=True**)[1])]

*#If the feature space is empty, return the mode target feature value of the direct parent node --> Note that*

*#the direct parent node is that node which has called the current run of the ID3 algorithm and hence*

*#the mode target feature value is stored in the parent\_node\_class variable.*

**elif** len(features) **==**0:

**return** parent\_node\_class

*#If none of the above holds true, grow the tree!*

**else**:

*#Set the default value for this node --> The mode target feature value of the current node*

parent\_node\_class **=** np**.**unique(data[target\_attribute\_name])[np**.**argmax(np**.**unique(data[target\_attribute\_name],return\_counts**=True**)[1])]

*#Select the feature which best splits the dataset*

item\_values **=** [InfoGain(data,feature,target\_attribute\_name) **for** feature **in** features] *#Return the information gain values for the features in the dataset*

best\_feature\_index **=** np**.**argmax(item\_values)

best\_feature **=** features[best\_feature\_index]

*#Create the tree structure. The root gets the name of the feature (best\_feature) with the maximum information*

*#gain in the first run*

tree **=** {best\_feature:{}}

*#Remove the feature with the best inforamtion gain from the feature space*

features **=** [i **for** i **in** features **if** i **!=** best\_feature]

*#Grow a branch under the root node for each possible value of the root node feature*

**for** value **in** np**.**unique(data[best\_feature]):

value **=** value

*#Split the dataset along the value of the feature with the largest information gain and therwith create sub\_datasets*

sub\_data **=** data**.**where(data[best\_feature] **==** value)**.**dropna()

*#Call the ID3 algorithm for each of those sub\_datasets with the new parameters --> Here the recursion comes in!*

subtree **=** ID3(sub\_data,dataset,features,target\_attribute\_name,parent\_node\_class)

print("\n ")

*#Add the sub tree, grown from the sub\_dataset to the tree under the root node*

tree[best\_feature][value] **=** subtree

**return**(tree)

**def** predict(query,tree,default **=** 1):

**for** key **in** list(query**.**keys()):

**if** key **in** list(tree**.**keys()):

**try**:

result **=** tree[key][query[key]]

**except**:

**return** default

result **=** tree[key][query[key]]

**if** isinstance(result,dict):

**return** predict(query,result)

**else**:

**return** result

**def** train\_test\_split(dataset):

training\_data **=** dataset**.**iloc[:14]**.**reset\_index(drop**=True**)*#We drop the index respectively relabel the index*

*#starting form 0, because we do not want to run into errors regarding the row labels / indexes*

testing\_data **=** dataset**.**iloc[14:]**.**reset\_index(drop**=True**)

**return** training\_data,testing\_data

training\_data **=** train\_test\_split(dataset)[0]

testing\_data **=** train\_test\_split(dataset)[1]

**def** test(data,tree):

*#Create new query instances by simply removing the target feature column from the original dataset and*

*#convert it to a dictionary*

queries **=** data**.**iloc[:,:**-**1]**.**to\_dict(orient **=** "records")

*#Create a empty DataFrame in whose columns the prediction of the tree are stored*

predicted **=** pd**.**DataFrame(columns**=**["predicted"])

*#Calculate the prediction accuracy*

**for** i **in** range(len(data)):

predicted**.**loc[i,"predicted"] **=** predict(queries[i],tree,1.0)

print('\n The prediction accuracy is: ',(np**.**sum(predicted["predicted"] **==** data["PlayTennis"])**/**len(data))**\***100,'%')

tree **=** ID3(training\_data,training\_data,training\_data**.**columns[:**-**1])

print("\n\n\nThe final Resultant Decision Tree")

pprint(tree)

test(testing\_data,tree)

Outlook ['Overcast' 'Rain' 'Sunny'] [4 5 5] 0.24674981977443933

Temperature ['Cool' 'Hot' 'Mild'] [4 4 6] 0.02922256565895487

humidity ['High' 'Normal'] [7 7] 0.15183550136234159

Wind ['Strong' 'Weak'] [6 8] 0.04812703040826949

Temperature ['Cool' 'Mild'] [2 3] 0.01997309402197489

humidity ['High' 'Normal'] [2 3] 0.01997309402197489

Wind ['Strong' 'Weak'] [2 3] 0.9709505944546686

Temperature ['Cool' 'Hot' 'Mild'] [1 2 2] 0.5709505944546686

humidity ['High' 'Normal'] [3 2] 0.9709505944546686

Wind ['Strong' 'Weak'] [2 3] 0.01997309402197489

The final Resultant Decision Tree

{'Outlook': {'Overcast': 'Yes',

'Rain': {'Wind': {'Strong': 'No', 'Weak': 'Yes'}},

'Sunny': {'humidity': {'High': 'No', 'Normal': 'Yes'}}}}

The prediction accuracy is: 88.88888888888889 %

In [ ]:

5th

# K-Nearest neighbor implementation

import math

# calculate the Euclidean distance between two vectors

def euclidean\_distance(row1, row2):

distance = 0.0

for i in range(n1-1):

distance += (row1[i] - row2[i])\*\*2

return math.sqrt(distance)

# Locate the most similar neighbors

def get\_neighbors(train, test\_row, num\_neighbors):

distances = list()

for train\_row in train:

dist = euclidean\_distance(test\_row, train\_row)

distances.append((train\_row, dist))

print('distances from neighbors to given data point')

print(distances)

distances.sort(key=lambda tup: tup[1])

neighbors = list()

for i in range(num\_neighbors):

neighbors.append(distances[i][0])

return neighbors

# Make a classification prediction with neighbors

def predict\_classification(train, test\_row, num\_neighbors):

neighbors = get\_neighbors(train, test\_row, num\_neighbors)

print('Neighbors of given data point are\n')

print(neighbors)

output\_values = [row[-1] for row in neighbors]

prediction = max(set(output\_values), key=output\_values.count)

return prediction

dataset=[];

n=int(input('enter no of data points'));

n1=int(input('enter no of dimensions in a dataset along with class label attribute'));

print('attr1,aatr2,.....attrn(class label)')

for i in range(n):

print('enter attribute values for point',i);

dataset.append([]);

counter=0;

while counter<n1:

b=float(input('enter value for each dimension'));

counter=counter+1;

dataset[i].append(b);

print('Given points in data set are')

print(dataset);

print('Enter a new data point to assign a label')

p=[];

i=0

n=n1-1

while i<n:

k=float(input('enter data point elements'))

p.append(k)

i=i+1

print('Enter K no.of neighbors')

k=int(input())

prediction = predict\_classification(dataset,p, k)

print('The assigned label for the given data point is is',prediction)

---------------------------------------------------------------------------------------------------------

output

enter no of data points4

enter no of dimensions in a dataset along with class label attribute2

attr1,aatr2,.....attrn(class label)

enter attribute values for point 0

enter value for each dimension5.2

enter value for each dimension1.0

enter attribute values for point 1

enter value for each dimension5.0

enter value for each dimension1.0

enter attribute values for point 2

enter value for each dimension4.0

enter value for each dimension0

enter attribute values for point 3

enter value for each dimension4.8

enter value for each dimension0

Given points in data set are

[[5.2, 1.0], [5.0, 1.0], [4.0, 0.0], [4.8, 0.0]]

Enter a new data point to assign a label

enter data point elements4.9

Enter K no.of neighbors

3

distances from neighbors to given data point

[([5.2, 1.0], 0.2999999999999998), ([5.0, 1.0], 0.09999999999999964), ([4.0, 0.0], 0.9000000000000004), ([4.8, 0.0], 0.10000000000000053)]

Neighbors of given data point are

[[5.0, 1.0], [4.8, 0.0], [5.2, 1.0]]

The assigned label for the given data point is is 1.0

6th

import math

import sys

m=int(input("no.of data points: "))

n=int(input("no.of dimensions per a data point: "))

#to read data points and centroid

l=[]

for i in range(m):

k=input("enter a data point with "+str(n)+" dimensions: ").split()

k=[int(i) for i in k]

if(len(k)==n):

l=l+[k]

else :

print("Data point with incorrect dimensions...!")

sys.exit()

print('\n\nThe given data points are',l)

c=int(input("enter no.of clusters: "))

g=[]

for i in range(c):

g=g+[l[i]]

print("\nInitialized centeriod matrix:")

print(g)

#calcute first distance and group matrix

dist = [[0] \* c for i in range(m)]

print('\nintial distance matrix',dist)

for i in range(m):

for j in range(c):

t=0

for k in range(n):

t=t+(l[i][k]-g[j][k])\*(l[i][k]-g[j][k])

t=math.sqrt(t)

dist[i][j]=t

print("\nDistance matrix after 1st iteration: ")

print(dist)

g = [[0] \* n for i in range(c)]

grp = [[0] \* c for i in range(m)]

grp1 = [[0] \* c for i in range(m)]

t=[0 for i in range(c)]

for i in range(m):

for j in range(c):

if(dist[i][j]==min(dist[i])):

grp[i][j]= j+1

for k in range(n):

g[j][k]+=l[i][k]

t[j]+=1

else:

grp[i][j]=0

for j in range(c):

for k in range(n):

g[j][k]/=t[j]

print("\nCluster Membership matrix after 1st iteration: ")

print(grp)

print("\nupdated centeriod matrix after 1st iteration:")

print(g)

#for next iterations

while(grp1!=grp) :

for i in range(m):

for j in range(c):

t=0

for k in range(n):

t=t+(l[i][k]-g[j][k])\*(l[i][k]-g[j][k])

t=math.sqrt(t)

dist[i][j]=t

print("\nDistance matrix in next iteration: ")

print(dist)

g = [[0] \* n for i in range(c)]

grp1=grp

grp = [[0] \* c for i in range(m)]

t=[0 for i in range(c)]

for i in range(m):

for j in range(c):

if(dist[i][j]==min(dist[i])):

grp[i][j]= j+1

for k in range(n):

g[j][k]+=l[i][k]

t[j]+=1

else:

grp[i][j]=0

for j in range(c):

for k in range(n):

g[j][k]/=t[j]

print("\nCluster Membership after next iteration: ")

print(grp)

print("\n\n\nUpdated centroid matrix after next iteration:")

print(g)

7th

**from** random **import** seed

**from** random **import** random

**import** math

*# Initialize a network*

**def** initialize\_network(n\_inputs, n\_hidden, n\_outputs):

network **=** list()

hidden\_layer **=** [{'weights':[random() **for** i **in** range(n\_inputs**+**1)]} **for** i **in** range(n\_hidden)]

network**.**append(hidden\_layer)

output\_layer **=** [{'weights':[random() **for** i **in** range(n\_hidden**+**1)]} **for** i **in** range(n\_outputs)]

network**.**append(output\_layer)

**return** network

*# Calculate neuron activation for an input*

**def** activate(weights, inputs):

activation **=** weights[**-**1]

**for** i **in** range(len(weights)**-**1):

activation **+=** weights[i] **\*** inputs[i]

**return** activation

*# Transfer neuron activation*

**def** transfer(activation):

**return** 1.0 **/** (1.0 **+** math**.**exp(**-**activation))

*# Forward propagate input to a network output*

**def** forward\_propagate(network, row):

inputs **=** row

**for** layer **in** network:

new\_inputs **=** []

**for** neuron **in** layer:

activation **=** activate(neuron['weights'], inputs)

neuron['output'] **=** transfer(activation)

new\_inputs**.**append(neuron['output'])

inputs **=** new\_inputs

**return** inputs

*# Calculate the derivative of an neuron output*

**def** transfer\_derivative(output):

**return** output **\*** (1.0 **-** output)

*# Backpropagate error and store in neurons*

**def** backward\_propagate\_error(network, expected):

**for** i **in** reversed(range(len(network))):

layer **=** network[i]

errors **=** list()

**if** i **!=** len(network)**-**1:

**for** j **in** range(len(layer)):

error **=** 0.0

**for** neuron **in** network[i **+** 1]:

error **+=** (neuron['weights'][j] **\*** neuron['delta'])

errors**.**append(error)

**else**:

**for** j **in** range(len(layer)):

neuron **=** layer[j]

errors**.**append(neuron['output'] **-** expected[j])

**for** j **in** range(len(layer)):

neuron **=** layer[j]

neuron['delta'] **=** errors[j] **\*** transfer\_derivative(neuron['output'])

*# Update network weights with error*

**def** update\_weights(network, row, l\_rate):

**for** i **in** range(len(network)):

inputs **=** row[:**-**1]

**if** i **!=** 0:

inputs **=** [neuron['output'] **for** neuron **in** network[i **-** 1]]

**for** neuron **in** network[i]:

**for** j **in** range(len(inputs)):

neuron['weights'][j] **-=** l\_rate **\*** neuron['delta'] **\*** inputs[j]

neuron['weights'][**-**1] **-=** l\_rate **\*** neuron['delta']

*# Train a network for a fixed number of epochs*

**def** train\_network(network, train, l\_rate, n\_epoch, n\_outputs):

**for** epoch **in** range(n\_epoch):

sum\_error **=** 0

**for** row **in** train:

outputs **=** forward\_propagate(network, row)

expected **=** [0 **for** i **in** range(n\_outputs)]

expected[row[**-**1]] **=** 1

sum\_error **+=** sum([(expected[i]**-**outputs[i])**\*\***2 **for** i **in** range(len(expected))])

backward\_propagate\_error(network, expected)

update\_weights(network, row, l\_rate)

print('\nAt Iteration=%d, lrate=%.3f, error=%.3f' **%** (epoch, l\_rate, sum\_error))

*# Training backprop algorithm*

seed(1)

print('\n Details of Intialized Neural Network Structure\n')

*#dataset=[[1,0,0,0,0],[0,0.5,0,0,1]]*

dataset**=**[[1,0,0,0,0], [1,0,0,1,0], [0.5,0,0,0,1], [0,0.5,0,0,1], [0,1,1,0,1], [0,1,1,1,0], [0.5,1,1,1,1], [1,0.5,0,0,0], [1,0.5,1,0,1], [0,0.5,1,0,1], [1,0.5,1,1,1], [0.5,0.5,0,1,1], [0.5,1,1,0,1]]

*#dataset = [[2.7810836,2.550537003,0],[1.465489372,2.362125076,0],[8.675418651,-0.242068655,1],[7.673756466,3.508563011,1]]*

n\_inputs **=** len(dataset[0])**-**1

print('Neurons at input layer',n\_inputs)

n\_hidden**=**2

print('Neurons at hidden layer',n\_hidden)

n\_outputs **=** len(set([row[**-**1] **for** row **in** dataset]))

print('Neurons at output layer',n\_outputs)

print('\n')

network **=** initialize\_network(n\_inputs, n\_hidden, n\_outputs)

lrate**=**0.5

n\_iter**=**10

print('\n Intialized weights')

**for** layer **in** network:

print(layer)

print('\n')

print('learning rate and error values at each iteration')

train\_network(network, dataset, lrate,n\_iter, n\_outputs)

print('\n')

print('\n Results of updated Weights,Outputs, delta values at each node of layer in ANN after training \n')

**for** layer **in** network:

print(layer)

Details of Intialized Neural Network Structure

Neurons at input layer 4

Neurons at hidden layer 2

Neurons at output layer 2

Intialized weights

[{'weights': [0.13436424411240122, 0.8474337369372327, 0.763774618976614, 0.2550690257394217, 0.49543508709194095]}, {'weights': [0.4494910647887381, 0.651592972722763, 0.7887233511355132, 0.0938595867742349, 0.02834747652200631]}]

[{'weights': [0.8357651039198697, 0.43276706790505337, 0.762280082457942]}, {'weights': [0.0021060533511106927, 0.4453871940548014, 0.7215400323407826]}]

learning rate and error values at each iteration

At Iteration=0, lrate=0.500, error=8.575

At Iteration=1, lrate=0.500, error=6.436

At Iteration=2, lrate=0.500, error=5.784

At Iteration=3, lrate=0.500, error=5.703

At Iteration=4, lrate=0.500, error=5.691

At Iteration=5, lrate=0.500, error=5.685

At Iteration=6, lrate=0.500, error=5.678

At Iteration=7, lrate=0.500, error=5.669

At Iteration=8, lrate=0.500, error=5.658

At Iteration=9, lrate=0.500, error=5.646

Results of updated Weights,Outputs, delta values at each node of layer in ANN after training

[{'weights': [0.06423373997572304, 0.8153245233660625, 0.7341160078737408, 0.22833442684909458, 0.38764550202770776], 'output': 0.8772168279416795, 'delta': -0.0017653379463034454}, {'weights': [0.1894606604585811, 0.7108270463144478, 0.9122284806512235, -0.00726753113079521, -0.08997409296386821], 'output': 0.8342656435135846, 'delta': -0.007166855154026416}]

[{'weights': [-0.2879183910479843, -0.5960791973137634, -0.4098666387950039], 'output': 0.2493288536117717, 'delta': 0.04666537966535675}, {'weights': [0.10320341931547322, 0.5451053949656808, 0.6020468745474502], 'output': 0.7481647214661503, 'delta': -0.04744936042198027}]

In [ ]:

8th

*#Linear Regression implementation*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**def** estimate\_coef(x, y):

*# number of observations/points*

n **=** np**.**size(x)

*# mean of x and y*

mean\_x **=** np**.**mean(x)

mean\_y **=** np**.**mean(y)

print('\n mean of salary is',mean\_x)

print('\n mean of loan amount is',mean\_y)

print('\n')

cov**=**0

**for** i **in** range(n):

x\_d**=**x[i]**-**mean\_x

y\_d**=**y[i]**-**mean\_y

xy**=**x\_d **\*** y\_d

cov**=**cov**+**xy

var**=**0

**for** i **in** range(n):

x\_d**=**x[i]**-**mean\_x

xx**=** x\_d **\*** x\_d

var**=**var**+**xx

*# calculating regression coefficients*

b1 **=** cov **/** var

b0 **=** mean\_y **-** b1 **\*** mean\_x

**return** (b0,b1)

**def** plot\_regression\_line(x, y, b):

*# plotting the actual points as scatter plot*

plt**.**scatter(x, y, color **=** "m",

marker **=** "o", s **=** 30)

*# predicted response vector*

y\_pred **=** b[0] **+** b[1] **\*** x

*# plotting the regression line*

plt**.**plot(x, y\_pred, color **=** "g")

*# putting labels*

plt**.**xlabel('Salary in lakhs')

plt**.**ylabel('Loan amount in Lakhs')

*# function to show plot*

plt**.**show()

*#Import the dataset and define the features*

dataframe **=** pd**.**read\_csv('C:\\Users\\mdtah\\Desktop\\ML prg\\8th\\reg-dataset.csv', names**=**['salary','loan\_amount\_sanctioned'])

print(dataframe)

print('\n')

sal **=** dataframe**.**iloc[:,0]

print('The values extraced for SALARY column in lakhs from dataframe')

print(sal)

print('\n')

loanamount **=** dataframe**.**iloc[:,1]

print('The values extraced for LOAN AMOUNT in Lakhs column from dataframe')

print(loanamount)

print('\n')

b **=** estimate\_coef(sal, loanamount)

print("Estimated coefficients:\n b0 = {} \

\n b1 = {}"**.**format(b[0], b[1]))

print('\n')

print('plot of a straight line')

*# plotting regression line*

plot\_regression\_line(sal, loanamount, b)

print('\n')

print('Enter the salary of a person to predict the loan amount to be sanctioned')

salary1**=**int(input())

predicted\_loan\_amount **=** b[0] **+** b[1] **\*** salary1

print('loan amount predicted for the entered salary is by Linear Regression: ')

print(predicted\_loan\_amount)

salary loan\_amount\_sanctioned

0 2 5

1 8 10

2 7 11

3 1 3

4 2 4

5 4 6

6 4 5

7 8 10

8 9 12

9 3 5

10 4 7

11 8 11

12 9 11

13 6 9

The values extraced for SALARY column in lakhs from dataframe

0 2

1 8

2 7

3 1

4 2

5 4

6 4

7 8

8 9

9 3

10 4

11 8

12 9

13 6

Name: salary, dtype: int64

The values extraced for LOAN AMOUNT in Lakhs column from dataframe

0 5

1 10

2 11

3 3

4 4

5 6

6 5

7 10

8 12

9 5

10 7

11 11

12 11

13 9

Name: loan\_amount\_sanctioned, dtype: int64

mean of salary is 5.357142857142857

mean of loan amount is 7.785714285714286

Estimated coefficients:

b0 = 2.0726643598615917

b1 = 1.0664359861591697

plot of a straight line



Enter the salary of a person to predict the loan amount to be sanctioned

123

loan amount predicted for the entered salary is by Linear Regression:

133.24429065743945

In [ ]:

9th

**import** pandas **as** pd

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

**from** sklearn.metrics **import** accuracy\_score

**from** sklearn.neighbors **import** KNeighborsClassifier

data**=**pd**.**read\_csv('C:\\Users\\mdtah\\Desktop\\rvr moodle\\4th Year\\ML Lab\\IT\\KNN-accuracy-caclculation\\apndcts.csv')

predictors**=**data**.**iloc[:,0:4]

print('Features of training dataset\n', predictors)

print('\n')

target**=**data**.**iloc[:,4]

print('Class of IRIS flower \n', target)

print('\n')

predictors\_train,predictors\_test,target\_train,target\_test **=** train\_test\_split(predictors,target,test\_size**=**0.3,random\_state**=None**)

knn**=**KNeighborsClassifier(n\_neighbors**=**5)

model**=**knn**.**fit(predictors\_train,target\_train)

accuracy**=**knn**.**score(predictors\_test,target\_test)

print('Accuracy of KNN classifier on IRIS data set',accuracy)

10th

**import** pandas **as** pd

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

**from** sklearn.metrics **import** accuracy\_score

**from** sklearn **import** svm

data**=**pd**.**read\_csv('C:\\Users\\mdtah\\Desktop\\ML prg\\10th\\apndcts.csv')

print('Features supplied for classification')

predictors**=**data**.**iloc[:,0:4]

print('\n',predictors)

print('Target class labels')

target**=**data**.**iloc[:,4]

print('\n',target)

predictors\_train,predictors\_test,target\_train,target\_test**=**train\_test\_split(predictors,target,test\_size**=**0.3,random\_state**=**123)

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

model**=**svm**.**fit(predictors\_train,target\_train)

prediction**=**svm**.**predict(predictors\_test)

print('Accuracy of SVM classifier', accuracy\_score(target\_test,prediction,normalize**=True**))