**X`**

1. **Implement A\* Search algorithm**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

from collections import deque

class Graph:

def \_\_init\_\_(self, adjac\_lis):

self.adjac\_lis = adjac\_lis

def get\_neighbors(self, v):

return self.adjac\_lis[v]

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start, stop):

open\_lst = set(start)

closed\_lst = set()

poo = {}

poo[start] = 0

par = {}

par[start] = start

while len(open\_lst) > 0:

n = None

for v in open\_lst:

if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == stop:

reconst\_path = []

while par[n] != n:

reconst\_path.append(n)

n = par[n]

reconst\_path.append(start)

reconst\_path.reverse()`

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_lst and m not in closed\_lst:

open\_lst.add(m)

par[m] = n

poo[m] = poo[n] + weight

else:

if poo[m] > poo[n] + weight:

poo[m] = poo[n] + weight

par[m] = n

if m in closed\_lst:

closed\_lst.remove(m)

open\_lst.add(m)

open\_lst.remove(n)

closed\_lst.add(n)

print('Path does not exist!')

return None

adjac\_lis = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

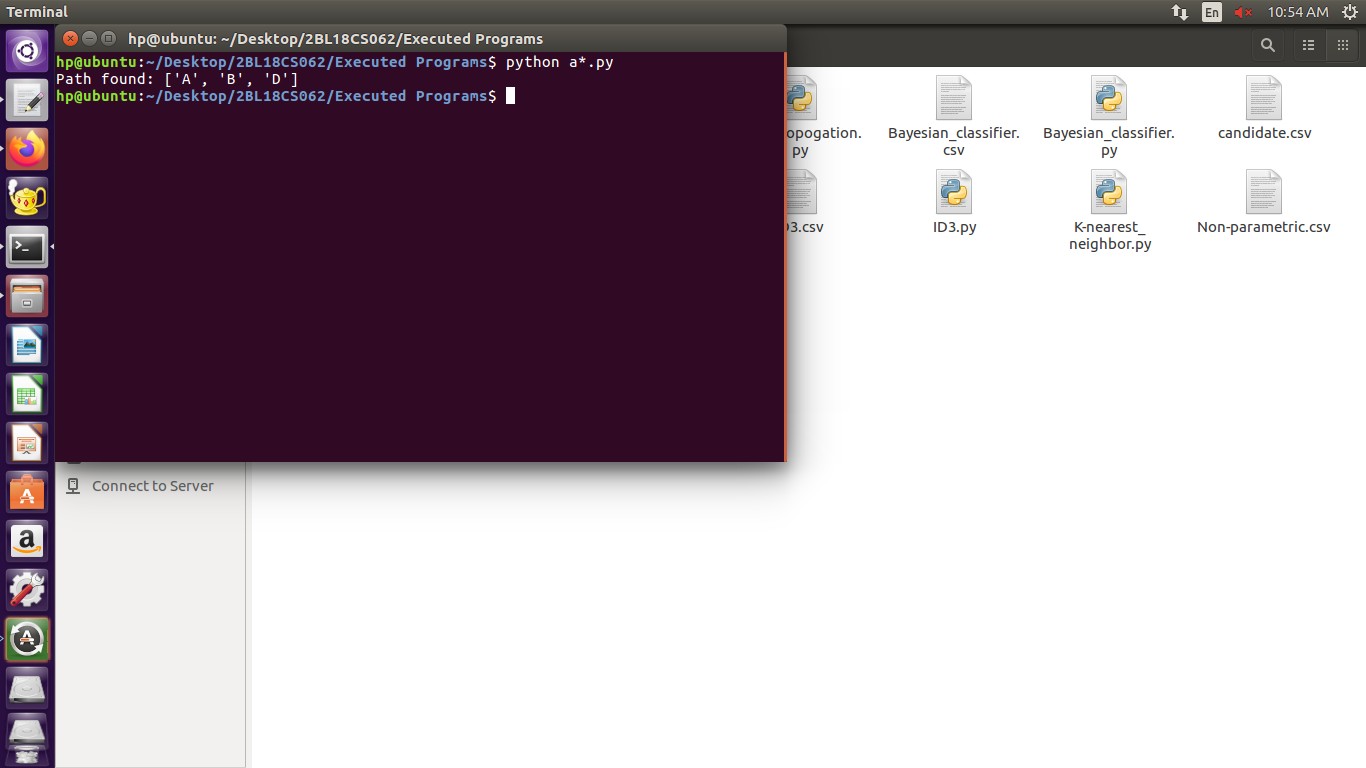
'C': [('D', 12)]

}

graph1 = Graph(adjac\_lis)

graph1.a\_star\_algorithm('A', 'D')

OUTPUT



1. **Implement AO\* Search algorithm.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

class Graph:

def \_\_init\_\_(self, graph, heuristicNodeList, startNode):

self.graph = graph

self.H=heuristicNodeList

self.start=startNode

self.parent={}

self.status={}

self.solutionGraph={}

def applyAOStar(self):

self.aoStar(self.start, False)

def getNeighbors(self, v):

return self.graph.get(v,'')

def getStatus(self,v):

return self.status.get(v,0)

def setStatus(self,v, val):

self.status[v]=val

def getHeuristicNodeValue(self, n):

return self.H.get(n,0)

def setHeuristicNodeValue(self, n, value):

self.H[n]=value

def printSolution(self):

print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)

print("------------------------------------------------------------")

print(self.solutionGraph)

print("------------------------------------------------------------")

def computeMinimumCostChildNodes(self, v):

minimumCost=0

costToChildNodeListDict={}

costToChildNodeListDict[minimumCost]=[]

flag=True

for nodeInfoTupleList in self.getNeighbors(v):

cost=0

nodeList=[]

for c, weight in nodeInfoTupleList:

cost=cost+self.getHeuristicNodeValue(c)+weight

nodeList.append(c)

if flag==True:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList

flag=False

else:

if minimumCost>cost:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList

return minimumCost, costToChildNodeListDict[minimumCost]

def aoStar(self, v, backTracking):

print("HEURISTIC VALUES :", self.H)

print("SOLUTION GRAPH :", self.solutionGraph)

print("PROCESSING NODE :", v)

print("-----------------------------------------------------------------------------------------")

if self.getStatus(v) >= 0:

minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

print(minimumCost, childNodeList)

self.setHeuristicNodeValue(v, minimumCost)

self.setStatus(v,len(childNodeList))

solved=True

for childNode in childNodeList:

self.parent[childNode]=v

if self.getStatus(childNode)!=-1:

solved=solved & False

if solved==True:

self.setStatus(v,-1)

self.solutionGraph[v]=childNodeList

if v!=self.start:

self.aoStar(self.parent[v], True)

if backTracking==False:

for childNode in childNodeList:

self.setStatus(childNode,0)

self.aoStar(childNode, False)

print ("Graph - 1")

h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}

graph1 = {

'A': [[('B', 1), ('C', 1)], [('D', 1)]],

'B': [[('G', 1)], [('H', 1)]],

'C': [[('J', 1)]],

'D': [[('E', 1), ('F', 1)]],

'G': [[('I', 1)]]

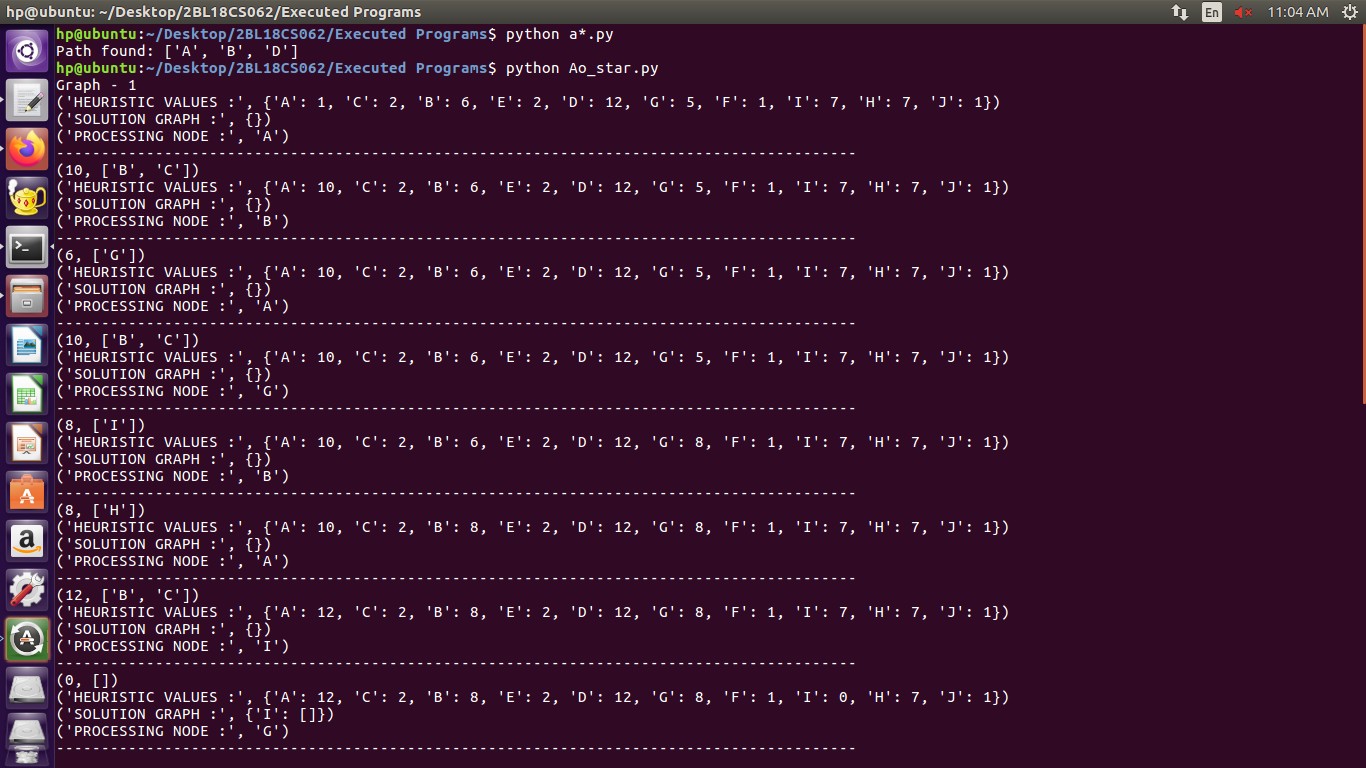
}

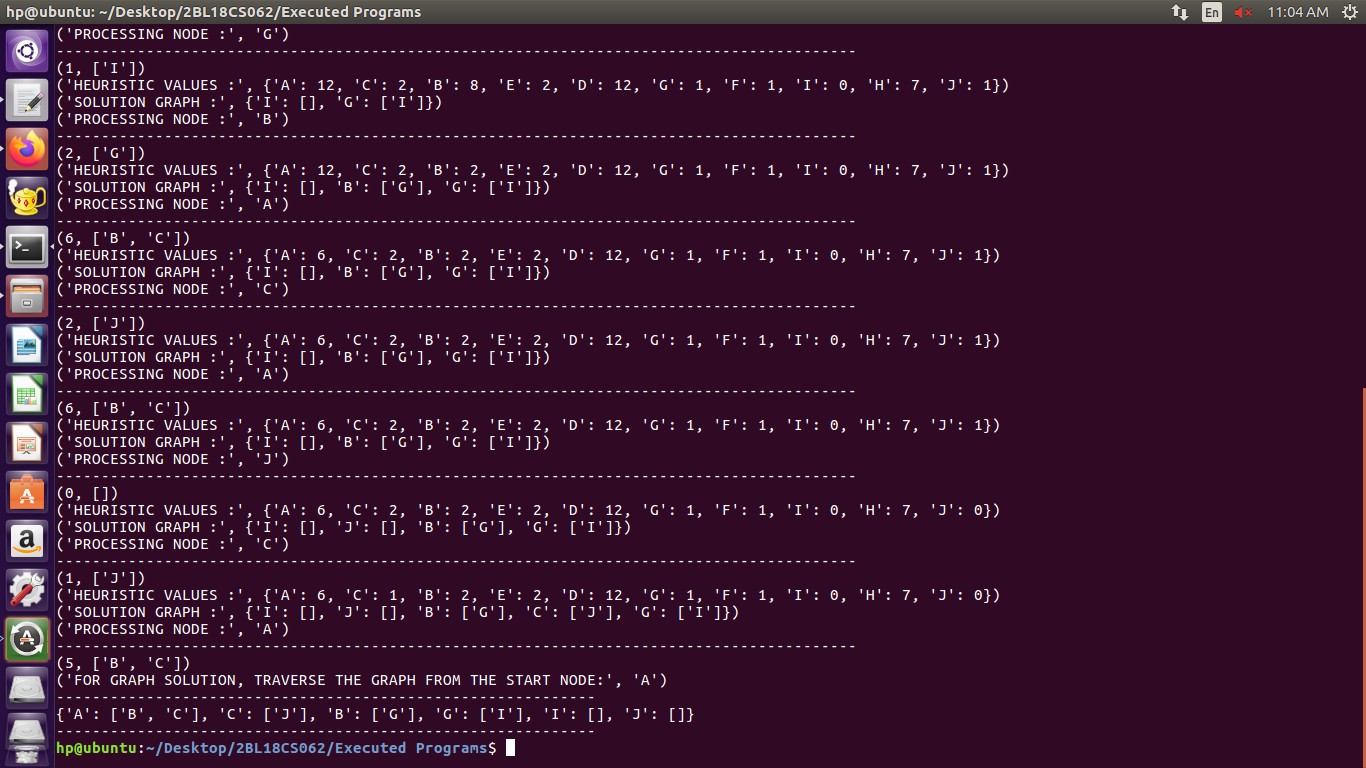
G1= Graph(graph1, h1, 'A')

G1.applyAOStar()

G1.printSolution()

**OUTPUT**





1. **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**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import numpy as np

import csv

with open('candidate.csv','r') as f:

reads=csv.reader(f)

tmp\_lst=np.array(list(reads))

concept=np.array(tmp\_lst[:,:-1])

target=np.array(tmp\_lst[:,-1])

for i in range(len(target)):

if(target[i]=='yes'):

specific\_h=concept[i]

break

h=[]

generic\_h=[['?' for i in range (len(specific\_h))]for i in range (len(specific\_h))]

print(type(generic\_h))

for i in range(len(target)):

if(target[i]=='yes'):

for j in range (len(specific\_h)):

if(specific\_h[j]!=concept[i][j]):

specific\_h[j]='?'

generic\_h[j][j]='?'

else:

for j in range(len(specific\_h)):

if(specific\_h[j]!=concept[i][j]):

generic\_h[j][j]=specific\_h[j]

else:

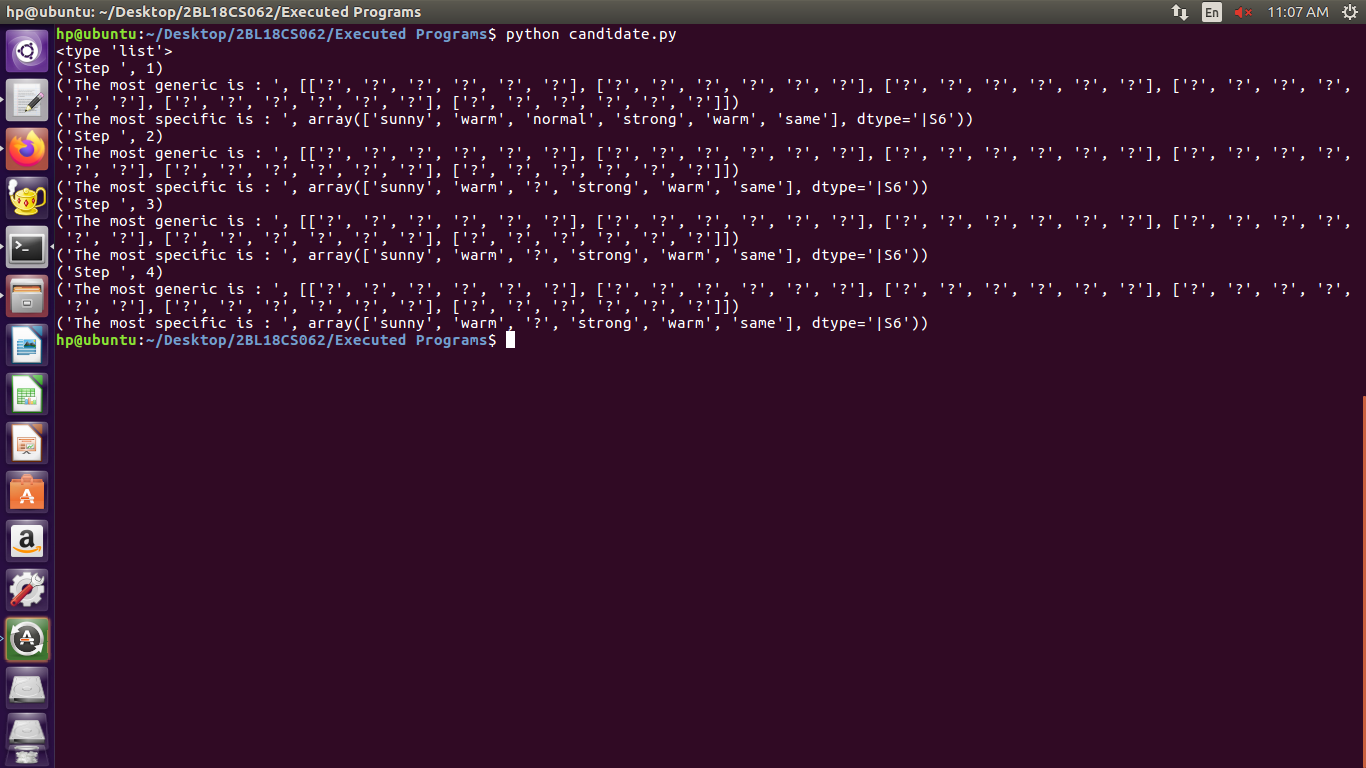
generic\_h[j][j]='?'

print("Step ",i+1)

print("The most generic is : ",generic\_h)

print("The most specific is : ",specific\_h)

**OUTPUT**



1. **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**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import numpy as np

import math

import csv

class Node:

def \_\_init\_\_ (self,attribute):

self.attribute=attribute

self.children=[]

self.answer=" "

def read\_data(filename):

with open(filename,'r') as csvfile:

datareader=csv.reader(csvfile,delimiter=',')

headers=next(datareader)

metadata=[]

traindata=[]

for name in headers:

metadata.append(name)

for row in datareader:

traindata.append(row)

return(metadata,traindata)

def subtables(data,col,delete):

dict={}

items=np.unique(data[:,col])

count=np.zeros((items.shape[0],1),dtype=np.int32)

for x in range(items.shape[0]):

for y in range(data.shape[0]):

if data[y,col]==items[x]:

count[x]+=1

for x in range(items.shape[0]):

dict[items[x]]=np.empty((int (count[x]),data.shape[1]),dtype="|S32")

pos=0

for y in range(data.shape[0]):

if data[y,col]==items[x]:

dict[items[x]][pos]=data[y]

pos+=1

if delete:

dict[items[x]]=np.delete(dict[items[x]],col,1)

return items,dict

def entropy(S):

items=np.unique(S)

if items.size==1:

return 0

counts = np.zeros((items.shape[0],1))

sums = 0

for x in range(items.shape[0]):

counts[x] = sum(S ==items[x])/(S.size\*1.0)

for count in counts:

sums +=-1\*count\*math.log(count,2)

return sums

def gain\_ratio(data,col):

items,dict=subtables(data,col,delete=False)

total\_size=data.shape[0]

entropies=np.zeros((items.shape[0],1))

intrinsic=np.zeros((items.shape[0],1))

for x in range(items.shape[0]):

ratio=dict[items[x]].shape[0]/(total\_size\*1.0)

entropies[x]=ratio\*entropy(dict[items[x]][:,-1])

intrinsic[x]=ratio\*math.log(ratio,2)

total\_entropy=entropy(data[:,-1])

iv=-1\*sum(intrinsic)

for x in range(entropies.shape[0]):

total\_entropy-=entropies[x]

return total\_entropy/iv

def create\_node(data,metadata):

if(np.unique(data[:,-1])).shape[0]==1:

node = Node(" ")

node.answer = np.unique(data[:,-1])[0]

return node

gains = np.zeros((data.shape[1]-1,1))

for col in range(data.shape[1]-1):

gains[col]=gain\_ratio(data,col)

split=np.argmax(gains)

node=Node(metadata[split])

metadata=np.delete(metadata,split,0)

items,dict=subtables(data,split,delete=True)

for x in range(items.shape[0]):

child = create\_node(dict[items[x]],metadata)

node.children.append((items[x],child))

return node

def empty(size):

S = " "

for x in range(size):

S+=" "

return S

def print\_tree(node,level):

if node.answer!=" ":

print(empty(level),node.answer)

return

print(empty(level),node.attribute)

for value,n in node.children:

print(empty(level+1),value)

print\_tree(n,level+2)

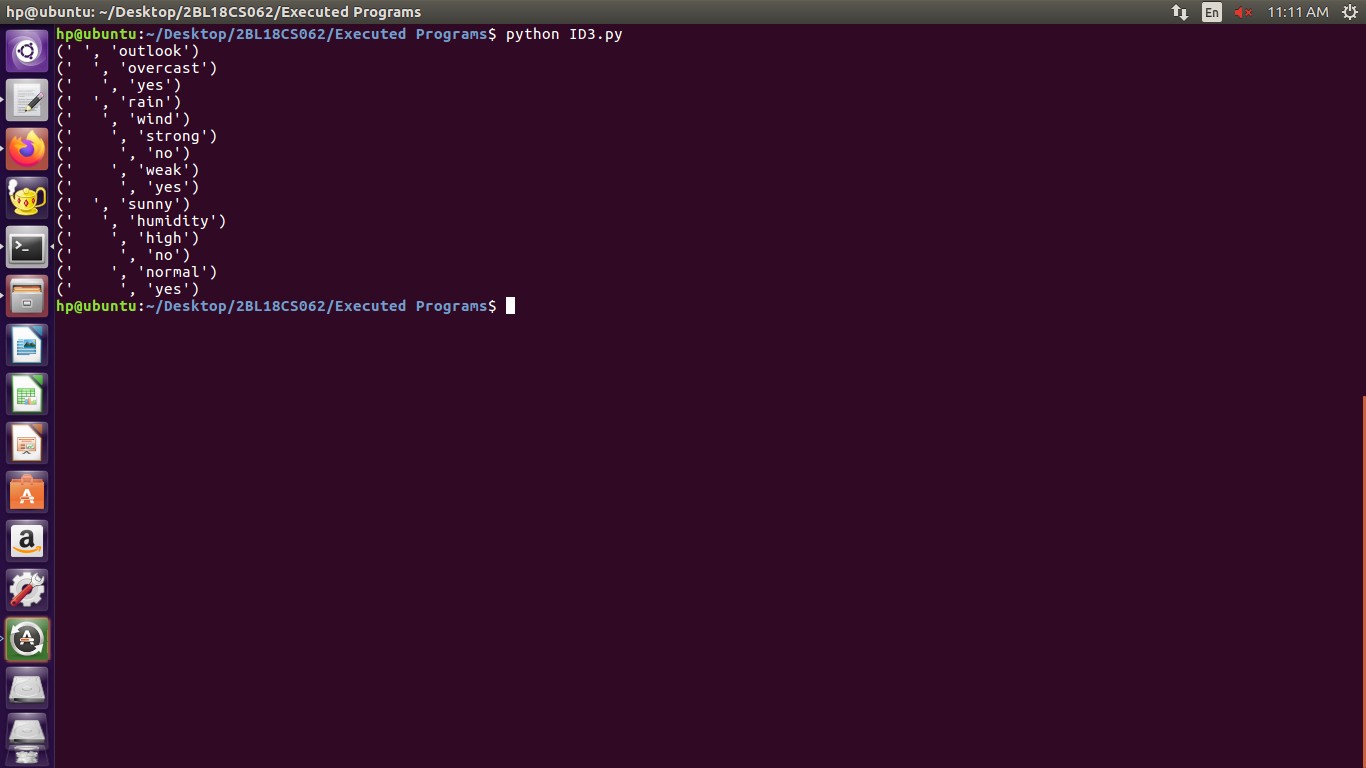
metadata,traindata=read\_data("ID3.csv")

data=np.array(traindata)

node=create\_node(data,metadata)

print\_tree(node,0)

**OUTPUT**



1. **Build an Artificial Neural Network by implementing the Back propagation algorithm and test the same using appropriate data sets.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

y=np.array(([92],[86],[89]),dtype=float)

X=X/np.amax(X,axis=0)

y=y/100

def sigmoid(x):

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

def derivatives\_sigmoid(x):

return x\*(1-x)

epoch=7000

lr=0.25

inputlayer\_neurons=2

hiddenlayer\_neurons=3

output\_neurons=1

wh=np.random.uniform(size=(inputlayer\_neurons,hiddenlayer\_neurons))

bh=np.random.uniform(size=(1,hiddenlayer\_neurons))

wout=np.random.uniform(size=(hiddenlayer\_neurons,output\_neurons))

bout=np.random.uniform(size=(1,output\_neurons))

for i in range(epoch):

hinp1=np.dot(X,wh)

hinp=hinp1+bh

hlayer\_act=sigmoid(hinp)

outinp1=np.dot(hlayer\_act,wout)

outinp=outinp1+bout

output=sigmoid(outinp)

EO=y-output

outgrad=derivatives\_sigmoid(output)

d\_output=EO\*outgrad

EH=d\_output.dot(wout.T)

hiddengrad=derivatives\_sigmoid(hlayer\_act)

d\_hiddenlayer=EH\*hiddengrad

wout+=hlayer\_act.T.dot(d\_output)\*lr

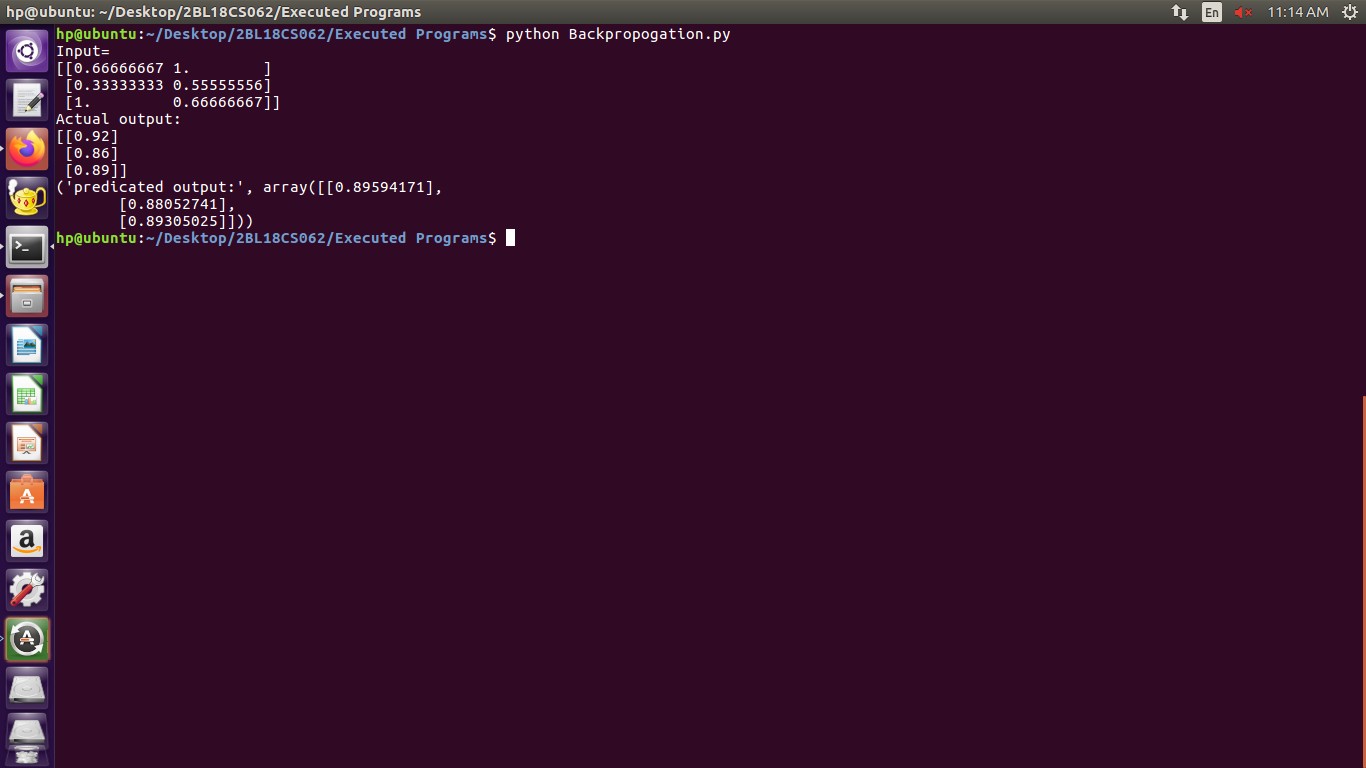
wh+=X.T.dot(d\_hiddenlayer)\*lr

print("Input=\n"+str(X))

print("Actual output:\n"+str(y))

print("predicated output:",output)

**OUTPUT**



1. **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.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

import csv

import random

import math

def loadCsv(filename):

lines = csv.reader(open(filename, "r"));

dataset = list(lines)

for i in range(len(dataset)):

dataset[i] = [float(x) for x in dataset[i]]

return dataset

def splitDataset(dataset, splitRatio):

trainSize = int(len(dataset) \* splitRatio);

trainSet = []

copy = list(dataset);

while len(trainSet) < trainSize:

index = random.randrange(len(copy));

trainSet.append(copy.pop(index))

return [trainSet, copy]

def separateByClass(dataset):

separated = {}

for i in range(len(dataset)):

vector = dataset[i]

if (vector[-1] not in separated):

separated[vector[-1]] = []

separated[vector[-1]].append(vector)

return separated

def mean(numbers):

return sum(numbers)/float(len(numbers))

def stdev(numbers):

avg = mean(numbers)

variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)

return math.sqrt(variance)

def summarize(dataset):

summaries = [(mean(attribute), stdev(attribute)) for attribute in zip(\*dataset)];

del summaries[-1]

return summaries

def summarizeByClass(dataset):

separated=separateByClass(dataset)

summaries={}

for classValue, instances in separated.items():

summaries[classValue]=summarize(instances)

return summaries

def calculateProbability(x,mean,stdev):

exponent = math.exp(-(math.pow(x-mean,2)/(2\*math.pow(stdev,2))))

return (1 / (math.sqrt(2\*math.pi) \* stdev)) \* exponent

def calculateClassProbabilities(summaries, inputVector):

probabilities = {}

for classValue, classSummaries in summaries.items():

probabilities[classValue] = 1

for i in range(len(classSummaries)):

mean, stdev = classSummaries[i]

x = inputVector[i]

probabilities[classValue] \*= calculateProbability(x, mean,stdev);

return probabilities

def predict(summaries, inputVector):

probabilities = calculateClassProbabilities(summaries, inputVector)

bestLabel, bestProb = None, -1

for classValue, probability in probabilities.items():

if bestLabel is None or probability > bestProb:

bestProb = probability

bestLabel=classValue

return bestLabel

def getPredictions(summaries,testSet):

predictions = []

for i in range(len(testSet)):

result = predict(summaries, testSet[i])

predictions.append(result)

return predictions

def getAccuracy(testSet, predictions):

correct = 0

for i in range(len(testSet)):

if testSet[i][-1] == predictions[i]:

correct += 1

return (correct/float(len(testSet))) \* 100.0

def main():

filename="Bayesian\_classifier.csv"

splitRatio=0.67

dataset=loadCsv(filename)

trainingSet,testSet=splitDataset(dataset,splitRatio)

print('Split{0} rows into train{1} and test={2}rows'.format(len(dataset),len(trainingSet),len(testSet)))

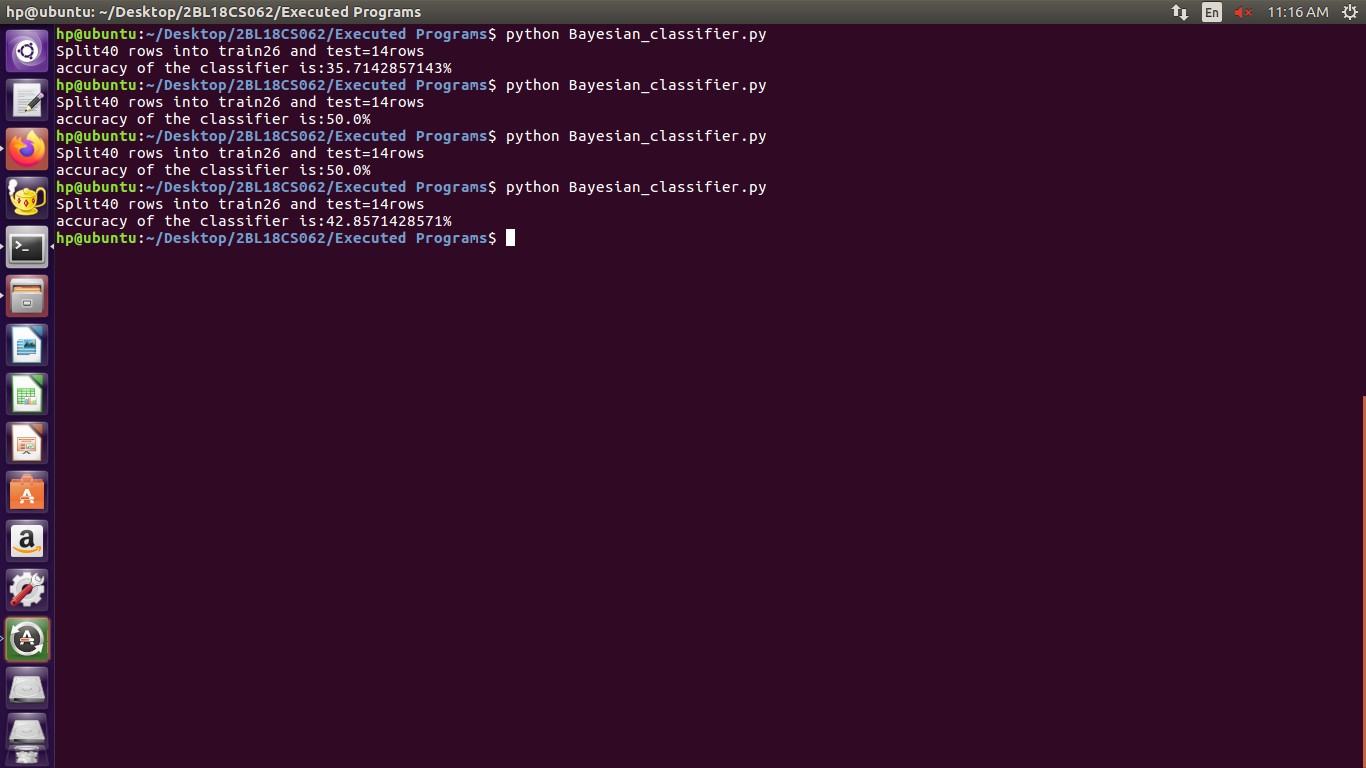
summaries = summarizeByClass(trainingSet);

predictions=getPredictions(summaries,testSet)

accuracy=getAccuracy(testSet,predictions)

print('accuracy of the classifier is:{0}%'.format(accuracy))

main()

**OUTPUT**

1. **Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-means algorithm. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

from sklearn.cluster import KMeans

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

data=pd.read\_csv("EM\_Algorithm.csv")

df1=pd.DataFrame(data)

print(df1)

f1 = df1['Distance\_Feature'].values

f2 = df1['Speeding\_Feature'].values

X=np.matrix(list(zip(f1,f2)))

plt.plot()

plt.xlim([0, 100])

plt.ylim([0, 50])

plt.title('Dataset')

plt.ylabel('speeding\_feature')

plt.xlabel('Distance\_Feature')

plt.scatter(f1,f2)

plt.show()

plt.plot()

colors = ['b', 'g', 'r']

markers = ['o', 'v', 's']

kmeans\_model = KMeans(n\_clusters=3).fit(X)

plt.plot()

for i, l in enumerate(kmeans\_model.labels\_):

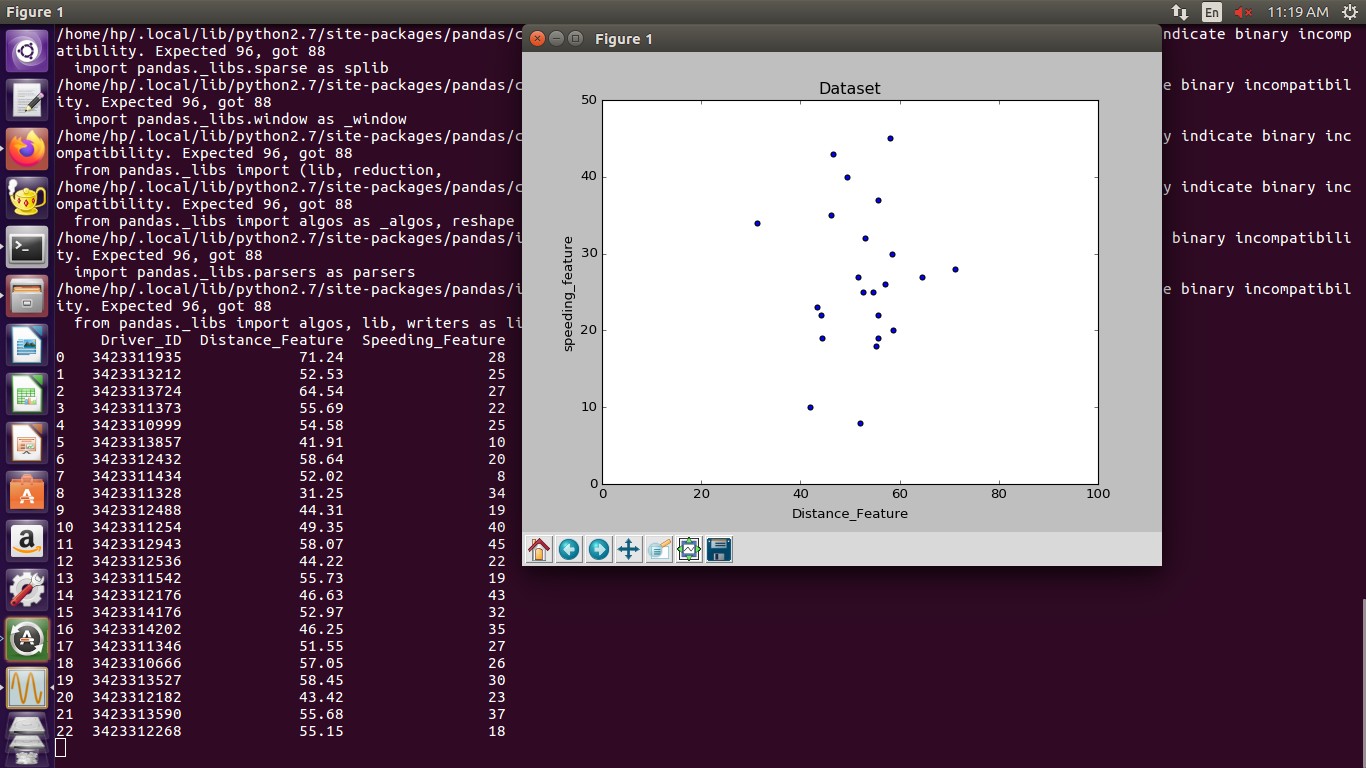
plt.plot(f1[i], f2[i], color=colors[l], marker=markers[l],ls='None')

plt.xlim([0, 100])

plt.ylim([0, 50])

plt.show()

**OUTPUT**



1. **Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem**.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

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

from sklearn.neighbors import KNeighborsClassifier

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

from sklearn import datasets

iris=datasets.load\_iris()

iris\_data=iris.data

iris\_labels=iris.target

print(iris\_data)

print(iris\_labels)

x\_train, x\_test, y\_train, y\_test=train\_test\_split(iris\_data,iris\_labels,test\_size=0.30)

classifier=KNeighborsClassifier(n\_neighbors=5)

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

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

print('confusion matrix is as follows')

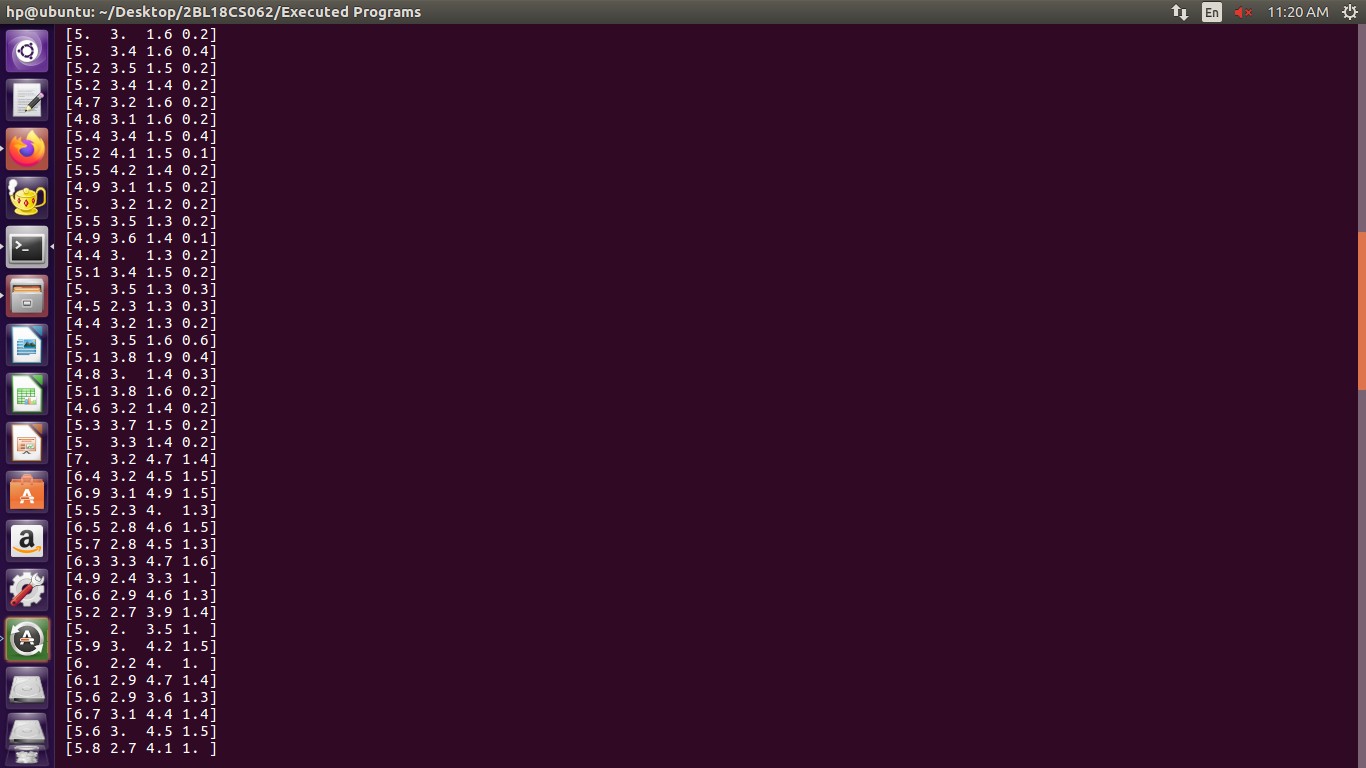
print(confusion\_matrix(y\_test,y\_pred))

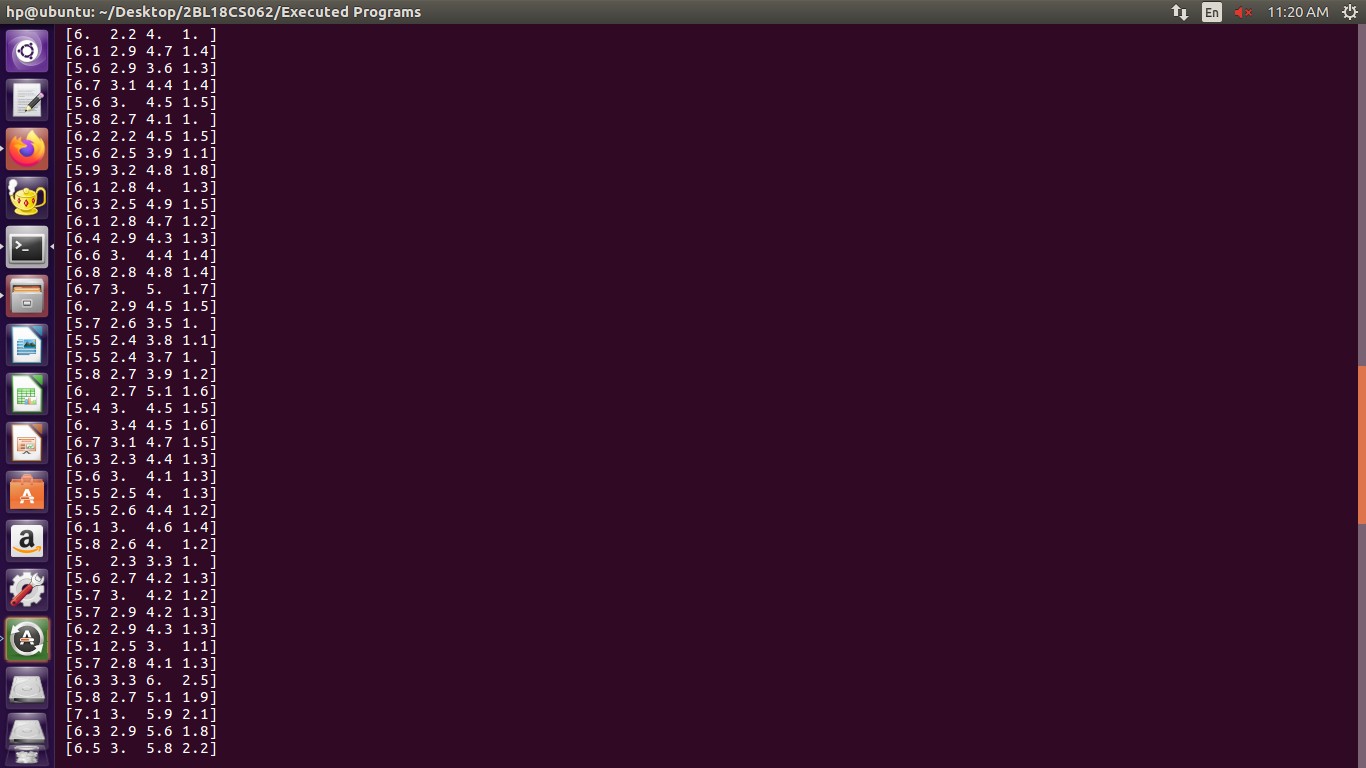
print('Accuracy metrics')

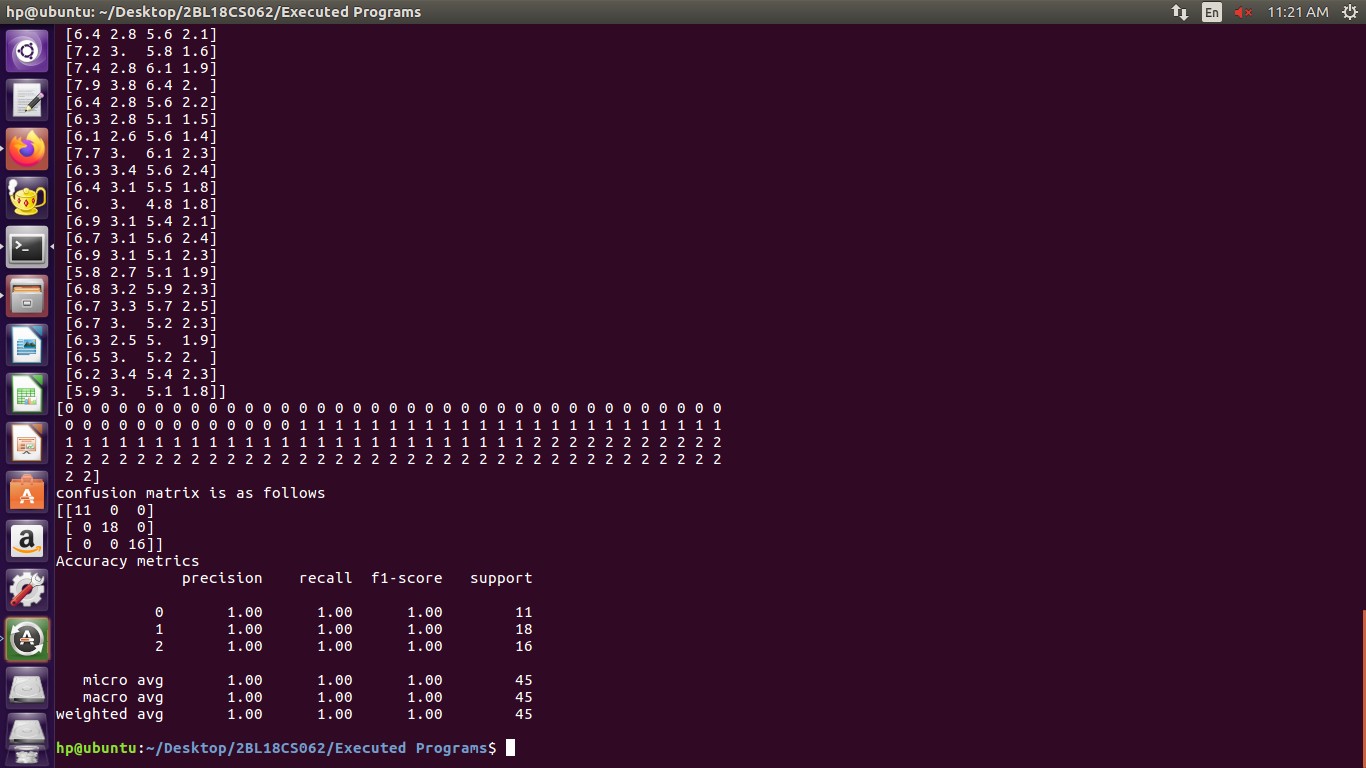
print(classification\_report(y\_test,y\_pred))

**OUTPUT**









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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#Importing Required Moudles

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np1

#Calculationg Weight Matrix Using e^(-(x-x0)^2/2\*r^2)

def kernel(point,xmat, k):

m,n = np1.shape(xmat)

weights = np1.mat(np1.eye((m)))

for j in range(m):

diff = point - X[j]

weights[j,j] = np1.exp(diff\*diff.T/(-2.0\*k\*\*2)) # Using above Formula

return weights

def localWeight(point,xmat,ymat,k):

wei = kernel(point,xmat,k)

W=(X.T\*(wei\*X)).I\*(X.T\*(wei\*ymat.T)) # Calculate Beta(model term parameter) Using β(xo) = (X^T WX)^-1 X^T Wy

return W

def localWeightRegression(xmat,ymat,k):

m,n = np1.shape(xmat)

ypred = np1.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('Non-Parametric.csv')

bill = np1.array(data.total\_bill)

tip = np1.array(data.tip)

#preparing and add 1 in bill

mbill = np1.mat(bill)

mtip = np1.mat(tip)

m= np1.shape(mbill)[1]

one = np1.mat(np1.ones(m))

X= np1.hstack((one.T,mbill.T))

#set k here

ypred = localWeightRegression(X,mtip,2)

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') #Giving Color to Points

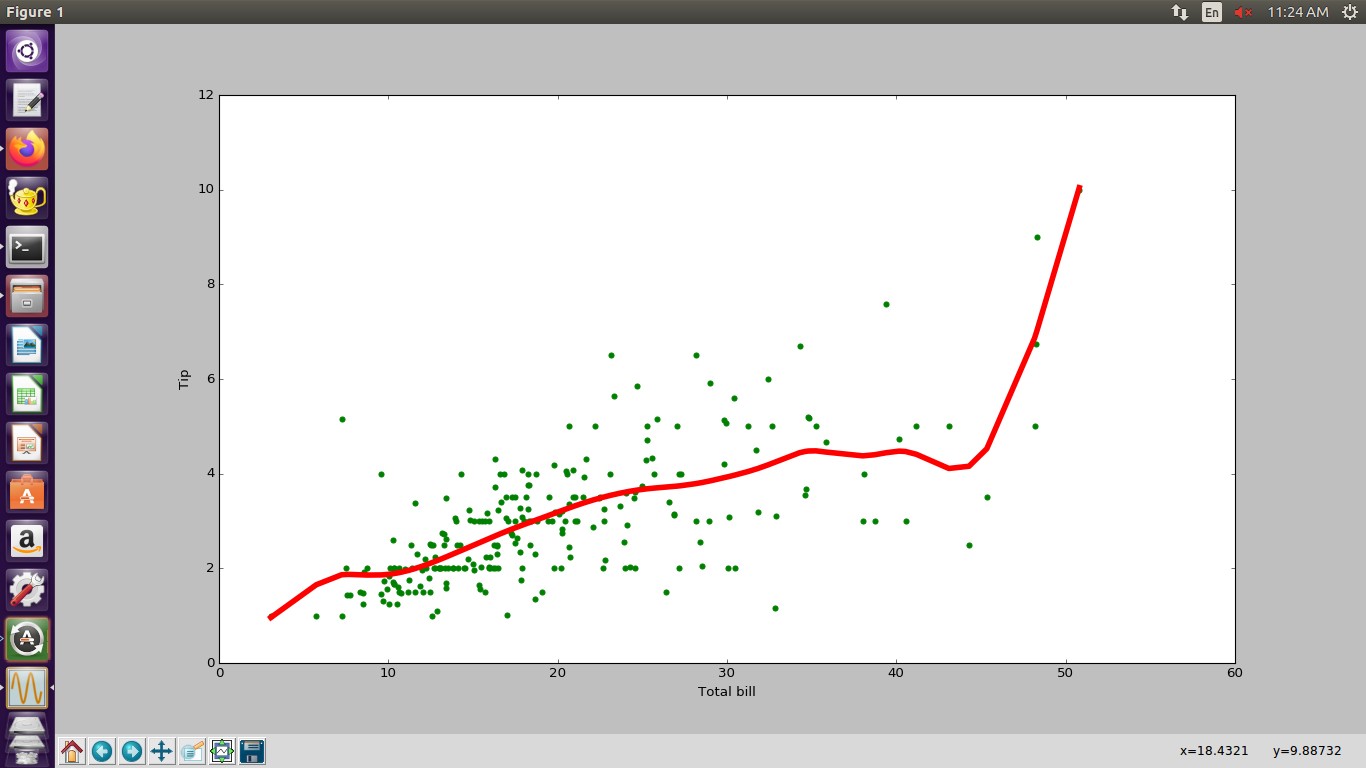
ax.plot(xsort[:,1],ypred[SortIndex], color = 'red', linewidth=5) #Giving Color to line

plt.xlabel('Total bill')

plt.ylabel('Tip')

plt.show()

**OUTPUT**



\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*