5. 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.

# Importing the libraries

import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

# Importing the dataset

dataset = pd.read\_csv('User\_Data.csv')

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

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

# Splitting the dataset into the Training set and Test set

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)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

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

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

#Displaying the dataset

dataset

OUTPUT:

|  | User ID | Gender | Age | EstimatedSalary | Purchased |
| --- | --- | --- | --- | --- | --- |
| 0 | 15624510 | Male | 19 | 19000 | 0 |
| 1 | 15810944 | Male | 35 | 20000 | 0 |
| 2 | 15668575 | Female | 26 | 43000 | 0 |
| 3 | 15603246 | Female | 27 | 57000 | 0 |
| 4 | 15804002 | Male | 19 | 76000 | 0 |
| ... | ... | ... | ... | ... | ... |
| 395 | 15691863 | Female | 46 | 41000 | 1 |
| 396 | 15706071 | Male | 51 | 23000 | 1 |
| 397 | 15654296 | Female | 50 | 20000 | 1 |
| 398 | 15755018 | Male | 36 | 33000 | 0 |
| 399 | 15594041 | Female | 49 | 36000 | 1 |
|  |  |  |  |  |  |

400 rows × 5 columns

# Fitting Naive Bayes to the Training set

from sklearn.naive\_bayes import GaussianNB

classifier = GaussianNB()

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

OUTPUT:GaussianNB()

# Predicting the Test set results

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

# Making the Confusion Matrix

from sklearn.metrics import confusion\_matrix

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

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

OUTPUT: Confusion Matrix :

[[65 3]

[ 7 25]]

# Visualising the Training set results

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_train, y\_train

X1, X2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step = 0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(X1, X2, classifier.predict(nm.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha = 0.75, cmap = ListedColormap(('purple', 'green')))

mtp.xlim(X1.min(), X1.max())

mtp.ylim(X2.min(), X2.max())

for i, j in enumerate(nm.unique(y\_set)):

mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

c = ListedColormap(('purple', 'green'))(i), label = j)

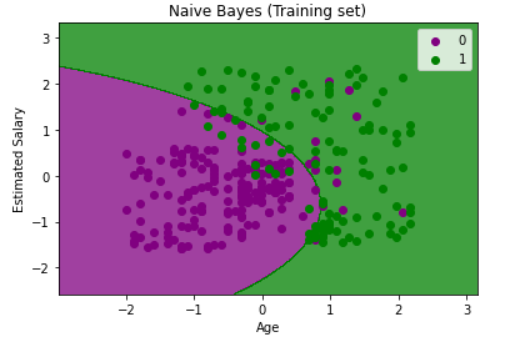
mtp.title('Naive Bayes (Training set)')

mtp.xlabel('Age')

mtp.ylabel('Estimated Salary')

mtp.legend()

mtp.show()



# Visualising the Test set results

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_test, y\_test

X1, X2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step = 0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(X1, X2, classifier.predict(nm.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),

alpha = 0.75, cmap = ListedColormap(('red', 'green')))

mtp.xlim(X1.min(), X1.max())

mtp.ylim(X2.min(), X2.max())

for i, j in enumerate(nm.unique(y\_set)):

mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

c = ListedColormap(('red', 'green'))(i), label = j)

mtp.title('Naive Bayes (test set)')

mtp.xlabel('Age')

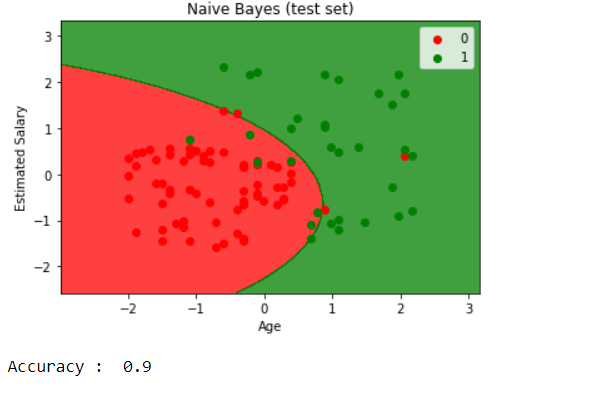
mtp.ylabel('Estimated Salary')

mtp.legend()

mtp.show()

from sklearn.metrics import accuracy\_score

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



11. Write a Program to Implement travelling salesman problem using Python.

*# Python3 program to implement traveling salesman*

*# problem using naive approach*.

from sys import maxsize

from itertools import permutations

V = 4

*# implementation of traveling Salesman Problem*

def travellingSalesmanProblem(graph, s):

*# store all vertex apart from source vertex*

vertex = []

for i in range(V):

if i != s:

vertex.append(i)

*# store minimum weight Hamiltonian Cycle*

min\_path = maxsize

next\_permutation=permutations(vertex)

for i in next\_permutation:

*# store current Path weight(cost)*

current\_pathweight = 0

*# compute current path weight*

k = s

for j in i:

current\_pathweight += graph[k][j]

k = j

current\_pathweight += graph[k][s]

*# Update minimum*

min\_path = min(min\_path, current\_pathweight)

return min\_path

*# Driver Code*

if \_\_name\_\_ == "\_\_main\_\_":

*# matrix representation of graph*

graph = [[0, 10, 15, 20], [10, 0, 35, 25],

[15, 35, 0, 30], [20, 25, 30, 0]]

s = 0

print(travellingSalesmanProblem(graph, s))

Output

80

6. Write a program to implement Logistic regression classifier to find accuracy for training and test fruit data set.

#Data Pre-procesing Step

# importing libraries

import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

#importing datasets

data\_set= pd.read\_csv('User\_Data.csv')

data\_set

Output:

User ID Gender Age EstimatedSalary Purchased

0 15624510 Male 19 19000 0

1 15810944 Male 35 20000 0

2 15668575 Female 26 43000 0

3 15603246 Female 27 57000 0

4 15804002 Male 19 76000 0

... ... ... ... ... ... ... ...

395 15691863 Female 46 41000 1

396 15706071 Male 51 23000 1

397 15654296 Female 50 20000 1

398 15755018 Male 36 33000 0

399 15594041 Female 49 36000 1

400 rows × 5 columns

#Data Pre-procesing Step

# importing libraries

import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

#importing datasets

data\_set= pd.read\_csv('User\_Data.csv')

#Extracting Independent and dependent Variable

x= data\_set.iloc[:, [2,3]].values

y= data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

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)

#feature Scaling

from sklearn.preprocessing import StandardScaler

st\_x= StandardScaler()

x\_train= st\_x.fit\_transform(x\_train)

x\_test= st\_x.transform(x\_test)

#Fitting Logistic Regression to the training set

from sklearn.linear\_model import LogisticRegression

classifier= LogisticRegression(random\_state=0)

classifier.fit(x\_train, y\_train)

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

                   intercept\_scaling=1, l1\_ratio=None, max\_iter=100,

                   multi\_class='warn', n\_jobs=None, penalty='l2',

                   random\_state=0, solver='warn', tol=0.0001, verbose=0,

                   warm\_start=False)

#Predicting the test set result

y\_pred= classifier.predict(x\_test)

print(y\_pred)

Output:

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

0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0

0 0 1 0 1 1 1 1 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1]

#Creating the Confusion matrix

from sklearn.metrics import confusion\_matrix

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

print("Confusion Matrix : \n",cm)

Confusion Matrix :

[[65 3]

[ 8 24]]

#Visualizing the training set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_train, y\_train

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(('purple','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

       c = ListedColormap(('purple', 'green'))(i), label = j)

mtp.title('Logistic Regression (Training set)')

mtp.xlabel('Age')

mtp.ylabel('Estimated Salary')

mtp.legend()

mtp.show()

Output:



#Visulaizing the test set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_test, y\_test

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),

nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(('purple','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

        c = ListedColormap(('purple', 'green'))(i), label = j)

mtp.title('Logistic Regression (Test set)')

mtp.xlabel('Age')

mtp.ylabel('Estimated Salary')

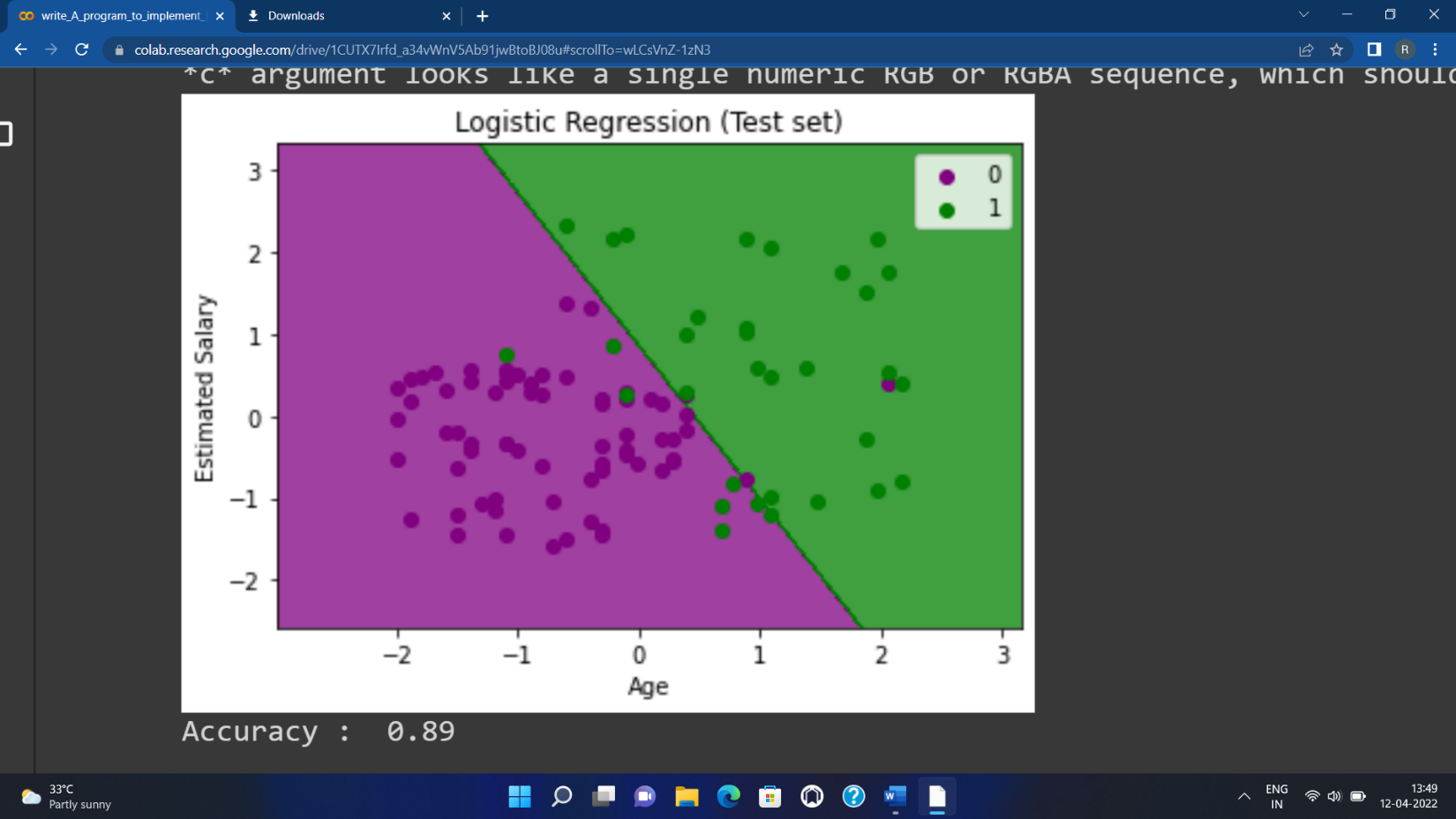
mtp.legend()

mtp.show()

from sklearn.metrics import accuracy\_score

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

Output:



Accuracy : 0.89

4. Build an Artificial Neural Network by implementing the Backpropagation 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) # maximum of X array longitudinally

y = y/100

#Sigmoid Function

def sigmoid (x):

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

#Derivative of Sigmoid Function

def derivatives\_sigmoid(x):

return x \* (1 - x)

#Variable initialization

epoch=7000 #Setting training iterations

lr=0.1 #Setting learning rate

inputlayer\_neurons = 2 #number of features in data set

hiddenlayer\_neurons = 3 #number of hidden layers neurons

output\_neurons = 1 #number of neurons at output layer

#weight and bias initialization

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))

#draws a random range of numbers uniformly of dim x\*y

for i in range(epoch):

#Forward Propogation

hinp1=np.dot(X,wh)

hinp=hinp1 + bh

hlayer\_act = sigmoid(hinp)

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

outinp= outinp1+ bout

output = sigmoid(outinp)

#Backpropagation

EO = y-output

outgrad = derivatives\_sigmoid(output)

d\_output = EO\* outgrad

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

hiddengrad = derivatives\_sigmoid(hlayer\_act) #how much hidden layer wts contributed to error

d\_hiddenlayer = EH \* hiddengrad

wout += hlayer\_act.T.dot(d\_output) \*lr # dotproduct of nextlayererror and currentlayerop

# bout += np.sum(d\_output, axis=0,keepdims=True) \*lr

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

#bh += np.sum(d\_hiddenlayer, axis=0,keepdims=True) \*lr

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

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

print("Predicted Output: \n",output)

Output:

Input:

[[0.66666667 1. ]

[0.33333333 0.55555556]

[1. 0.66666667]]

Actual Output:

[[0.92]

[0.86]

[0.89]]

Predicted Output:

[[0.8946056 ]

[0.88075857]

[0.89439617]]

PART B:

1.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

import csv

hypo=['%','%','%','%','%','%']

with open('Training\_examples.csv') as csv\_file:

readcsv = csv.reader(csv\_file, delimiter=',')

data=[]

print("\nThe given training examples are:")

for row in readcsv:

print(row)

if row[len(row)-1] =='Yes':

data.append(row)

print("\nThe positive examples are:")

for x in data:

print(x)

TotalExamples=len(data)

i=0

j=0

k=0

print("\nThe steps of the Find-s algorithm are\n",hypo)

list =[]

p=0

d=len(data[p])-1

for j in range(d):

list.append(data[i][j])

hypo=list

for i in range(1,TotalExamples):

for k in range(d):

if hypo[k]!=data[i][k]:

hypo[k]='?'

else:

hypo[k]

print(hypo)

print("\nThe maximally specific Find-s hypothesis for the given training examples is");

list=[]

for i in range(d):

list.append(hypo[i])

print(list)

OUTPUT:

The given training examples are:

['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']

The positive examples are:

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

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

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

The steps of the Find-s algorithm are

['%', '%', '%', '%', '%', '%']

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

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

The maximally specific Find-s hypothesis for the given training examples is

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

**6. Write a Program to Implement Tower of Hanoi using Python.**

def  TowerOfHanoi(n , source, destination, auxiliary):

if n==1:

print ("Move disk 1 from source",source,"to destination",destination)

return

TowerOfHanoi(n-1, source, auxiliary, destination)

print ("Move disk",n,"from source",source,"to destination",destination)

TowerOfHanoi(n-1, auxiliary, destination, source)

n = 3

TowerOfHanoi(n,'A','B','C')

**OUTPUT:**

Move disk 1 from source A to destination B

Move disk 2 from source A to destination C

Move disk 1 from source B to destination C

Move disk 3 from source A to destination B

Move disk 1 from source C to destination A

Move disk 2 from source C to destination B

Move disk 1 from source A to destination B

7 write a program to implement N queen problem using python

global N

N = 4

def printSolution(board):

  for i in range(N):

    for j in range(N):

      print (board[i][j], end = " ")

    print()

def isSafe(board, row, col):

  for i in range(col):

    if board[row][i] == 1:

      return False

  for i, j in zip(range(row, -1, -1),

          range(col, -1, -1)):

    if board[i][j] == 1:

      return False

  for i, j in zip(range(row, N, 1),

          range(col, -1, -1)):

    if board[i][j] == 1:

      return False

  return True

def solveNQUtil(board, col):

  if col >= N:

    return True

  for i in range(N):

    if isSafe(board, i, col):

      board[i][col] = 1

      if solveNQUtil(board, col + 1) == True:

        return True

      board[i][col] = 0

  return False

def solveNQ():

  board = [ [0, 0, 0, 0],

      [0, 0, 0, 0],

      [0, 0, 0, 0],

      [0, 0, 0, 0] ]

  if solveNQUtil(board, 0) == False:

    print ("Solution does not exist")

    return False

  printSolution(board)

  return True

solveNQ()

**output:**

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

True

**9.Write a program to implement AO \* algorithm in python.**

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 :**

Graph - 1

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

SOLUTION GRAPH : {}

PROCESSING NODE : A

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

10 ['B', 'C']

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : B

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

6 ['G']

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : A

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

10 ['B', 'C']

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : G

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

8 ['I']

HEURISTIC VALUES : {'A': 10, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : B

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

8 ['H']

HEURISTIC VALUES : {'A': 10, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : A

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

12 ['B', 'C']

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 7, 'J': 1}

SOLUTION GRAPH : {}

PROCESSING NODE : I

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

0 []

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 8, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': []}

PROCESSING NODE : G

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

1 ['I']

HEURISTIC VALUES : {'A': 12, 'B': 8, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': [], 'G': ['I']}

PROCESSING NODE : B

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

2 ['G']

HEURISTIC VALUES : {'A': 12, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

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

6 ['B', 'C']

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : C

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

2 ['J']

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : A

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

6 ['B', 'C']

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 1}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G']}

PROCESSING NODE : J

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

0 []

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': []}

PROCESSING NODE : C

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

1 ['J']

HEURISTIC VALUES : {'A': 6, 'B': 2, 'C': 1, 'D': 12, 'E': 2, 'F': 1, 'G': 1, 'H': 7, 'I': 0, 'J': 0}

SOLUTION GRAPH : {'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J']}

PROCESSING NODE : A

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

5 ['B', 'C']

FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE: A

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

{'I': [], 'G': ['I'], 'B': ['G'], 'J': [], 'C': ['J'], 'A': ['B', 'C']}

**8. Write a program to implement A\* algorithm using python.**

class Node():

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position

def astar(maze, start, end):

start\_node = Node(None, start)

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = Node(None, end)

end\_node.g = end\_node.h = end\_node.f = 0

open\_list = []

closed\_list = []

open\_list.append(start\_node)

while len(open\_list) > 0:

current\_node = open\_list[0]

current\_index = 0

for index, item in enumerate(open\_list):

if item.f < current\_node.f:

current\_node = item

current\_index = index

open\_list.pop(current\_index)

closed\_list.append(current\_node)

if current\_node == end\_node:

path = []

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

return path[::-1] # Return reversed path

children = []

for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

if node\_position[0] > (len(maze) - 1) or node\_position[0] < 0 or node\_position[1] > (len(maze[len(maze)-1]) -1) or node\_position[1] < 0:

continue

if maze[node\_position[0]][node\_position[1]] != 0:

continue

new\_node = Node(current\_node, node\_position)

children.append(new\_node)

for child in children:

for closed\_child in closed\_list:

if child == closed\_child:

continue

child.g = current\_node.g + 1

child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + ((child.position[1] - end\_node.position[1]) \*\* 2)

child.f = child.g + child.h

for open\_node in open\_list:

if child == open\_node and child.g > open\_node.g:

continue

open\_list.append(child)

def main():

maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)

end = (7, 6)

path = astar(maze, start, end)

print(path)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**OUTPUT:**



[(0, 0), (1, 1), (2, 2),(3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]

**3. Python implementation of automatic Tic Tac Toe game using random number**

import numpy as np

import random

from time import sleep

def create\_board():

return(np.array([[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]))

def possibilities(board):

l = []

for i in range(len(board)):

for j in range(len(board)):

if board[i][j] == 0:

l.append((i, j))

return(l)

def random\_place(board, player):

selection = possibilities(board)

current\_loc = random.choice(selection)

board[current\_loc] = player

return(board)

def row\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[x, y] != player:

win = False

continue

if win == True:

return(win)

return(win)

def col\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[y][x] != player:

win = False

continue

if win == True:

return(win)

return(win)

def diag\_win(board, player):

win = True

y = 0

for x in range(len(board)):

if board[x, x] != player:

win = False

if win:

return win

win = True

if win:

for x in range(len(board)):

y = len(board) - 1 - x

if board[x, y] != player:

win = False

return win

def evaluate(board):

winner = 0

for player in [1, 2]:

if (row\_win(board, player) or

col\_win(board,player) or

diag\_win(board,player)):

winner = player

if np.all(board != 0) and winner == 0:

winner = -1

return winner

def play\_game():

board, winner, counter = create\_board(), 0, 1

print(board)

sleep(2)

while winner == 0:

for player in [1, 2]:

board = random\_place(board, player)

print("Board after " + str(counter) + " move")

print(board)

sleep(2)

counter += 1

winner = evaluate(board)

if winner != 0:

break

return(winner)

print("Winner is: " + str(play\_game()))

Output:

[[0 0 0]

[0 0 0]

[0 0 0]]

Board after 1 move

[[0 0 0]

[0 0 0]

[0 0 1]]

Board after 2 move

[[0 0 2]

[0 0 0]

[0 0 1]]

Board after 3 move

[[0 1 2]

[0 0 0]

[0 0 1]]

Board after 4 move

[[0 1 2]

[0 0 2]

[0 0 1]]

Board after 5 move

[[0 1 2]

[0 0 2]

[0 1 1]]

Board after 6 move

[[0 1 2]

[2 0 2]

[0 1 1]]

Board after 7 move

[[0 1 2]

[2 0 2]

[1 1 1]]

Winner is: 1

12.Writea program to implement 8 puzzle problem using python

import copy

from heapq import heappush, heappop

n = 3

row = [ 1, 0, -1, 0 ]

col = [ 0, -1, 0, 1 ]

class priorityQueue:

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

self.heap = []

def push(self, k):

heappush(self.heap, k)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class node:

def \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

cost, level):

self.parent = parent

self.mat = mat

self.empty\_tile\_pos = empty\_tile\_pos

self.cost = cost

self.level = level

def \_\_lt\_\_(self, nxt):

return self.cost < nxt.cost

def calculateCost(mat, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mat[i][j]) and

(mat[i][j] != final[i][j])):

count += 1

return count

def newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

level, parent, final) -> node:

new\_mat = copy.deepcopy(mat)

x1 = empty\_tile\_pos[0]

y1 = empty\_tile\_pos[1]

x2 = new\_empty\_tile\_pos[0]

y2 = new\_empty\_tile\_pos[1]

new\_mat[x1][y1], new\_mat[x2][y2] = new\_mat[x2][y2], new\_mat[x1][y1]

cost = calculateCost(new\_mat, final)

new\_node = node(parent, new\_mat, new\_empty\_tile\_pos,

cost, level)

return new\_node

def printMatrix(mat):

for i in range(n):

for j in range(n):

print("%d " % (mat[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatrix(root.mat)

print()

def solve(initial, empty\_tile\_pos, final):

pq = priorityQueue()

cost = calculateCost(initial, final)

root = node(None, initial,empty\_tile\_pos, cost, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.cost == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_pos = [

minimum.empty\_tile\_pos[0] + row[i],

minimum.empty\_tile\_pos[1] + col[i], ]

if isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

child=newNode(minimum.mat,minimum.empty\_tile\_pos,new\_tile\_pos,

minimum.level + 1minimum, final,)

pq.push(child)

initial = [ [ 1, 2, 3 ],[ 5, 6, 0 ],[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],[ 5, 8, 6 ],[ 0, 7, 4 ] ]

empty\_tile\_pos = [ 1, 2 ]

solve(initial, empty\_tile\_pos, final)

Output:

1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

5 8 6

7 0 4

1 2 3

5 8 6

0 7 4

9. Write a program to implement SVM classifier to find accuracy for training and testing fruit data set.

import pandas as pd

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

from sklearn.preprocessing import MinMaxScaler

from sklearn.svm import SVC

fruits = pd.read\_table('fruit\_data\_with\_colors.txt')

feature\_names = ['mass', 'width', 'height', 'color\_score']

X = fruits[feature\_names]

Y = fruits['fruit\_label']

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y)

X\_train = MinMaxScaler().fit\_transform(X\_train)

X\_test = MinMaxScaler().fit\_transform(X\_test)

svm = SVC().fit(X\_train, Y\_train)

print('Accuracy of SVM classifier on training set : {:.2f}'.format(svm.score(X\_train, Y\_train)))

print('Accuracy of SVM classifier on testing set : {:.2f}'.format(svm.score(X\_test, Y\_test)))

output:

Accuracy of SVM classifier on training set : 0.98

Accuracy of SVM classifier on testing set : 0.93

**2. WRITE A PROGRAM TO IMPLEMENT CANDIDATE ELIMINATION ALOGRITHM**

**import csv**

**with open("Training\_examples.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:**

**Steps of Candidate Elimination Algorithm 1**

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

**[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]**

**Steps of Candidate Elimination Algorithm 2**

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

**[['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]**

**Steps of Candidate Elimination Algorithm 3**

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

**[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', 'Same']]**

**Steps of Candidate Elimination Algorithm 4**

**['Sunny', 'Warm', '?', 'Strong', '?', '?']**

**[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]**

**Final specific hypothesis:**

**['Sunny', 'Warm', '?', 'Strong', '?', '?']**

**Final general hypothesis:**

**[['Sunny', '?', '?', '?', '?', '?'], ['?', 'Warm', '?', '?', '?', '?']]**

**1. Write a Program to Implement Breadth First Search using Python.**

graph = {

'1' : ['2','10'],

'2' : ['3','8'],

'3' : ['4'],

'4' : ['5','6','7'],

'5' : [],

'6' : [],

'7' : [],

'8' : ['9'],

'9' : [],

'10' : []

}

visited = []

queue = []

def bfs(visited, graph, node):

visited.append(node)

queue.append(node)

while queue:

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("Following is the Breadth-First Search")

bfs(visited, graph, '1')

**Output:**

Following is the Breadth-First Search

1 2 10 3 8 4 9 5 6 7

**PART B:**

**3. Write a program to demonstrate the working of the ID3 algorithm**

import ast

import csv

import math

import os

def load\_csv\_to\_header\_data(filename):

path = os.path.normpath(os.getcwd() + filename)

print(path)

fs = csv.reader(open(path))

all\_row = []

for r in fs:

all\_row.append(r)

headers = all\_row[0]

idx\_to\_name, name\_to\_idx = get\_header\_name\_to\_idx\_maps(headers)

data = { 'header': headers,'rows': all\_row[1:],'name\_to\_idx': name\_to\_idx,'idx\_to\_name': idx\_to\_name}

return data

def get\_header\_name\_to\_idx\_maps(headers):

name\_to\_idx = {}

idx\_to\_name = {}

for i in range(0, len(headers)):

name\_to\_idx[headers[i]] = i

idx\_to\_name[i] = headers[i]

return idx\_to\_name, name\_to\_idx

def project\_columns(data, columns\_to\_project):

data\_h = list(data['header'])

data\_r = list(data['rows'])

all\_cols = list(range(0,len(data\_h)))

columns\_to\_project\_ix = [data['name\_to\_idx'][name] for name in columns\_to\_project]

columns\_to\_remove = [cidx for cidx in all\_cols if cidx not in columns\_to\_project\_ix]

for delc in sorted(columns\_to\_remove, reverse=True):

del data\_h[delc]

for r in data\_r:

del r[delc]

idx\_to\_name, name\_to\_idx = get\_header\_name\_to\_idx\_maps(data\_h)

return {'header': data\_h, 'rows': data\_r,'name\_to\_idx': name\_to\_idx,'idx\_to\_name': idx\_to\_name}

def get\_uniq\_values(data):

idx\_to\_name = data['idx\_to\_name']

idxs = idx\_to\_name.keys()

val\_map = {}

for idx in iter(idxs):

val\_map[idx\_to\_name[idx]] = set()

for data\_row in data['rows']:

for idx in idx\_to\_name.keys():

att\_name = idx\_to\_name[idx]

val = data\_row[idx]

if val not in val\_map.values():

val\_map[att\_name].add(val)

return val\_map

def get\_class\_labels(data,target\_attribute):

rows = data['rows']

col\_idx = data['name\_to\_idx'][target\_attribute]

labels = {}

for r in rows:

val = r[col\_idx]

if val in labels:

labels[val] = labels[val] + 1

else:

labels[val] = 1

return labels

def entropy(n, labels):

ent = 0

for label in labels.keys():

p\_x = labels[label] / n

ent += - p\_x \* math.log(p\_x, 2)

return ent

def partition\_data(data, group\_att):

partitions = {}

data\_rows = data['rows']

partition\_att\_idx = data['name\_to\_idx'][group\_att]

for row in data\_rows:

row\_val = row[partition\_att\_idx]

if row\_val not in partitions.keys():

partitions[row\_val] = {'name\_to\_idx': data['name\_to\_idx'],'idx\_to\_name': data['idx\_to\_name'],'rows': list()}

partitions[row\_val]['rows'].append(row)

return partitions

def avg\_entropy\_w\_partitions(data, splitting\_att, target\_attribute): # find uniq values of splitting att

data\_rows = data['rows']

n = len(data\_rows)

partitions = partition\_data(data, splitting\_att)

avg\_ent = 0

for partition\_key in partitions.keys():

partitioned\_data = partitions[partition\_key]

partition\_n = len(partitioned\_data['rows'])

partition\_labels = get\_class\_labels(partitioned\_data, target\_attribute)

partition\_entropy = entropy(partition\_n, partition\_labels)

avg\_ent += partition\_n / n \* partition\_entropy

return avg\_ent, partitions

def most\_common\_label(labels):

mcl = max(labels, key=lambda k: labels[k])

return mcl

def id3(data, uniqs, remaining\_atts, target\_attribute):

labels = get\_class\_labels(data, target\_attribute)

node = {}

if len(labels.values()) == 1:

node['label'] = next(iter(labels.keys()))

return node

if len(remaining\_atts) == 0:

node['label'] = most\_common\_label(labels)

return node

n = len(data['rows'])

ent = entropy(n, labels)

max\_info\_gain = None

max\_info\_gain\_att = None

max\_info\_gain\_partitions = None

for remaining\_att in remaining\_atts:

avg\_ent, partitions = avg\_entropy\_w\_partitions(data, remaining\_att, target\_attribute)

info\_gain = ent - avg\_ent

if max\_info\_gain is None or info\_gain > max\_info\_gain:

max\_info\_gain = info\_gain

max\_info\_gain\_att = remaining\_att

max\_info\_gain\_partitions = partitions

if max\_info\_gain is None:

node['label'] = most\_common\_label(labels)

return node

node['attribute'] = max\_info\_gain\_att

node['nodes'] = {}

remaining\_atts\_for\_subtrees = set(remaining\_atts)

remaining\_atts\_for\_subtrees.discard(max\_info\_gain\_att)

uniq\_att\_values = uniqs[max\_info\_gain\_att]

for att\_value in uniq\_att\_values:

if att\_value not in max\_info\_gain\_partitions.keys():

node['nodes'][att\_value] = {'label': most\_common\_label(labels)}

continue

partition = max\_info\_gain\_partitions[att\_value]

node['nodes'][att\_value] = id3(partition, uniqs, remaining\_atts\_for\_subtrees, target\_attribute)

return node

def load\_config(config\_file):

with open(config\_file, 'r') as myfile:

data = myfile.read().replace('\n', '')

print(data)

return ast.literal\_eval(data)

def pretty\_print\_tree(root):

stack = []

rules = set()

def traverse(node, stack, rules):

if 'label' in node:

stack.append(' THEN ' + node['label'])

rules.add(''.join(stack))

stack.pop()

elif 'attribute' in node:

ifnd = 'IF ' if not stack else ' AND '

stack.append(ifnd + node['attribute'] + ' EQUALS ')

for subnode\_key in node['nodes']:

stack.append(subnode\_key)

traverse(node['nodes'][subnode\_key], stack, rules)

stack.pop()

stack.pop()

traverse(root, stack, rules)

print(os.linesep.join(rules))

def main():

argv ='tennis.cfg'

print("Command line args are {}: ".format(argv))

config = load\_config(argv)

print(config)

data = load\_csv\_to\_header\_data(config['data\_file'])

data = project\_columns(data, config['data\_project\_columns'])

target\_attribute = config['target\_attribute']

remaining\_attributes = set(data['header'])

remaining\_attributes.remove(target\_attribute)

print(remaining\_attributes)

uniqs = get\_uniq\_values(data)

root = id3(data, uniqs, remaining\_attributes, target\_attribute)

pretty\_print\_tree(root)

if \_\_name\_\_ == "\_\_main\_\_": main()

**OUTPUT:**

Command line args are tennis.cfg:

{ 'data\_file' : '//tennis.csv', 'data\_mappers' : [], 'data\_project\_columns' : ['Outlook', 'Temperature', 'Humidity', 'Windy', 'PlayTennis'], 'target\_attribute' : 'PlayTennis'}

{'data\_file': '//tennis.csv', 'data\_mappers': [], 'data\_project\_columns': ['Outlook', 'Temperature', 'Humidity', 'Windy', 'PlayTennis'], 'target\_attribute': 'PlayTennis'}

C:\Users\ADMIN\machine learning\tennis.csv

{'Outlook', 'Humidity', 'Temperature', 'Windy'}

IF Outlook EQUALS Sunny AND Humidity EQUALS Normal THEN Yes

IF Outlook EQUALS Overcast THEN Yes

IF Outlook EQUALS Rainy AND Windy EQUALS False THEN Yes

IF Outlook EQUALS Sunny AND Humidity EQUALS High THEN No

IF Outlook EQUALS Rainy AND Windy EQUALS True THEN No

7. Write a program implementation of the KNN algorithm

# importing libraries

import numpy as nm

import matplotlib.pyplot as mtp

import pandas as pd

#importing datasets

data\_set= pd.read\_csv('User\_Data.csv')

#Extracting Independent and dependent Variable

x= data\_set.iloc[:, [2,3]].values

y= data\_set.iloc[:, 4].values

# Splitting the dataset into training and test set.

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)

#feature Scaling

from sklearn.preprocessing import StandardScaler

st\_x= StandardScaler()

x\_train= st\_x.fit\_transform(x\_train)

x\_test= st\_x.transform(x\_test)

#Fitting K-NN classifier to the training set

from sklearn.neighbors import KNeighborsClassifier

classifier= KNeighborsClassifier(n\_neighbors=5, metric='minkowski',p=2)classifier.fit(x\_train, y\_train)

**OUTPUT :**

KNeighborsClassifier()

#Predicting the test set result

y\_pred= classifier.predict(x\_test)

#Creating the Confusion matrix

from sklearn.metrics import confusion\_matrix

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

print("Confusion Matrix : \n",cm)

**OUTPUT :**

Confusion Matrix :

[[64 4]

[ 3 29]]

#Visulaizing the trianing set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_train, y\_train

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01),nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),

alpha = 0.75, cmap = ListedColormap(('red','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

        c = ListedColormap(('red', 'green'))(i), label = j)

mtp.title('K-NN Algorithm (Training set)')

mtp.xlabel('Age')

mtp.ylabel('Estimated Salary')

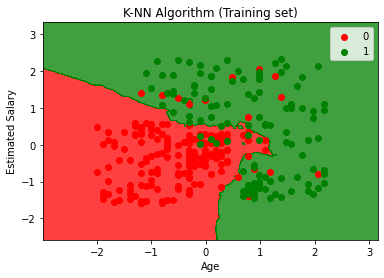
mtp.legend()

mtp.show()

**OUTPUT** :

\*c\* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with \*x\* & \*y\*. Please use the \*color\* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.

\*c\* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with \*x\* & \*y\*. Please use the \*color\* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.



#Visualizing the test set result

from matplotlib.colors import ListedColormap

x\_set, y\_set = x\_test, y\_test

x1, x2 = nm.meshgrid(nm.arange(start = x\_set[:, 0].min() - 1, stop = x\_set[:, 0].max() + 1, step  =0.01), nm.arange(start = x\_set[:, 1].min() - 1, stop = x\_set[:, 1].max() + 1, step = 0.01))

mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1.shape), alpha = 0.75, cmap = ListedColormap(('red','green' )))

mtp.xlim(x1.min(), x1.max())

mtp.ylim(x2.min(), x2.max())

for i, j in enumerate(nm.unique(y\_set)):

    mtp.scatter(x\_set[y\_set == j, 0], x\_set[y\_set == j, 1],

        c = ListedColormap(('red', 'green'))(i), label = j)

mtp.title('K-NN algorithm(Test set)')

mtp.xlabel('Age')

mtp.ylabel('Estimated Salary')

mtp.legend()

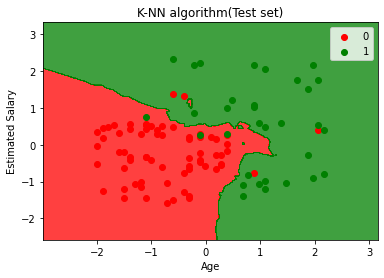
mtp.show()

from sklearn.metrics import accuracy\_score

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

**OUTPUT :** \*c\* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with \*x\* & \*y\*. Please use the \*color\* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.

\*c\* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with \*x\* & \*y\*. Please use the \*color\* keyword-argument or provide a 2-D array with a single row if you intend to specify the same RGB or RGBA value for all points.



Write a program to implement gradient boosting problem in python

Gradient boosting algorithm

from sklearn import datasets

from sklearn.preprocessing import StandardScaler

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

from sklearn.pipeline import make\_pipeline

from sklearn.ensemble import GradientBoostingRegressor

from sklearn.decomposition import PCA

from sklearn.metrics import mean\_squared\_error

bhp = datasets.load\_boston()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(bhp.data, bhp.target, random\_state=42, test\_size=0.1)

sc = StandardScaler()

X\_train\_std = sc.fit\_transform(X\_train)

X\_test\_std = sc.transform(X\_test)

gbr\_params = {'n\_estimators': 1000,

          'max\_depth': 3,

          'min\_samples\_split': 5,

          'learning\_rate': 0.01,

          'loss': 'ls'}

gbr = GradientBoostingRegressor(\*\*gbr\_params)

gbr.fit(X\_train\_std, y\_train)

print("Model Accuracy: %.3f" % gbr.score(X\_test\_std, y\_test))

mse = mean\_squared\_error(y\_test, gbr.predict(X\_test\_std))

print("The mean squared error (MSE) on test set: {:.4f}".format(mse))

output

Model Accuracy: 0.918

The mean squared error (MSE) on test set: 5.1449

import numpy as np

import matplotlib.pyplot as plt

from sklearn.inspection import permutation\_importance

feature\_importance = gbr.feature\_importances\_

sorted\_idx = np.argsort(feature\_importance)

pos = np.arange(sorted\_idx.shape[0]) + .5

fig = plt.figure(figsize=(8, 8))

plt.barh(pos, feature\_importance[sorted\_idx], align='center')

plt.yticks(pos, np.array(bhp.feature\_names)[sorted\_idx])

plt.title('Feature Importance (MDI)')

result = permutation\_importance(gbr, X\_test\_std, y\_test, n\_repeats=10,

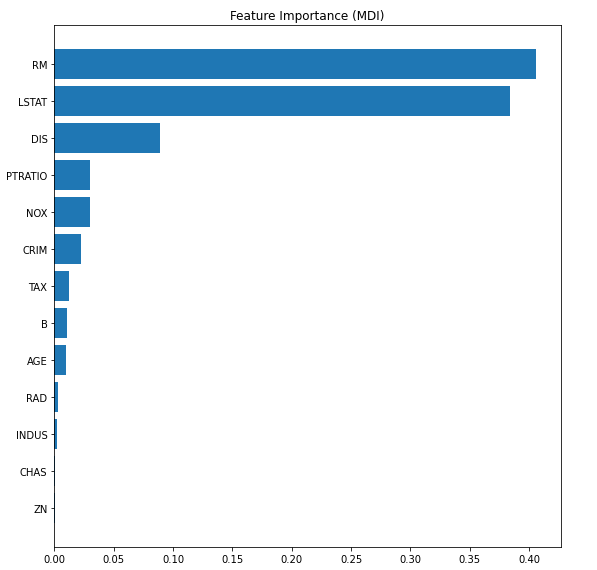
                                random\_state=42, n\_jobs=2)

sorted\_idx = result.importances\_mean.argsort()

fig.tight\_layout()

plt.show()

Output



test\_score = np.zeros((gbr\_params['n\_estimators'],), dtype=np.float64)

for i, y\_pred in enumerate(gbr.staged\_predict(X\_test\_std)):

    test\_score[i] = gbr.loss\_(y\_test, y\_pred)

fig = plt.figure(figsize=(8, 8))

plt.subplot(1, 1, 1)

plt.title('Deviance')

plt.plot(np.arange(gbr\_params['n\_estimators']) + 1, gbr.train\_score\_, 'b-',

         label='Training Set Deviance')

plt.plot(np.arange(gbr\_params['n\_estimators']) + 1, test\_score, 'r-',

         label='Test Set Deviance')

plt.legend(loc='upper right')

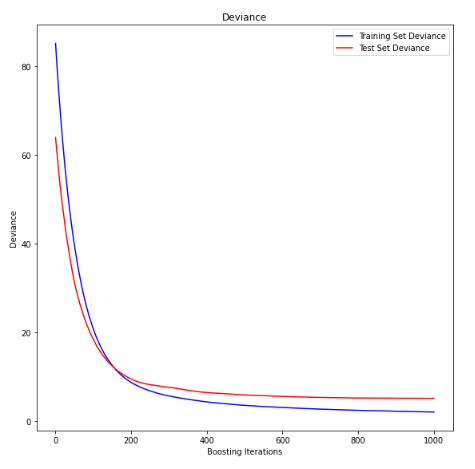
plt.xlabel('Boosting Iterations')

plt.ylabel('Deviance')

fig.tight\_layout()

plt.show()

Output



**2. write a program to implement BEST FIRST SEARCH**

from queue import PriorityQueue

import matplotlib.pyplot as plt

import networkx as nx

# for implementing BFS | returns path having lowest cost

def best\_first\_search(source, target, n):

    visited = [0] \* n

    visited[source] = True

    pq = PriorityQueue()

    pq.put((0, source))

    while pq.empty() == False:

        u = pq.get()[1]

        print(u, end=" ") # the path having lowest cost

        if u == target:

            break

        for v, c in graph[u]:

            if visited[v] == False:

                visited[v] = True

                pq.put((c, v))

    print()

# for adding edges to graph

def addedge(x, y, cost):

    graph[x].append((y, cost))

    graph[y].append((x, cost))

v = int(input("Enter the number of nodes: "))

graph = [[] for i in range(v)] # undirected Graph

e = int(input("Enter the number of edges: "))

print("Enter the edges along with their weights:")

for i in range(e):

    x, y, z = list(map(int, input().split()))

    addedge(x, y, z)

source = int(input("Enter the Source Node: "))

target = int(input("Enter the Target/Destination Node: "))

print("\nPath: ", end = "")

best\_first\_search(source, target, v)

**OUTPUT:**

Enter the number of nodes : 4

Enter the number of edges: 5

Enter the edges along with their weights:

0 1 1

0 2 1

0 3 2

2 3 2

1 3 3

Enter the source node:2

Enter the Target/Destination node:1

Path :2 0 1

**5.Write a program to implement water jug problem using python**

from collections import defaultdict

jug1, jug2, aim = 4, 3, 2

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

  if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

    print(amt1, amt2)

    return True

  if visited[(amt1, amt2)] == False:

    print(amt1, amt2)

    visited[(amt1, amt2)] = True

    return (waterJugSolver(0, amt2) or

        waterJugSolver(amt1, 0) or

        waterJugSolver(jug1, amt2) or

        waterJugSolver(amt1, jug2) or

        waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

        amt2 - min(amt2, (jug1-amt1))) or

        waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

        amt2 + min(amt1, (jug2-amt2))))

  else:

    return False

print("Steps: ")

waterJugSolver(0, 0)

**Output:**

**Steps:**

**0 0**

**4 0**

**4 3**

**0 3**

**3 0**

**3 3**

**4 2**

**0 2**

**True**

**6 . Write a program to implement K-means clustering using random samples**

from copy import deepcopy

import numpy as np # linear algebra

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

from matplotlib import pyplot as plt

# Set three centers, the model should predict similar results

center\_1 = np.array([1,1])

center\_2 = np.array([5,5])

center\_3 = np.array([8,1])

# Generate random data and center it to the three centers

data\_1 = np.random.randn(200, 2) + center\_1

data\_2 = np.random.randn(200,2) + center\_2

data\_3 = np.random.randn(200,2) + center\_3

data = np.concatenate((data\_1, data\_2, data\_3), axis = 0)

plt.scatter(data[:,0], data[:,1], s=7)

# Number of clusters

k = 3

# Number of training data

n = data.shape[0]

# Number of features in the data

c = data.shape[1]

# Generate random centers, here we use sigma and mean to ensure it represent the whole data

mean = np.mean(data, axis = 0)

std = np.std(data, axis = 0)

centers = np.random.randn(k,c)\*std + mean

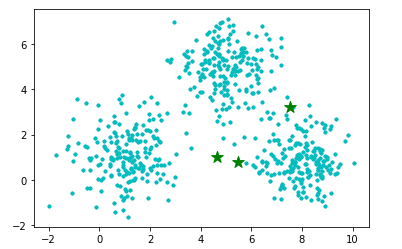
# Plot the data and the centers generated as random

plt.scatter(data[:,0], data[:,1], s=9,color='c')

plt.scatter(centers[:,0], centers[:,1], marker='\*', c='g', s=150)

plt.show()

**Output**



4. Program to implement Depth First Search

# Using a Python dictionary to act as an adjacency list

graph = {

&#39;5&#39; : [&#39;3&#39;,&#39;7&#39;],

&#39;3&#39; : [&#39;2&#39;, &#39;4&#39;],

&#39;7&#39; : [&#39;6&#39;],

&#39;6&#39;: [],

&#39;2&#39; : [&#39;1&#39;],

&#39;1&#39;:[],

&#39;4&#39; : [&#39;8&#39;],

&#39;8&#39; : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print(&quot;Following is the Depth-First Search&quot;)

dfs(visited, graph, &#39;5&#39;)

OUTPUT

Following is the Depth-First Search

5

3

2

1

4

8

7

6