1)

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {}

parents = {}

g[start\_node] = 0

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

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

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

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

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

g[m] = g[n] + weight

parents[m] = n

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 5,

'D': 7,

'E': 3,

'F': 6,

'G': 5,

'H': 3,

'I': 1,

'J': 0

}

return H\_dist[n]

Graph\_nodes = {

'A': [('B', 6), ('F', 3)],

'B': [('A', 6), ('C', 3), ('D', 2)],

'C': [('B', 3), ('D', 1), ('E', 5)],

'D': [('B', 2), ('C', 1), ('E', 8)],

'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],

'F': [('A', 3), ('G', 1), ('H', 7)],

'G': [('F', 1), ('I', 3)],

'H': [('F', 7), ('I', 2)],

'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],

}

aStarAlgo('A', 'J')

################################################################################################################################

2)

# Recursive implementation of AO\* aglorithm by Dr. K PARAMESHA, Professor, VVCE, Mysuru, INDIA

class Graph:

def \_\_init\_\_(self, graph, heuristicNodeList, startNode): #instantiate graph object with graph topology, heuristic values, start node

self.graph = graph

self.H=heuristicNodeList

self.start=startNode

self.parent={}

self.status={}

self.solutionGraph={}

def applyAOStar(self): # starts a recursive AO\* algorithm

self.aoStar(self.start, False)

def getNeighbors(self, v): # gets the Neighbors of a given node

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

def getStatus(self,v): # return the status of a given node

return self.status.get(v,0)

def setStatus(self,v, val): # set the status of a given node

self.status[v]=val

def getHeuristicNodeValue(self, n):

return self.H.get(n,0) # always return the heuristic value of a given node

def setHeuristicNodeValue(self, n, value):

self.H[n]=value # set the revised heuristic value of a given node

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): # Computes the Minimum Cost of child nodes of a given node v

minimumCost=0

costToChildNodeListDict={}

costToChildNodeListDict[minimumCost]=[]

flag=True

for nodeInfoTupleList in self.getNeighbors(v): # iterate over all the set of child node/s

cost=0

nodeList=[]

for c, weight in nodeInfoTupleList:

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

nodeList.append(c)

if flag==True: # initialize Minimum Cost with the cost of first set of child node/s

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s

flag=False

else: # checking the Minimum Cost nodes with the current Minimum Cost

if minimumCost>cost:

minimumCost=cost

costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child node/s

return minimumCost, costToChildNodeListDict[minimumCost] # return Minimum Cost and Minimum Cost child node/s

def aoStar(self, v, backTracking): # AO\* algorithm for a start node and backTracking status flag

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

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

print("PROCESSING NODE :", v)

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

if self.getStatus(v) >= 0: # if status node v >= 0, compute Minimum Cost nodes of v

minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)

self.setHeuristicNodeValue(v, minimumCost)

self.setStatus(v,len(childNodeList))

solved=True # check the Minimum Cost nodes of v are solved

for childNode in childNodeList:

self.parent[childNode]=v

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

solved=solved & False

if solved==True: # if the Minimum Cost nodes of v are solved, set the current node status as solved(-1)

self.setStatus(v,-1)

self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes which may be a part of solution

if v!=self.start: # check the current node is the start node for backtracking the current node value

self.aoStar(self.parent[v], True) # backtracking the current node value with backtracking status set to true

if backTracking==False: # check the current call is not for backtracking

for childNode in childNodeList: # for each Minimum Cost child node

self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)

self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false

h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1, 'T': 3}

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()

h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7} # Heuristic values of Nodes

graph2 = { # Graph of Nodes and Edges

'A': [[('B', 1), ('C', 1)], [('D', 1)]], # Neighbors of Node 'A', B, C & D with repective weights

'B': [[('G', 1)], [('H', 1)]], # Neighbors are included in a list of lists

'D': [[('E', 1), ('F', 1)]] # Each sublist indicate a "OR" node or "AND" nodes

}

G2 = Graph(graph2, h2, 'A') # Instantiate Graph object with graph, heuristic values and start Node

G2.applyAOStar() # Run the AO\* algorithm

G2.printSolution() # Print the solution graph as output of the AO\* algorithm search

################################################################################################################################

3)

import pandas as pd

import numpy as np

def learn(concepts,target):

specific\_h=concepts[0].copy()

general\_h=[["?" for i in range (len(specific\_h))]for i in range (len(specific\_h))]

for i,h in enumerate(concepts):

if target[i]=="yes":

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

if h[x]!=specific\_h[x]:

specific\_h[x]='?'

general\_h[x][x]='?'

else:

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

if h[x]!=specific\_h[x]:

general\_h[x][x]=specific\_h[x]

else:

general\_h[x][x]='?'

general\_h=[general\_h[i] for i,val in enumerate(general\_h) if val != ['?' for x in range(len(specific\_h))]]

return specific\_h,general\_h

data=pd.read\_csv('finds.csv')

concepts=np.array(data.iloc[:,0:-1])

target=np.array(data.iloc[:,-1])

s\_final,g\_final=learn(concepts,target)

print("final s:",s\_final)

print("final g:",g\_final)

data.head()

################################################################################################################################

4)

import math

import csv

class Node:

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

self.attribute=attribute

self.children=[]

self.answer=" "

def load\_csv(filename):

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

dataset=list(lines)

headers=dataset.pop(0)

return dataset,headers

def subtables(data,col,delete):

dic={}

coldata=[row[col] for row in data]

attr=list(set(coldata))

for k in attr:

dic[k]=[]

for y in range(len(data)):

key=data[y][col]

if delete:

del data[y][col]

dic[key].append(data[y])

return attr,dic

def entropy(s):

attr=list(set(s))

if len(attr)==1: # if all are +ve/-ve then entropy=0

return 0

counts=[0,0] # only two values posible 'yes' or 'no'

for i in range(2):

counts[i]=sum([1 for x in s if attr[i]==x])/(len(s)\*1.0)

sums=0

for cnt in counts:

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

return sums

def compute\_gain(data,col):

attvalues,dic = subtables(data,col,delete=False)

total\_entropy = entropy([row[-1] for row in data])

for x in range(len(attvalues)):

ratio=len(dic[attvalues[x]])/(len(data)\*1.0)

entro=entropy([row[-1] for row in dic[attvalues[x]]])

total\_entropy-=ratio\*entro

return total\_entropy

def build\_tree(data,features):

lastcol=[row[-1] for row in data]

if(len(set(lastcol)))==1: #if all the samples have same label reurn that label

node=Node(" ")

node.answer=lastcol[0]

return node

n=len(data[0])-1

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*",n,"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

gains=[compute\_gain(data,col)for col in range(n)]

split=gains.index(max(gains))

node = Node(features[split])

fea=features[:split]+features[split+1:]

attr,dic=subtables(data,split,delete=True)

for x in range(len(attr)):

child=build\_tree(dic[attr[x]],fea)

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

return node

def print\_tree(node,level):

if node.answer!=" ":

print(" "\*level,node.answer)

return

print(""\*level,node.attribute)

for value,n in node.children:

print(" "\*(level+1),value)

print\_tree(n,level+2)

def classify(node,x\_test,features):

if node.answer!=" ":

print(node.answer)

return

pos=features.index(node.attribute)

for value,n in node.children:

if x\_test[pos]==value:

classify(n,x\_test,features)

#main program

dataset,features=load\_csv("playtennis.csv")

node=build\_tree(dataset,features)

print("The decision tree for the dataset using id3 algorithm is")

print\_tree(node,0)

testdata,features=load\_csv("test\_tennis.csv")

for xtest in testdata:

print("the test instance:",xtest)

print("the predicted label:",end=" ")

classify(node,xtest,features)

################################################################################################################################

5)

import random

from math import exp

from random import seed

import matplotlib.pyplot as plt

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

network = list()

hidden\_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]

network.append(hidden\_layer)

output\_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range(n\_hidden + 1)]} for i in range(n\_outputs)]

network.append(output\_layer)

return network

def activate(weights, inputs):

activation = weights[-1]

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

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

return activation

def transfer(activation):

return 1.0 / (1.0 + exp(-activation))

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

def transfer\_derivative(output):

return output \* (1.0 - output)

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(expected[j] - neuron['output'])

for j in range(len(layer)):

neuron = layer[j]

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

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']

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('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l\_rate, sum\_error))

seed(1)

dataset = [[2.7810836,2.550537003,0],

[1.465489372,2.362125076,0],

[3.396561688,4.400293529,0],

[1.38807019,1.850220317,0],

[3.06407232,3.005305973,0],

[7.627531214,2.759262235,1],

[5.332441248,2.088626775,1],

[6.922596716,1.77106367,1],

[8.675418651,-0.242068655,1],

[7.673756466,3.508563011,1]]

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

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

network = initialize\_network(n\_inputs, 2, n\_outputs)

print(network)

train\_network(network, dataset, 0.5,20, n\_outputs)

for layer in network:

print(layer)

################################################################################################################################

6)

from sklearn import datasets

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

from sklearn.naive\_bayes import GaussianNB

iris=datasets.load\_iris()

print("Iris dataset loaded...")

X\_train,X\_test,y\_train,y\_test=train\_test\_split(iris.data,iris.target,test\_size=0.2,random\_state=70)

print("Data set is split into training and testing ...")

print("Size of training data and its lable",X\_train.shape,y\_train.shape)

print("Size of testining data and its lable",X\_test.shape,y\_test.shape)

for i in range(len(iris.target\_names)):

print("Lable",i,"-",str(iris.target\_names[i]))

gnb=GaussianNB()

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

print("Results of classification using Naive Bayes")

for r in range(0,len(X\_test)):

print("sample : ",str(X\_test[r]),"Actual\_Lable :",str(y\_test[r]),"Predicted \_Lable :",str(y\_pred[r]))

print("classification accuracy:",gnb.score(X\_test,y\_test))

################################################################################################################################

7)

from sklearn.cluster import KMeans

from sklearn.mixture import GaussianMixture

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn import datasets

iris = datasets.load\_iris()

X = pd.DataFrame(iris.data,)

X.columns = ['Sepal\_Length','Sepal\_Width','Petal\_Length','Petal\_Width']

y = pd.DataFrame(iris.target)

y.columns = ['Targets']

colormap=np.array(['red','lime','black'])

plt.figure(figsize=(14,7))

plt.subplot(1,3,1)

plt.title('Real')

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y.Targets])

plt.subplot(1,3,2)

model=KMeans(n\_clusters=3).fit(X)

plt.title('KMeans')

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[model.labels\_])

gmm=GaussianMixture(n\_components=3)

y\_cluster\_gmm=gmm.fit(X).predict(X)

plt.subplot(1,3,3)

plt.title('GMM Classification')

plt.scatter(X.Petal\_Length,X.Petal\_Width,c=colormap[y\_cluster\_gmm])

print('Observation : The GMM using EM algorithm based clustering matched the True label more closely than the K-MEANS')

################################################################################################################################

8)

from sklearn import datasets

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

from sklearn.neighbors import KNeighborsClassifier

iris=datasets.load\_iris()

print("Iris dataset loaded...")

x\_train,x\_test,y\_train,y\_test=train\_test\_split(iris.data,iris.target,test\_size=0.2)

print("Dataset is split into training and testing...")

print("Size of training data and its label",x\_train.shape,y\_train.shape)

print("size of testing data and its label",x\_test.shape,y\_test.shape)

for i in range(len(iris.target\_names)):

print("Label",i,"-",str(iris.target\_names[i]))

classifier=KNeighborsClassifier(n\_neighbors=1)

y\_pred=classifier.fit(x\_train,y\_train).predict(x\_test)

print("Results of Classification using K-nn with K=1")

for r in range(len(x\_test)):

print("Sample:",str(x\_test[r]),"Actual-label:",str(y\_test[r]),"Predicted-label:",str(y\_pred[r]))

print("Classification accuracy:",classifier.score(x\_test,y\_test));

################################################################################################################################

9)

import numpy as np

import pandas as pd

from sklearn.datasets import load\_boston

import matplotlib.pyplot as plt

def h(x,a,b):

return a\*x + b

def error(a,x,b,y,w):

e = 0

m = len(x)

for i in range(m):

e += np.power(h(x[i],a,b)-y[i],2)\*w[i]

return (e/(2\*m))

def step\_gradient(a,x,b,y,learning\_rate,w):

grad\_a = 0

grad\_b = 0

m = len(x)

for i in range(m):

grad\_a += (2/m)\*((h(x[i],a,b)-y[i])\*x[i])\*w[i]

grad\_b += (2/m)\*(h(x[i],a,b)-y[i])\*w[i]

a = a - (grad\_a \* learning\_rate)

b = b - (grad\_b \* learning\_rate)

return a,b

def descend(initial\_a, initial\_b, x, y, learning\_rate, iterations,w):

a = initial\_a

b = initial\_b

for i in range(iterations):

e = error(a,x,b,y,w)

if i%1000 == 0:

print(f"Error: {e}-- a:{a}, b:{b}")

a, b = step\_gradient(a,x,b,y, learning\_rate,w)

return a,b

boston = load\_boston()

features = pd.DataFrame(boston.data, columns=boston.feature\_names)

x = features['RM']

X1 = sorted(np.array(x/x.mean()))

X=X1+[i+1 for i in X1]

Y=np.sin(X)

plt.plot(X,Y,'g.')

n = int(0.8 \* len(X))

x\_train = X[:n]

y\_train = Y[:n]

x\_test = X[n:]

y\_test = Y[n:]

w=np.exp([-(1.2-i)\*\*2/(0.2) for i in x\_train])

plt.plot(x\_train, y\_train,'r.',x\_train,w,'b.')

a = 1.8600662368042573

b = -0.7962243178421666

learning\_rate = 0.01

iterations = 10000

final\_a, final\_b = descend(a,b,x\_train,y\_train, learning\_rate, iterations,w)

H=[i\*final\_a+final\_b for i in x\_train]

plt.plot(x\_train, y\_train,'r.',x\_train, H,'b')

print(error(a,x\_test,b,y\_test,w))

print(error(final\_a,x\_test, final\_b,y\_test,w))

plt.plot(x\_test,y\_test,'b.',x\_train,y\_train,'r.')