**LAB**

1. **1. A-star.**

**Sol-**

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

**open\_set = set(start\_node) # {A}, len{open\_set}=1**

**closed\_set = set()**

**g = {} # store the distance from starting node**

**parents = {}**

**g[start\_node] = 0**

**parents[start\_node] = start\_node # parents['A']='A"**

**while len(open\_set) > 0 :**

**n = None**

**for v in open\_set: # v='B'/'F'**

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

**n = v # n='A'**

**if n == stop\_node or Graph\_nodes[n] == None:**

**pass**

**else:**

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

**# nodes 'm' not in first and last set are added to first**

**# n is set its parent**

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

**open\_set.add(m) # m=B weight=6 {'F','B','A'} len{open**

**parents[m] = n # parents={'A':A,'B':A} len{parent}=2**

**g[m] = g[n] + weight # g={'A':0,'B':6, 'F':3} len{g}=2**

**else:**

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

**#update g(m)**

**g[m] = g[n] + weight**

**#change parent of m to n**

**parents[m] = n**

**if m in closed\_set:**

**closed\_set.remove(m)**

**open\_set.add(m)**

**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**

**open\_set.remove(n)# {'F','B'} len=2**

**closed\_set.add(n) #{A} len=1**

**print('Path does not exist!')**

**return None**

**def get\_neighbors(v):**

**if v in Graph\_nodes:**

**return Graph\_nodes[v]**

**else:**

**return None**

**def heuristic(n):**

**H\_dist = {**

**'A': 10,**

**'B': 8,**

**'C': 5,**

**'D': 7,**

**'E': 3,**

**'F': 6,**

**'G': 5,**

**'H': 3,**

**'I': 1,**

**'J': 0**

**}**

**return H\_dist[n]**

**#Describe your graph here**

**Graph\_nodes = {**

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

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

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

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

**'E': [('I', 5), ('J', 5)],**

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

**'G': [('I', 3)],**

**'H': [('I', 2)],**

**'I': [('E', 5), ('J', 3)],**

**}**

**aStarAlgo('A', 'J')**

**OP - Path found: ['A', 'F', 'G', 'I', 'J']**

**['A', 'F', 'G', 'I', 'J']**

**------------------------------------------------------------------------------------------------2. AO-star**

**Sol - 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)**

**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:**

**status as solved(-1)**

**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)**

**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 = {**

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

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

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

**}**

**G2 = Graph(graph2, h2, 'A')**

**G2.applyAOStar()**

**G2.printSolution()**

**Output - {'E': [], 'F': [], 'D': ['E', 'F'], 'A': ['D']}**

**3. Candidate-Elimination**

**import csv**

**with open("trainingexamples.csv") as f:**

**csv\_file = csv.reader(f)**

**data = list(csv\_file)**

**specific = data[0][:-1]**

**general = [['?' for i in range(len(specific))] for j in range(len(specific))]**

**for i in data:**

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

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

**if i[j] != specific[j]:**

**specific[j] = "?"**

**general[j][j] = "?"**

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

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

**if i[j] != specific[j]:**

**general[j][j] = specific[j]**

**else:**

**general[j][j] = "?"**

**print("\nStep " + str(data.index(i)+1) + " of Candidate Elimination Algorithm")**

**print(specific)**

**print(general)**

**gh = [] # gh = general Hypothesis**

**for i in general:**

**for j in i:**

**if j != '?':**

**gh.append(i)**

**break**

**print("\nFinal Specific hypothesis:\n", specific)**

**print("\nFinal General hypothesis:\n", gh)**

**Table**

**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**

**Op- Final Specific hypothesis:**

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

**Final General hypothesis:**

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

**4. ID3.**

**Sol import pandas as pd**

**from pprint import pprint**

**from sklearn.feature\_selection import mutual\_info\_classif**

**from collections import Counter**

**def id3 (df, target\_attribute, attribute\_names, default\_class=None):**

**cnt=Counter(x for x in df[target\_attribute])**

**if len(cnt)==1:**

**return next(iter(cnt))**

**elif df.empty or (not attribute\_names):**

**return default\_class**

**else:**

**gainz = mutual\_info\_classif (df[attribute\_names],df[target\_attribute],discrete\_features=True)**

**index\_of\_max=gainz.tolist().index(max(gainz))**

**best\_attr=attribute\_names[index\_of\_max]**

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

**remaining\_attribute\_names=**

**[i for i in attribute\_names if i!=best\_attr]**

**for attr\_val, data\_subset in df.groupby(best\_attr):**

**subtree=id3(data\_subset, target\_attribute, remaining\_attribute\_names,default\_class)**

**tree[best\_attr][attr\_val]=subtree**

**return tree**

**df=pd.read\_csv("p-tennis.csv")**

**attribute\_names=df.columns.tolist()**

**print("List of attribut name")**

**attribute\_names.remove("PlayTennis")**

**for colname in df.select\_dtypes("object"):**

**df[colname], \_ = df[colname].factorize()**

**print(df)**

**tree= id3(df,"PlayTennis", attribute\_names)**

**print("The tree structure")**

**pprint(tree)**

**op- The tree structure**

**{'Outlook': {0: {'Humidity': {0: 0, 1: 1}},**

**1: 1,**

**2: {'Windy': {False: 1, True: 0}}}}**

**5. BackPropagation**

**Sol - import numpy as np**

**inputNeurons=2**

**hiddenlayerNeurons=4**

**outputNeurons=2**

**iteration=6000**

**input = np.random.randint(1,5,inputNeurons)**

**output = np.array([1.0,0.0])**

**hidden\_layer=np.random.rand(1,hiddenlayerNeurons)**

**hidden\_biass=np.random.rand(1,hiddenlayerNeurons)**

**output\_bias=np.random.rand(1,outputNeurons)**

**hidden\_weights=np.random.rand(inputNeurons,hiddenlayerNeurons)**

**output\_weights=np.random.rand(hiddenlayerNeurons,outputNeurons)**

**def sigmoid (layer):**

**return 1/(1 + np.exp(-layer))**

**def gradient(layer):**

**return layer\*(1-layer)**

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

**hidden\_layer=np.dot(input,hidden\_weights)**

**hidden\_layer=sigmoid(hidden\_layer+hidden\_biass)**

**output\_layer=np.dot(hidden\_layer,output\_weights)**

**output\_layer=sigmoid(output\_layer+output\_bias)**

**error = (output-output\_layer)**

**gradient\_outputLayer=gradient(output\_layer)**

**error\_terms\_output=gradient\_outputLayer \* error**

**error\_terms\_hidden=gradient(hidden\_layer)\*np.dot(error\_terms\_output,output\_weights.T)**

**gradient\_hidden\_weights = np.dot(input.reshape(inputNeurons,1),error\_terms\_hidden.reshape(1,hiddenlayerNeurons))**

**gradient\_ouput\_weights = np.dot(hidden\_layer.reshape(hiddenlayerNeurons,1),error\_terms\_output.reshape(1,outputNeurons))**

**hidden\_weights = hidden\_weights + 0.05\*gradient\_hidden\_weights**

**output\_weights = output\_weights + 0.05\*gradient\_ouput\_weights**

**if i<50 or i>iteration-50:**

**print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")**

**print("iteration:",i,"::::",error)**

**print("###output########",output\_layer)**

**op -**

**iteration: 0 :::: [[ 0.0720921 -0.94906383]]**

**###output######## [[0.9279079 0.94906383]]**

**iteration: 1 :::: [[ 0.07203592 -0.94865304]]**

**###output######## [[0.92796408 0.94865304]]**

**6. A-NaiveBayes**

**Sol import pandas as pd**

**from sklearn import tree**

**from sklearn.preprocessing import LabelEncoder**

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

**# Load Data from CSV**

**data = pd.read\_csv('p-tennis.csv')**

**print("The first 5 Values of data is :\n", data.head())**

**# obtain train data and train output**

**X = data.iloc[:, :-1]**

**print("\nThe First 5 values of the train data is\n", X.head())**

**y = data.iloc[:, -1]**

**print("\nThe First 5 values of train output is\n", y.head())**

**le\_outlook = LabelEncoder()**

**X.Outlook = le\_outlook.fit\_transform(X.Outlook)**

**le\_Temperature = LabelEncoder()**

**X.Temperature = le\_Temperature.fit\_transform(X.Temperature)**

**le\_Humidity = LabelEncoder()**

**X.Humidity = le\_Humidity.fit\_transform(X.Humidity)**

**le\_Windy = LabelEncoder()**

**X.Windy = le\_Windy.fit\_transform(X.Windy)**

**print("\nNow the Train output is\n", X.head())**

**le\_PlayTennis = LabelEncoder()**

**y = le\_PlayTennis.fit\_transform(y)**

**print("\nNow the Train output is\n",y)**

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

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y, test\_size = 0.20)**

**classifier = GaussianNB()**

**classifier.fit(X\_train, y\_train)**

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

**print("Accuracy is:", accuracy\_score(classifier.predict(X\_test), y\_test))**

**table-**

**Outlook,Temperature,Humidity,Windy,PlayTennis**

**Sunny,Hot,High,False,No**

**Sunny,Hot,High,True,No**

**Overcast,Hot,High,False,Yes**

**Rainy,Mild,High,False,Yes**

**Rainy,Cool,Normal,False,Yes**

**Rainy,Cool,Normal,True,No**

**Overcast,Cool,Normal,True,Yes**

**Sunny,Mild,High,False,No**

**Sunny,Cool,Normal,False,Yes**

**Rainy,Mild,Normal,False,Yes**

**Sunny,Mild,Normal,True,Yes**

**Overcast,Mild,High,True,Yes**

**Overcast,Hot,Normal,False,Yes**

**Rainy,Mild,High,True,No**

**Op- Now the Train output is**

**[0 0 1 1 1 0 1 0 1 1 1 1 1 0]**

**Accuracy is: 0.6666666666666666**

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

**7. S-EM-Kmeans**

**Sol –**

**import matplotlib.pyplot as plt**

**from sklearn import datasets**

**from sklearn.cluster import KMeans**

**from sklearn.mixture import GaussianMixture**

**import sklearn.metrics as sm**

**import pandas as pd**

**import numpy as np**

**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']**

**print(X)**

**print(Y)**

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

**plt.subplot(1,2,1)**

**plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[Y.Targets], s=40)**

**plt.title('Real Clustering')**

**model1 = KMeans(n\_clusters=3)**

**model1.fit(X)**

**plt.subplot(1,2,2)**

**plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[model1.labels\_], s=40)**

**plt.title('K Mean Clustering')**

**plt.show()**

**model2 = GaussianMixture(n\_components=3)**

**model2.fit(X)**

**plt.subplot(1,2,1)**

**plt.scatter(X.Petal\_Length, X.Petal\_Width, c=colormap[model2.predict(X)], s=40)**

**plt.title('EM Clustering')**

**plt.show()**

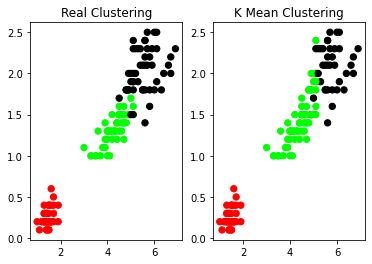
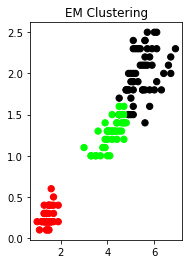
**print("Actual Target is:\n", iris.target)**

**print("K Means:\n",model1.labels\_)**

**print("EM:\n",model2.predict(X))**

**print("Accuracy of KMeans is ",sm.accuracy\_score(Y,model1.labels\_))**

**print("Accuracy of EM is ",sm.accuracy\_score(Y, model2.predict(X)))**

** **

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

**8. S-KNN.**

**Sol-**

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

**from sklearn.neighbors import KNeighborsClassifier**

**from sklearn import datasets**

**iris=datasets.load\_iris()**

**print("Iris Data set loaded...")**

**x\_train, x\_test, y\_train, y\_test = train\_test\_split(iris.data,iris.target,test\_size=0.1)**

**#random\_state=0**

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

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

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

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

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

**print("Results of Classification using K-nn with K=5 ")**

**for r in range(0,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));**

**op - Iris Data set loaded...**

**Label 0 - setosa**

**Label 1 - versicolor**

**Label 2 - virginica**

**Results of Classification using K-nn with K=5**

**Sample: [6.8 3.2 5.9 2.3] Actual-label: 2 Predicted-label: 2**

**Classification Accuracy : 0.9333333333333333**

**Sample: [5.8 2.6 4. 1.2] Actual-label: 1 Predicted-label: 1**

**Table gitub code iris table**

**9. S-LinearRegression**

**Sol - import numpy as np**

**import matplotlib.pyplot as plt**

**x = np.linspace(-5, 5, 1000)**

**y = np.log(np.abs((x \*\* 2) - 1) + 0.5)**

**x = x + np.random.normal(scale=0.05, size=1000)**

**plt.scatter(x, y, alpha=0.3)**

**def local\_regression(x0, x, y, tau):**

**x0 = np.r\_[1, x0]**

**x = np.c\_[np.ones(len(x)), x]**

**xw =x.T \* radial\_kernel(x0, x, tau)**

**beta = np.linalg.pinv(xw @ x) @ xw @ y**

**return x0 @ beta**

**def radial\_kernel(x0, x, tau):**

**return np.exp(np.sum((x - x0) \*\* 2, axis=1) / (-2 \* tau \*\* 2))**

**def plot\_lr(tau):**

**domain = np.linspace(-5, 5, num=500)**

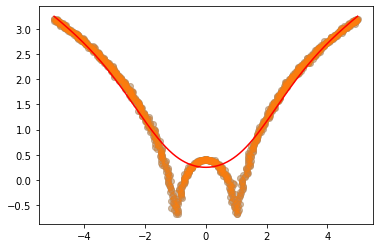
**pred = [local\_regression(x0, x, y, tau) for x0 in domain]**

**plt.scatter(x, y, alpha=0.3)**

**plt.plot(domain, pred, color="red")**

**return plt**

**plot\_lr(1).show()**

**op - **