**from collections import defaultdict graph=defaultdict(list)**

**def addEdge(u,v):**

**graph[u].append(v)**

**def dfid(start,goal,max\_depth):**

**print("Start node us: ",start," Goal node is :",goal) for i in range(max\_depth):**

**print("At level :",i+1)**

**print("Path taken is: ",end=" ") isfound=dfs(start,goal,i)**

**if isfound:**

**print("\nNode found !\n") return**

**else:**

**print("\nNode not found!\n") def dfs(start,goal,depth):**

**print(start,end=" ") if start==goal:**

**return True if depth<=0:**

**return**

**for i in graph[start]:**

**if dfs(i,goal,depth-1): return True**

**return False**

**goal=defaultdict(list) addEdge('A','B')**

**addEdge('A','C')**

**addEdge('A','D')**

**addEdge('C','E')**

**addEdge('C','F')**

**addEdge('D','G')**

**addEdge('D','H')**

**addEdge('G','I')**

**addEdge('H','K')**

**addEdge('H','L')**

**addEdge('I','J')**

**addEdge('K','O')**

**addEdge('L','M')**

**addEdge('M','N')**

**dfid('A','O',4)**

**SuccList = {**

**'S': [['A', 3], ['B', 6], ['C', 5]],**

**'A': [['E', 8], ['D', 9]],**

**'B': [['G', 14], ['F', 12]],**

**'C': [['H', 7]],**

**'H': [['J', 6], ['I', 5]],**

**'I': [['M', 2], ['L', 10], ['K', 1]]**

**}**

**def best\_first\_search(start, goal): open\_list = [[start, 5]]**

**closed\_list = [] i = 1**

**while open\_list:**

**print(f"\n<<<<<<<<<---({i})-->>>>>>>>>\n")**

**n = open\_list.pop(0) closed\_list.append(n) print(f"N= {n}")**

**print(f"CLOSED= {closed\_list}")**

**if n[0] == goal:**

**return closed\_list, True**

**children = [child for child in SuccList.get(n[0], []) if child not in open\_list and child not in closed\_list]**

**print(f"CHILD= {children}")**

**open\_list.extend(children)**

**print(f"Unsorted OPEN= {open\_list}") open\_list.sort(key=lambda x: x[1])**

**print(f"Sorted OPEN= {open\_list}") i += 1**

**return closed\_list, False**

**start = input("Enter Source node >> ").upper() goal = input("Enter Goal node >> ").upper()**

**path, found = best\_first\_search(start, goal)**

**print("Best First Search Path >>>>>", path, "<<<<", found)**

**import heapq**

**def astar(graph, start, goal, heuristic):**

**queue = [(0 + heuristic[start], start, [])] # (priority, node, path) cost\_so\_far = {start: 0}**

**iteration\_step = 0 while queue:**

**iteration\_step += 1**

**# Get the node to explore next**

**\_, current, path = heapq.heappop(queue)**

**print("Iteration", iteration\_step, "- Current node:", current, "- Path:", path) if current == goal:**

**return path + [current], iteration\_step for neighbor, cost in graph[current].items():**

**new\_cost = cost\_so\_far[current] + cost**

**# Update cost if the new path is cheaper or this is the first time visiting the node if neighbor not in cost\_so\_far or new\_cost < cost\_so\_far[neighbor]:**

**cost\_so\_far[neighbor] = new\_cost priority = new\_cost + heuristic[neighbor]**

**heapq.heappush(queue, (priority, neighbor, path + [current]))**

**print(" -> Neighbor:", neighbor, "- Path:", path + [current], "- Cost:", new\_cost, "- Priority:",**

**priority)**

**return None, iteration\_step graph = {**

**'A': {'B': 1, 'C': 3},**

**'B': {'D': 3, 'E': 4},**

**'C': {'F': 2}, 'D': {},**

**'E': {'G': 5}, 'F': {},**

**'G': {}**

**}**

**heuristic = { 'A': 10,**

**'B': 5,**

**'C': 8,**

**'D': 4,**

**'E': 3,**

**'F': 2,**

**'G': 0**

**}**

**start\_node = 'A' goal\_node = 'G'**

**path, iterations = astar(graph, start\_node, goal\_node, heuristic) if path:**

**print("Path found:", path)**

**total\_cost = sum(graph[path[i]][path[i+1]] for i in range(len(path)-1)) print("Total cost:", total\_cost)**

**else:**

**print("No path found.") print("Iterations:", iterations)**

**import math**

**def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth): # Base case: targetDepth reached**

**if curDepth == targetDepth: return scores[nodeIndex]**

**if maxTurn:**

**left = minimax(curDepth + 1, nodeIndex \* 2, False, scores, targetDepth) right = minimax(curDepth + 1, nodeIndex \* 2 + 1, False, scores, targetDepth) print("Maximizing node at depth", curDepth, "with value", max(left, right)) return max(left, right)**

**else:**

**left = minimax(curDepth + 1, nodeIndex \* 2, True, scores, targetDepth) right = minimax(curDepth + 1, nodeIndex \* 2 + 1, True, scores, targetDepth) print("Minimizing node at depth", curDepth, "with value", min(left, right)) return min(left, right)**

**# Driver code**

**scores = [5, 2, 1, 3, 6, 2, 0, 7]**

**treeDepth = math.floor(math.log(len(scores), 2)) # Use floor to ensure integer depth print("The optimal value is:", minimax(0, 0, True, scores, treeDepth))**

import csv a = []

with open('enjoysport.csv', 'r') as csvfile:

next(csvfile) # Skip the header for row in csv.reader(csvfile):

a.append(row)

print(a)

print("\nThe total number of training instances are: ", len(a)) num\_attribute = len(a[0]) - 1

hypothesis = ['0'] \* num\_attribute

print("\nThe initial hypothesis is: ", hypothesis)

for i in range(len(a)):

if a[i][num\_attribute] == 'yes':

print("\nInstance", i + 1, "is", a[i], "and is a Positive Instance") for j in range(num\_attribute):

if hypothesis[j] == '0' or hypothesis[j] == a[i][j]:

hypothesis[j] = a[i][j] else:

hypothesis[j] = '?'

print("The hypothesis for the training instance", i + 1, "is: ", hypothesis, "\n") else:

print("\nInstance", i + 1, "is", a[i], "and is a Negative Instance Hence Ignored") print("The hypothesis for the training instance", i + 1, "is: ", hypothesis, "\n")

print("\nThe Maximally specific hypothesis for the training instance is: ", hypothesis) # OR Gate

def OR():

w1 = 0

w2 = 0

a = 0.2

t = 0

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

Y = [0, 1, 1, 1]

while True:

Out = [] count = 0 for i in X:

step = (w1 \* i[0] + w2 \* i[1]) if step <= t:

O = 0

else:

O = 1

if O == Y[count]:

Out.append(O) count += 1

else:

if step <= t:

w1 += a \* i[0] w2 += a \* i[1]

print(w1, w2) print(" >")

if Out == Y:

print("Final Output of OR ::\n")

print("Weights: w1={} and w2={} >>>> {}".format(w1, w2, Out)) break

OR()

def AND():

w1 = 0

w2 = 0

a = 0.2

t = 1

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

Y = [0, 0, 0, 1]

while True:

Out = [] count = 0 for i in X:

step = (w1 \* i[0] + w2 \* i[1]) if step < t:

O = 0

else:

O = 1

if O == Y[count]:

Out.append(O) count += 1

else:

print('Weights changed to..') w1 += a \* i[0]

w2 += a \* i[1]

print("w1={} w2={}".format(round(w1, 2), round(w2, 2))) print(w1, w2, Out)

print(" >")

if Out == Y:

print("\nFinal Output of AND::\n")

print("Weights: w1={} and w2={} >>>> {}".format(round(w1, 2), round(w2, 2), Out)) break

AND()

def NOT():

X = [0, 1]

Y = [1, 0]

weight = -1

bias = 1 Out = [] for i in X:

j = weight \* i + bias if j >= 0:

Out.append(1) else:

Out.append(0)

print("\nFinal Output of NOT ::\n") for i in range(len(X)):

print("NOT Gate {}--> {}".format(X[i], Out[i])) NOT()

import numpy as np

def sigmoid(x):

return 1 / (1 + np.exp(-x)) def sigmoid\_derivative(x):

return x \* (1 - x)

inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

expected\_output = np.array([[0], [1], [1], [0]])

epochs = 10000

lr = 0.5

inputLayerNeurons, hiddenLayerNeurons, outputLayerNeurons = 2, 2, 1

# Random weights and bias initialization

hidden\_weights = np.random.uniform(size=(inputLayerNeurons, hiddenLayerNeurons)) hidden\_bias = np.random.uniform(size=(1, hiddenLayerNeurons))

output\_weights = np.random.uniform(size=(hiddenLayerNeurons, outputLayerNeurons)) output\_bias = np.random.uniform(size=(1, outputLayerNeurons))

print("Initial hidden weights: ", end='') print(\*hidden\_weights)

print("Initial hidden biases: ", end='') print(\*hidden\_bias)

print("Initial output weights: ", end='') print(\*output\_weights)

print("Initial output biases: ", end='') print(\*output\_bias)

for \_ in range(epochs):

hidden\_layer\_activation = np.dot(inputs, hidden\_weights) hidden\_layer\_activation += hidden\_bias hidden\_layer\_output = sigmoid(hidden\_layer\_activation)

output\_layer\_activation = np.dot(hidden\_layer\_output, output\_weights) output\_layer\_activation += output\_bias

predicted\_output = sigmoid(output\_layer\_activation)

error = expected\_output - predicted\_output

d\_predicted\_output = error \* sigmoid\_derivative(predicted\_output)

error\_hidden\_layer = d\_predicted\_output.dot(output\_weights.T)

d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_layer\_output)

output\_weights += hidden\_layer\_output.T.dot(d\_predicted\_output) \* lr output\_bias += np.sum(d\_predicted\_output, axis=0, keepdims=True) \* lr hidden\_weights += inputs.T.dot(d\_hidden\_layer) \* lr

hidden\_bias += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* lr

print("Final hidden weights: ", end='') print(\*hidden\_weights)

print("Final hidden bias: ", end='') print(\*hidden\_bias)

print("Final output weights: ", end='') print(\*output\_weights)

print("Final output bias: ", end='') print(\*output\_bias)

print("\nOutput from neural network after epochs :" + str(epochs)) print(\*predicted\_output)

x1=[1,1]

x2=[1,-1]

x3=[-1,1]

x4=[-1,-1]

xilist=[x1,x2,x3,x4] y=[1,-1,-1,-1]

**w1=w2=bw=0 b=1**

**def heb\_learn(): global w1,w2,bw**

**print("dw1\tdw2\tdb\tw1\tw2\t") i=0**

**for xi in xilist: dw1=xi[0]\*y[i] dw2=xi[1]\*y[i] db=y[i] w1=w1+dw1 w2=w2+dw2 bw+=db**

**print(dw1,dw2,db,w1,w2,bw,sep='\t') i+=1**

**print("Learning...") heb\_learn()**

**print("Learning completed")**

**print("Output of AND gate using obtained w1,w2,bw") print("x1\tx2\ty")**

**for xi in xilist:**

**print(xi[0],xi[1],1 if w1\*xi[0]+w2\*xi[1]+b\*bw>0 else -1,sep='\t') print("Final weights are: w1= "+str(w1) + " w2= " +str(w2))**

**import numpy as np**

**import matplotlib.pyplot as plt def euclidean\_distance(a, b):**

**return np.sqrt(np.sum((a - b) \*\* 2)) def initialize\_centroids(X, k):**

**indices = np.random.choice(X.shape[0], k, replace=False) return X[indices]**

**def assign\_clusters(X, centroids): clusters = []**

**for x in X:**

**distances = [euclidean\_distance(x, centroid) for centroid in centroids] cluster = np.argmin(distances)**

**clusters.append(cluster) return np.array(clusters)**

**def update\_centroids(X, clusters, k): new\_centroids = []**

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

**cluster\_points = X[clusters == i]**

**if len(cluster\_points) == 0: # If a cluster is empty, reinitialize its centroid randomly new\_centroid = X[np.random.choice(X.shape[0])]**

**else:**

**new\_centroid = np.mean(cluster\_points, axis=0) new\_centroids.append(new\_centroid)**

**return np.array(new\_centroids)**

**def kmeans(X, k, max\_iters=100, tol=1e-4): centroids = initialize\_centroids(X, k)**

**for \_ in range(max\_iters):**

**clusters = assign\_clusters(X, centroids) new\_centroids = update\_centroids(X, clusters, k) if np.all(np.abs(new\_centroids - centroids) <= tol):**

**break**

**centroids = new\_centroids return centroids, clusters**

**# Generate sample data np.random.seed(42)**

**X = np.vstack([np.random.randn(100, 2) + np.array([3, 3]), np.random.randn(100, 2) + np.array([-3, -3]),**

**np.random.randn(100, 2) + np.array([-3, 3]),**

**np.random.randn(100, 2) + np.array([3, -3])])**

**# Apply K-means algorithm k = 4**

**centroids, clusters = kmeans(X, k)**

**# Plotting the sample data plt.figure(figsize=(8, 6))**

**plt.scatter(X[:, 0], X[:, 1])**

**plt.title("Sample Data for Clustering") plt.xlabel("Feature 1")**

**plt.ylabel("Feature 2") plt.show()**

**# Plotting K-means clustering result plt.figure(figsize=(8, 6))**

**colors = ['r', 'g', 'b', 'y'] for i in range(k):**

**cluster\_points = X[clusters == i]**

**plt.scatter(cluster\_points[:, 0], cluster\_points[:, 1], label=f'Cluster {i+1}') plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', label='Centroids', marker='X') plt.title("K-means Clustering")**

**plt.xlabel("Feature 1")**

**plt.ylabel("Feature 2") plt.legend()**

**plt.show()**

**print("Final Centroids:") print(centroids)**

import numpy as np import pandas as pd

data = {

'Weather': ['Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Rainy', 'Sunny', 'Overcast', 'Overcast', 'Rainy'],

'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],

'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],

'Windy': [False, True, False, False, False, True, True, False, False, False, True, True, False, True], 'Play': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']

}

df = pd.DataFrame(data)

frequency\_table = df.groupby(['Play', 'Weather']).size().unstack().fillna(0) print("Frequency Table:\n", frequency\_table)

P\_Play = df['Play'].value\_counts(normalize=True) print("\nPrior Probabilities:\n", P\_Play)

P\_Weather\_given\_Play = frequency\_table.div(frequency\_table.sum(axis=1), axis=0) print("\nLikelihoods:\n", P\_Weather\_given\_Play)

P\_Sunny = df['Weather'].value\_counts(normalize=True)['Sunny'] P\_Sunny\_given\_Yes = P\_Weather\_given\_Play.loc['Yes', 'Sunny']

P\_Yes\_given\_Sunny = (P\_Sunny\_given\_Yes \* P\_Play['Yes']) / P\_Sunny print("\nP(Yes|Sunny):", P\_Yes\_given\_Sunny)

class NaiveBayesClassifier:

def init (self):

self.priors = {} self.likelihoods = {}

def fit(self, X, y):

data = pd.concat([X, y], axis=1)

self.priors = y.value\_counts(normalize=True)

self.likelihoods = {col: data.groupby([y.name, col]).size().unstack().fillna(0).div(y.value\_counts(), axis=0) for col in X.columns}

def predict(self, X):

results = []

for i in range(X.shape[0]): probs = self.priors.copy() for cls in self.priors.index:

for col in X.columns:

probs[cls] \*= self.likelihoods[col].loc[cls].get(X.iloc[i][col], 0) results.append(probs.idxmax())

return results

X = df[['Weather', 'Temperature', 'Humidity', 'Windy']] y = df['Play']

model = NaiveBayesClassifier() model.fit(X, y)

predictions = model.predict(X) print("\nPredictions:\n", predictions)

example = pd.DataFrame([{'Weather': 'Sunny', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': False}]) prediction = model.predict(example)

print("\nExample Prediction for Weather:Sunny, Temperature:Cool, Humidity:High, Windy:False -> Play:", prediction)