Program 1

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] + heuristicenvy < g[n] + heuristicno:

n = v # n='A'

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

pass

else:

for (m, weight) in get\_neighborsno:

# 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\_set}=2

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

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

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,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.appendno

n = parents[n]

path.append(start\_node)

path.reverse()

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

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

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

closed\_set.addno #{A} len=1

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighborsenvy:

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristicno:

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')

Program2

#AO\*

def compute\_minimum\_cost\_child\_nodes(node, h, graph):

min\_cost = float('inf')

min\_cost\_nodes = []

for info\_list in graph[node]:

cost, nodes = sum(h[c] + w for c, w in info\_list), [c for c, \_ in info\_list]

if cost < min\_cost:

min\_cost = cost

min\_cost\_nodes = nodes

return min\_cost, min\_cost\_nodes

def aostar(status, h, graph, solution, node):

print("HEURISTIC VALUES :", h)

print("SOLUTION GRAPH :", solution)

print("PROCESSING NODE :", node)

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

if status[node] >= 0:

min\_cost, child\_nodes = compute\_minimum\_cost\_child\_nodes(node, h, graph)

h[node] = min\_cost

status[node] = len(child\_nodes)

solved = all(status[child] != -1 for child in child\_nodes)

for child in child\_nodes:

parent[child] = node

if solved:

status[node] = -1

solution[node] = child\_nodes

if node != start:

aostar(status, h, graph, solution, parent[node])

for child in child\_nodes:

status[child] = 0

aostar(status, h, graph, solution, child)

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

graph = {

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

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

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

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

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

'I': [],

'J': []

}

start = 'A'

parent = {}

status = {node: 0 for node in h}

solution = {}

aostar(status, h, graph, solution, start)

print("SOLUTION GRAPH:", solution)

Program 3

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)

program 4

import pandas as pd

import math

def entropy(data):

value\_counts = data.value\_counts()

probabilities = value\_counts / len(data)

entropy\_value = -sum(p \* math.log2(p) for p in probabilities)

return entropy\_value

def information\_gain(data, feature, target):

return entropy(data[target]) - sum(

len(subset) / len(data) \* entropy(subset[target]) for \_, subset in data.groupby(feature)

)

def construct\_tree(data, target, features):

if len(set(data[target])) == 1:

return data[target].iloc[0]

if not features:

return data[target].mode().iloc[0]

best\_feature = max(features, key=lambda f: information\_gain(data, f, target))

tree = {best\_feature: {}}

for value, subset in data.groupby(best\_feature):

subset = subset.drop(columns=[best\_feature])

tree[best\_feature][value] = construct\_tree(subset, target, features - {best\_feature})

return tree

def print\_tree(tree, indent=""):

if isinstance(tree, dict):

for key, value in tree.items():

print(indent + key)

print\_tree(value, indent + " ")

else:

print(indent + f"=> {tree}")

def main():

# Load dataset from playtennis.csv

df = pd.read\_csv("playtennis.csv")

# Extract target and features

target\_column = "Play Tennis"

features = set(df.columns) - {target\_column}

# Construct the decision tree

tree = construct\_tree(df, target=target\_column, features=features)

# Print the decision tree

print\_tree(tree)

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

main()

program 5

Program 5: Back Propagation

import numpy as np

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

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

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

y = y/100

def sigmoid (x):

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

def derivatives\_sigmoid(x):

return x \* (1 - x)

epoch=7000

lr=0.1

inputlayer\_neurons = 2

hiddenlayer\_neurons = 3

output\_neurons = 1

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

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

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

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

for i in range(epoch):

#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)

d\_hiddenlayer = EH \* hiddengrad

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

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

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

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

print("Predicted Output: \n" ,output)

program 6

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())

# convert them in numbers

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\_transformyes

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

accuracy = accuracy\_score(classifier.predict(X\_test), y\_test) \* 100

print("Accuracy is:", accuracy)

program 7:

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)

printyes

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

plt.subplot(1,3,1)

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(1,3,2)

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(1,3,3)

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)))

program 8

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));

program 9

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()