AM IL LAB EXERCISE-1

BFS

pip install networkx matplotlib

import networkx as nx

import matplotlib.pyplot as plt

def bfs(graph, start\_node):

visited = set()

queue = [start\_node]

visited.add(start\_node)

bfs\_order = [start\_node]

while queue:

node = queue.pop(0)

for neighbor in graph.neighbors(node):

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

bfs\_order.append(neighbor)

return bfs\_order

G = nx.DiGraph()

G.add\_edge(0, 1)

G.add\_edge(0, 3)

G.add\_edge(1, 2)

G.add\_edge(1, 3)

G.add\_edge(2, 1)

G.add\_edge(2, 3)

G.add\_edge(3, 3)

start\_node = 2

bfs\_order = bfs(G, start\_node)

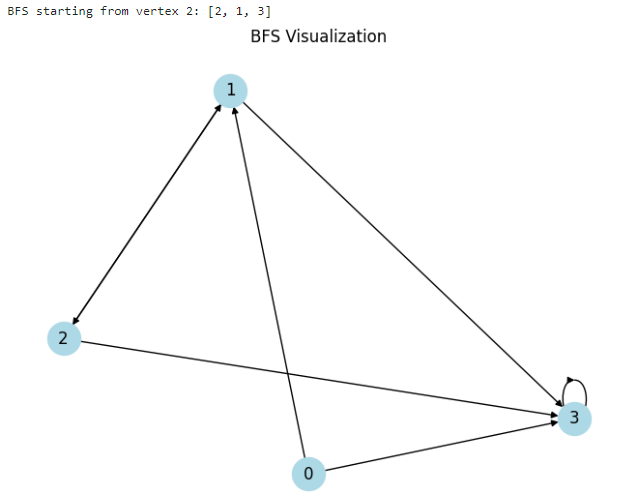
print("BFS starting from vertex 2:", bfs\_order)

pos = nx.spring\_layout(G)

nx.draw(G, pos, with\_labels=True, node\_color='lightblue', node\_size=600)

plt.title("BFS Visualization")

plt.show()



DFS

pip install networkx matplotlib

import matplotlib.pyplot as plt

import networkx as nx

class Graph:

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

self.graph = {}

def add\_edge(self, u, v):

if u not in self.graph:

self.graph[u] = []

self.graph[u].append(v)

def dfs(self, start):

visited = set()

dfs\_order = []

def dfs\_recursive(node):

visited.add(node)

dfs\_order.append(node)

for neighbor in self.graph.get(node, []):

if neighbor not in visited:

dfs\_recursive(neighbor)

dfs\_recursive(start)

return dfs\_order

def visualize\_dfs(graph, dfs\_order):

G = nx.DiGraph()

for node, neighbors in graph.graph.items():

for neighbor in neighbors:

G.add\_edge(node, neighbor)

pos = nx.spring\_layout(G)

labels = {node: node for node in G.nodes()}

node\_colors = ['blue' if node in dfs\_order else 'gray' for node in G.nodes()]

nx.draw(G, pos, labels=labels, node\_color=node\_colors, with\_labels=True, arrows=True)

plt.show()

# Example usage:

g = Graph()

g.add\_edge(0, 1)

g.add\_edge(0, 3)

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 1)

g.add\_edge(2, 3)

g.add\_edge(2, 4)

g.add\_edge(3, 3)

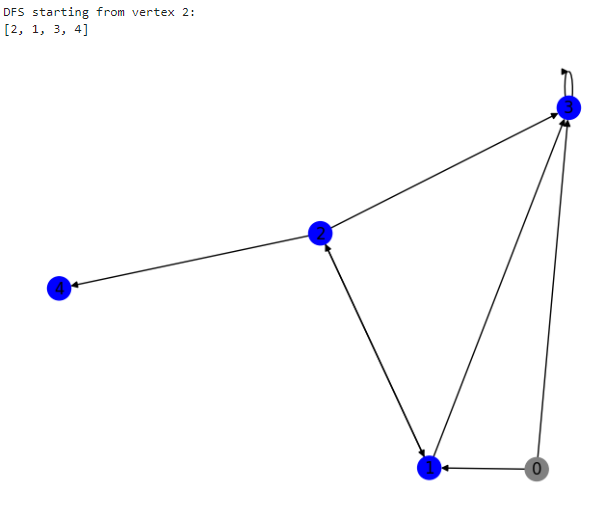
start\_node = 2

dfs\_order = g.dfs(start\_node)

print("DFS starting from vertex 2:")

print(dfs\_order)

visualize\_dfs(g, dfs\_order)



EX-2

import heapq

class PriorityQueue:

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

self.priorityFunction = priorityFunction

self.heap = []

def push(self, item):

heapq.heappush(self.heap, (self.priorityFunction(item), item))

def pop(self):

(\_, item) = heapq.heappop(self.heap)

return item

def empty(self):

return len(self.heap) == 0

def astarGraphSearch(problem, heuristic):

def totalCost(state):

return len(state[1]) + heuristic(state)

return graphSearch(problem, PriorityQueue(totalCost))

def pacmanPathFinder(problem, food):

def manhattanDistanceHeuristic(state):

return abs(state[0][0] - food[0]) + abs(state[0][1] - food[1])

return astarGraphSearch(problem, manhattanDistanceHeuristic)

# The code for the Node class is missing.

# Here's a corrected and complete version of your code for the Node class and A\* search:

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]

children = []

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

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:", path)

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

main()

OUTPUT:

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

EX-6

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import confusion\_matrix

from matplotlib.colors import ListedColormap

# Importing the dataset

dataset = pd.read\_csv('G:\AI ML\diabetes.csv')

X = dataset.iloc[:, [0, 7]].values

y = dataset.iloc[:, 8].values

# Splitting the dataset into the Training set and Test set

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.33, random\_state=42)

# Feature Scaling

sc = StandardScaler()

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

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

# Fitting Naive Bayes to the Training set

classifier = GaussianNB()

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

# Predicting the Test set results

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

# Making the Confusion Matrix

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

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

# Visualizing the Training set results

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

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

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

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

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

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

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

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

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

color=ListedColormap(['purple', 'green'])(i), label=j)

plt.title("Naive Bayes (Training set)")

plt.xlabel('Age')

plt.ylabel('Diabetic')

plt.legend()

plt.show()

# Visualizing the Test set results

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

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

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

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

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

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

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

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

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

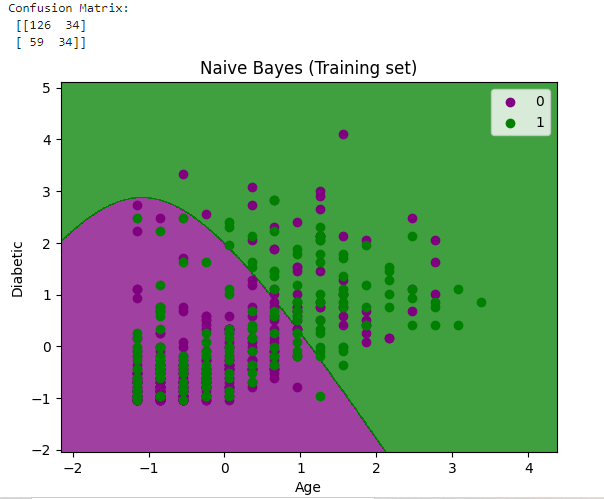
color=ListedColormap(['purple', 'green'])(i), label=j)

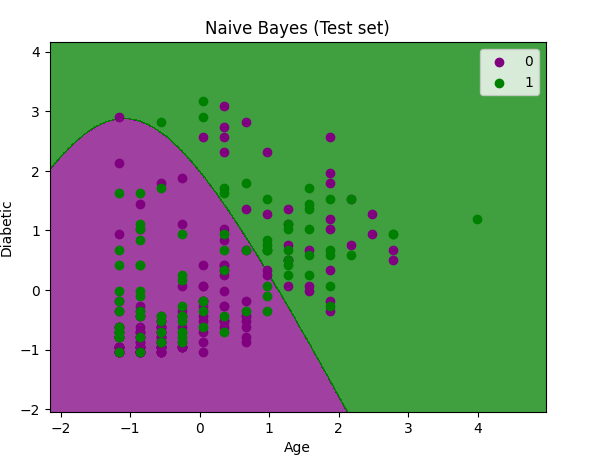
plt.title("Naive Bayes (Test set)")

plt.xlabel('Age')

plt.ylabel('Diabetic')

plt.legend()

plt.show()



EX-7

import numpy as np

import pandas as pd

import sys

sys.executable

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.models import BayesianNetwork

from pgmpy.inference import VariableElimination

# Load the Heart Disease dataset from a CSV file

heart\_disease = pd.read\_csv('G:\AI ML\heart.csv')

# Replace missing values with NaN using NumPy

heart\_disease.replace('?', np.nan, inplace=True)

# Display sample instances from the dataset

print("Sample instances from the dataset are given below:")

print(heart\_disease.head())

# Display attributes and their data types

print("\nAttributes and datatypes:")

print(heart\_disease.dtypes)

# Define the structure of the Bayesian Network

model = BayesianNetwork([('age', 'heartdisease'), ('gender', 'heartdisease'),

('exang', 'heartdisease'), ('cp', 'heartdisease'),

('heartdisease', 'restecg', 'chol')])

print("\nLearning CPD using Maximum Likelihood Estimators")

# Fit the model to the data using Maximum Likelihood Estimation

model.fit(heart\_disease, estimator=MaximumLikelihoodEstimator)

print("\nInferencing with Bayesian Network:")

# Create an inference object using Variable Elimination

heart\_disease\_infer = VariableElimination(model)

# 1. Probability of Heart Disease given evidence - restecg

q1 = heart\_disease\_infer.query(variables=['heartdisease'], evidence={'restecg': 1})

print("1. Probability of Heart Disease given evidence - restecg")

print(q1)

# 2. Probability of Heart Disease given evidence - cp

q2 = heart\_disease\_infer.query(variables=['heartdisease'], evidence={'cp': 2})

print("\n2. Probability of Heart Disease given evidence - cp")

print(q2)

EX-8

# Import the necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.mixture import GaussianMixture

# Load the Iris dataset

iris = datasets.load\_iris()

# Select the first two columns (features) of the dataset

X = iris.data[:, :2]

# Create a DataFrame from the selected features

df = pd.DataFrame(X, columns=['Feature 1', 'Feature 2'])

# Plot the data

plt.scatter(df['Feature 1'], df['Feature 2'])

# Initialize a Gaussian Mixture Model with 3 components

gmm = GaussianMixture(n\_components=3)

# Fit the GMM model to the dataset

gmm.fit(df)

# Assign a cluster label to each sample

labels = gmm.predict(df)

# Add the labels to the DataFrame

df['Labels'] = labels

# Separate the data into clusters based on the labels

cluster\_0 = df[df['Labels'] == 0]

cluster\_1 = df[df['Labels'] == 1]

cluster\_2 = df[df['Labels'] == 2]

# Plot the three clusters with different colors

plt.scatter(cluster\_0['Feature 1'], cluster\_0['Feature 2'], c='r', label='Cluster 0')

plt.scatter(cluster\_1['Feature 1'], cluster\_1['Feature 2'], c='y', label='Cluster 1')

plt.scatter(cluster\_2['Feature 1'], cluster\_2['Feature 2'], c='g', label='Cluster 2')

# Print the converged log-likelihood value

print("Converged Log-Likelihood Value:", gmm.lower\_bound\_)

# Print the number of iterations needed for convergence

print("Number of Iterations to Converge:", gmm.n\_iter\_)

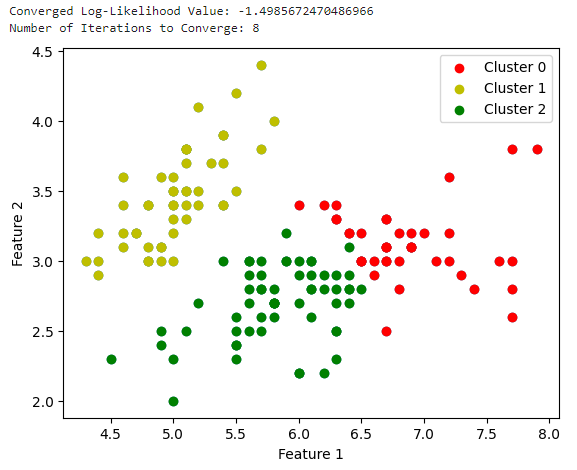
# Add labels and legend

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.legend()

plt.show()



EX-9

from random import seed

from random import randrange

from csv import reader

from math import sqrt

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

from sklearn import neighbors, datasets

# Load a CSV file

def load\_csv(filename):

dataset = list()

with open(filename, 'r') as file:

csv\_reader = reader(file)

for row in csv\_reader:

if not row:

continue

dataset.append(row)

return dataset

# Convert string column to float

def str\_column\_to\_float(dataset, column):

for row in dataset:

row[column] = float(row[column].strip())

# Convert string column to integer

def str\_column\_to\_int(dataset, column):

class\_values = [row[column] for row in dataset]

unique = set(class\_values)

lookup = dict()

for i, value in enumerate(unique):

lookup[value] = i

for row in dataset:

row[column] = lookup[row[column]]

return lookup

# Find the min and max values for each column

def dataset\_minmax(dataset):

minmax = list()

for i in range(len(dataset[0])):

col\_values = [row[i] for row in dataset]

value\_min = min(col\_values)

value\_max = max(col\_values)

minmax.append([value\_min, value\_max])

return minmax

# Rescale dataset columns to the range 0-1

def normalize\_dataset(dataset, minmax):

for row in dataset:

for i in range(len(row)):

row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds

def cross\_validation\_split(dataset, n\_folds):

dataset\_split = list()

dataset\_copy = list(dataset)

fold\_size = int(len(dataset) / n\_folds)

for \_ in range(n\_folds):

fold = list()

while len(fold) < fold\_size:

index = randrange(len(dataset\_copy))

fold.append(dataset\_copy.pop(index))

dataset\_split.append(fold)

return dataset\_split

# Calculate accuracy percentage

def accuracy\_metric(actual, predicted):

correct = 0

for i in range(len(actual)):

if actual[i] == predicted[i]:

correct += 1

return correct / float(len(actual)) \* 100.0

# Euclidean distance calculation

def euclidean\_distance(row1, row2):

distance = 0.0

for i in range(len(row1) - 1):

distance += (row1[i] - row2[i]) \*\* 2

return sqrt(distance)

# Locate the most similar neighbors

def get\_neighbors(train, test\_row, num\_neighbors):

distances = list()

for train\_row in train:

dist = euclidean\_distance(test\_row, train\_row)

distances.append((train\_row, dist))

distances.sort(key=lambda tup: tup[1])

neighbors = list()

# Check if there are enough neighbors

if len(distances) < num\_neighbors:

num\_neighbors = len(distances)

for i in range(num\_neighbors):

neighbors.append(distances[i][0])

return neighbors

# Make a prediction with neighbors

def predict\_classification(train, test\_row, num\_neighbors):

neighbors = get\_neighbors(train, test\_row, num\_neighbors)

output\_values = [row[-1] for row in neighbors]

prediction = max(set(output\_values), key=output\_values.count)

return prediction

# KNN Algorithm

def k\_nearest\_neighbors(train, test, num\_neighbors):

predictions = list()

for row in test:

output = predict\_classification(train, row, num\_neighbors)

predictions.append(output)

return predictions

# Evaluate the KNN algorithm using k-fold cross-validation

def evaluate\_algorithm(dataset, algorithm, n\_folds, \*args):

folds = cross\_validation\_split(dataset, n\_folds)

scores = []

for fold in folds:

train\_set = list(folds)

train\_set.remove(fold)

train\_set = [item for sublist in train\_set for item in sublist] # Flatten the list

test\_set = [row for row in fold]

predicted = algorithm(train\_set, test\_set, \*args)

actual = [row[-1] for row in fold]

accuracy = accuracy\_metric(actual, predicted)

scores.append(accuracy)

return scores

# Test the KNN on the Iris Flowers dataset

seed(1)

filename = 'G:\AI ML\iris.csv'

dataset = load\_csv(filename)

for i in range(len(dataset[0]) - 1):

str\_column\_to\_float(dataset, i)

str\_column\_to\_int(dataset, len(dataset[0]) - 1)

# Evaluate the algorithm

n\_folds = 5

num\_neighbors = 5

scores = evaluate\_algorithm(dataset, k\_nearest\_neighbors, n\_folds, num\_neighbors)

print('Scores:', scores)

print("Mean Accuracy: %.3f%%" % (sum(scores) / float(len(scores)))

# Visualization with scikit-learn's KNeighborsClassifier

n\_neighbors = 6

# Import some data to play with

iris = datasets.load\_iris()

X = iris.data[:, :2] # Select the first two features for visualization

y = iris.target

h = 0.02 # Step size in the mesh

# Create color maps

cmap\_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#00AAFF'])

cmap\_bold = ListedColormap(['#FF0000', '#00FF00', '#00AABB'])

# We create an instance of Neighbors Classifier and fit the data.

clf = neighbors.KNeighborsClassifier(n\_neighbors, weights='distance')

clf.fit(X, y)

# Calculate min, max, and limits

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

# Predict class using data and KNN classifier

Z = clf.predict(np.c\_[xx.ravel(), yy.ravel()])

# Put the result into a color plot

Z = Z.reshape(xx.shape)

plt.figure()

plt.pcolormesh(xx, yy, Z, cmap=cmap\_light)

# Plot also the training points

plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap\_bold)

plt.xlim(xx.min(), xx.max())

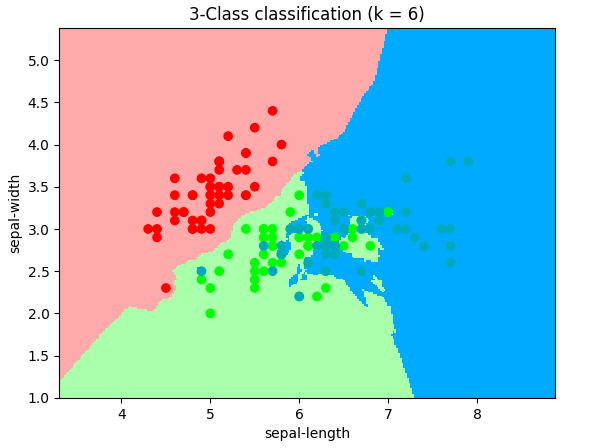
plt.ylim(yy.min(), yy.max())

plt.title("3-Class classification (k = %i)" % n\_neighbors)

plt.xlabel("sepal-length")

plt.ylabel("sepal-width")

plt.show()

`

EX-10

import pandas as pd

import numpy as np

from sklearn.metrics import accuracy\_score, confusion\_matrix

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

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

from six import StringIO

from IPython.display import Image

import pydotplus

import os

import matplotlib.pyplot as plt

# Read and print the training dataset

traindatainput = pd.read\_csv('monks2\_train.csv', delimiter=",")

print("Training Dataset")

print(traindatainput)

# Read and print the test dataset

testdatainput = pd.read\_csv('monks2\_test.csv', delimiter=",")

print("Test Dataset")

print(testdatainput)

# Define the feature columns and target column

X\_train = traindatainput[['al', 'a2', 'a3', 'a4', 'a5', 'a6']].values

Y\_train = traindatainput["label"]

X\_test = testdatainput[['al', 'a2', 'a3', 'a4', 'a5', 'a6']].values

Y\_test = testdatainput["label"]

# Split the data into training and testing sets

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X\_train, Y\_train, test\_size=0.3, random\_state=3)

# Create a Decision Tree with criterion 'entropy' and max\_depth 4

monksTree = DecisionTreeClassifier(criterion="entropy", max\_depth=4)

monksTree.fit(X\_train, Y\_train)

predicted = monksTree.predict(X\_test)

print(predicted)

# Find and print the accuracy of the decision tree model

print("Decision Tree's Accuracy:", accuracy\_score(Y\_test, predicted))

# Visualize the decision tree

dot\_data = StringIO()

export\_graphviz(monksTree, out\_file=dot\_data, filled=True, rounded=True, special\_characters=True)

graph = pydotplus.graph\_from\_dot\_data(dot\_data.getvalue())

graph.write\_png('tree.png')

Image(graph.create\_png())

# Try different criteria and check accuracy

monksTree = DecisionTreeClassifier(criterion='gini')

monksTree.fit(X\_train, Y\_train)

pred = monksTree.predict(X\_test)

print('Criterion=gini', accuracy\_score(Y\_test, pred))

monksTree = DecisionTreeClassifier(criterion='entropy')

monksTree.fit(X\_train, Y\_train)

pred = monksTree.predict(X\_test)

print('Criterion=entropy', accuracy\_score(Y\_test, pred))

# Check if pruning can improve results

max\_depth = []

acc\_gini = []

acc\_entropy = []

for i in range(1, 30):

monksTree = DecisionTreeClassifier(criterion='gini', max\_depth=i)

monksTree.fit(X\_train, Y\_train)

pred = monksTree.predict(X\_test)

acc\_gini.append(accuracy\_score(Y\_test, pred))

monksTree = DecisionTreeClassifier(criterion='entropy', max\_depth=i)

monksTree.fit(X\_train, Y\_train)

pred = monksTree.predict(X\_test)

acc\_entropy.append(accuracy\_score(Y\_test, pred))

max\_depth.append(i)

d = pd.DataFrame({'acc\_gini': pd.Series(acc\_gini), 'acc\_entropy': pd.Series(acc\_entropy), 'max\_depth': pd.Series(max\_depth)})

# Visualize changes in parameters

plt.plot('max\_depth', 'acc\_gini', data=d, label='gini')

plt.plot('max\_depth', 'acc\_entropy', data=d, label='entropy')

plt.xlabel('max\_depth')

plt.ylabel('accuracy')

plt.legend()

plt.show()

# Check accuracy with a shorter tree (max depth of 7) and criterion of entropy

monksTree = DecisionTreeClassifier(criterion='entropy', max\_depth=7)

monksTree.fit(X\_train, Y\_train)

pred = monksTree.predict(X\_test)

print("Accuracy after reducing depth and using Entropy:", accuracy\_score(Y\_test, pred))

EX-11

# Import libraries

import numpy as np

import pandas as pd

from sklearn.preprocessing import OneHotEncoder

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

import sklearn.metrics

import matplotlib.pyplot as plt

# Load data

data = pd.read\_csv(‘G:\AI ML\blood-transf.csv’)

data = data.sample(frac=1).reset\_index(drop=True)

# Define X and Y

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

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

# One-hot encode Y

one\_hot\_encoder = OneHotEncoder(sparse=False)

Y = one\_hot\_encoder.fit\_transform(np.array(Y).reshape(-1, 1))

# Split data into training and testing sets

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.25)

# Define the neural network architecture

layers = [len(X[0]), 11, 8, 5, 5, len(Y[0])]

learning\_rate = 0.1

epochs = 500

# Train the neural network

weights = NeuralNetwork(X\_train, Y\_train, X\_test, Y\_test, epochs=epochs, nodes=layers, lr=learning\_rate)

# Print the final weights

print("Final weights:\n", weights)

# Test the model and print the accuracy

accuracy = Accuracy(X\_test, Y\_test, weights, display=True)

print("Testing Accuracy: {}".format(accuracy))

# Predict and evaluate

Y\_result = [Predict(x, weights) for x in X\_test]

Y\_test = Y\_test.tolist()

Y\_result = [list(map(int, y)) for y in Y\_result]

r2\_score = sklearn.metrics.r2\_score(Y\_test, Y\_result)

print("R2 score: {}".format(r2\_score))

# Print the classification report

print(sklearn.metrics.classification\_report(Y\_test, Y\_result))

# Compute and plot the ROC curve

fpr, tpr, thresholds = sklearn.metrics.roc\_curve(Y\_test, Y\_result)

plot\_roc\_curve(fpr, tpr)

EX-12

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.utils import shuffle

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

from sklearn.metrics import accuracy\_score

# Load the dataset

df = pd.read\_csv(r'G:\AI ML\breast\_cancer.csv') # Fixed path by using r''

# Extract the feature values

X = df[['radius\_mean', 'concavity\_mean']].values

# Create separate datasets for open and high values

open\_x = X[:50, 0]

open\_y = X[:50, 1]

high\_x = X[50:100, 0]

high\_y = X[50:100, 1]

# Plot the data

plt.figure(figsize=(10, 8))

plt.scatter(open\_x, open\_y, marker='+', color='green', label='Open')

plt.scatter(high\_x, high\_y, marker='o', color='red', label='High')

plt.xlabel('Radius Mean')

plt.ylabel('Concavity Mean')

plt.legend()

plt.show()

# Extract target values (diagnosis) and convert to -1 or 1

Y = df['diagnosis'].map({'M': -1, 'B': 1}).values

# Drop diagnosis column from the dataframe

df = df.drop(['diagnosis'], axis=1)

# Shuffle the data

X, Y = shuffle(X, Y)

# Split the data into training and test sets

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, train\_size=0.9)

# Initialize parameters

w1 = np.zeros(1)

w2 = np.zeros(1)

epochs = 10000

alpha = 0.0001

# Training the SVM using gradient descent

for epoch in range(epochs):

y = w1 \* X\_train[:, 0] + w2 \* X\_train[:, 1]

count = 0

for val in y:

if val >= 1:

cost = 0

w1 = w1 - alpha \* (2 \* (1 / epochs) \* w1)

w2 = w2 - alpha \* (2 \* (1 / epochs) \* w2)

else:

cost = 1 - val

w1 = w1 + alpha \* (X\_train[count, 0] \* Y\_train[count] - 2 \* (1 / epochs) \* w1)

w2 = w2 + alpha \* (X\_train[count, 1] \* Y\_train[count] - 2 \* (1 / epochs) \* w2)

count += 1

# Clip the weights

index = list(range(10, X\_train.shape[0]))

w1 = np.delete(w1, index)

w2 = np.delete(w2, index)

# Extract the test data features

test\_fl = X\_test[:, 0].reshape(-1, 1)

test\_f2 = X\_test[:, 1].reshape(-1, 1)

# Predict

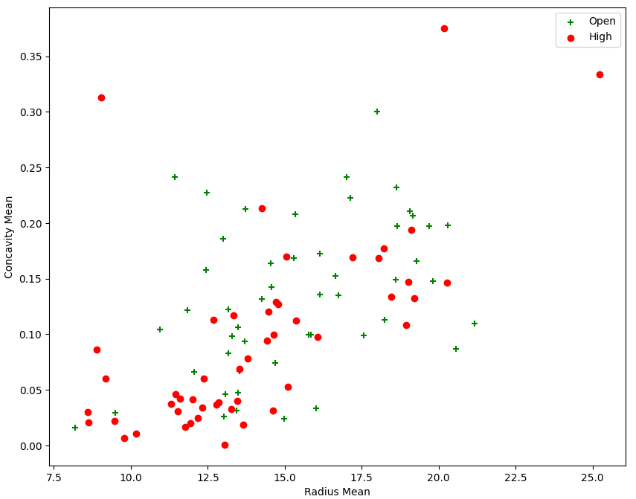
y\_pred = w1 \* test\_fl + w2 \* test\_f2

predictions = np.where(y\_pred > 1, 1, -1)

# Calculate and print the accuracy

accuracy = accuracy\_score(Y\_test, predictions)

print("Accuracy: ", accuracy)



EX-13

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Load the dataset

data = pd.read\_csv("G:\AI ML\diabetes-1.csv")

# Check the shape and first few rows of the dataset

print(data.shape)

print(data.head())

# Separate features (X) and target variable (Y)

X = data.drop(['SkinThickness', 'Outcome'], axis=1)

Y = data['Outcome']

# Split the data into training and testing sets

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.2, random\_state=42)

# Import the logistic regression model

from sklearn.linear\_model import LogisticRegression

# Create a logistic regression classifier

clf = LogisticRegression()

# Train the model on the training data

clf.fit(X\_train, y\_train)

# Predict the response for the test dataset

y\_pred = clf.predict(X\_test)

# Import the accuracy metric

from sklearn.metrics import accuracy\_score

# Calculate and print the accuracy of the model

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

**Output**

(767, 9)

Pregnancies Glucose BloodPressure SkinThickness Insulin BMI \

0 6 148 72 35 0 33.6

1 1 85 66 29 0 26.6

2 8 183 64 0 0 23.3

3 1 89 66 23 94 28.1

4 0 137 40 35 168 43.1

DiabetesPedigreeFunction Age Outcome

0 0.627 50 1

1 0.351 31 0

2 0.672 32 1

3 0.167 21 0

4 2.288 33 1

Accuracy: 0.7857142857142857