**PRACTICAL NO 1A**

from collections import deque

gn = {

'Mumbai': {'Pune': 75, 'Ahmedabad': 118, 'Bangalore': 140},

'Delhi': {'Jaipur': 85, 'Ahmedabad': 90, 'Bangalore': 101, 'Hyderabad': 211},

'Bangalore': {'Hyderabad': 120, 'Chennai': 138, 'Mumbai': 146},

'Hyderabad': {'Bangalore': 120, 'Delhi': 211},

'Chennai': {'Bangalore': 138},

'Pune': {'Mumbai': 75},

'Ahmedabad': {'Mumbai': 118, 'Delhi': 90},

'Jaipur': {'Delhi': 85}

}

def bfs(start, goal):

queue, visited = deque([start]), {start}

path = []

while queue:

node = queue.popleft()

path.append(node)

if node == goal:

return path

for neighbor in gn[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

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

print(" -> ".join(bfs('Mumbai', 'Delhi')))

**PRACTICAL NO 1B**

# Neighbors (g) values

gn = {

'Mumbai': {'Pune': 75, 'Ahmedabad': 118, 'Bangalore': 140},

'Delhi': {'Jaipur': 85, 'Ahmedabad': 90, 'Bangalore': 101, 'Hyderabad': 211},

'Bangalore': {'Hyderabad': 120, 'Chennai': 138, 'Mumbai': 146},

'Hyderabad': {'Bangalore': 120, 'Delhi': 211},

'Chennai': {'Bangalore': 138},

'Pune': {'Mumbai': 75},

'Ahmedabad': {'Mumbai': 118, 'Delhi': 90},

'Jaipur': {'Delhi': 85}

}

def dfs(start, goal):

stack, visited, result = [start], set(), ''

while stack:

current = stack.pop()

if current not in visited:

visited.add(current)

result += current + ' '

if current == goal:

return result

stack.extend(reversed([n for n in gn[current] if n not in visited]))

start, goal = 'Mumbai', 'Delhi'

result = dfs(start, goal)

if isinstance(result, str):

result = result.split()

print(f"DFS Traversal from {start} to {goal}:")

print(' -> '.join(result))

**PRACTICAL NO 2A**

import queue as Q

hn = {

'Mumbai': 336, 'Delhi': 0, 'Bangalore': 160, 'Hyderabad': 242,

'Chennai': 161, 'Pune': 77, 'Ahmedabad': 151, 'Jaipur': 226

}

gn = {

'Mumbai': {'Pune': 75, 'Ahmedabad': 118, 'Bangalore': 140},

'Delhi': {'Jaipur': 85, 'Ahmedabad': 90, 'Bangalore': 101, 'Hyderabad': 211},

'Bangalore': {'Hyderabad': 120, 'Chennai': 138, 'Mumbai': 146},

'Hyderabad': {'Bangalore': 120, 'Delhi': 211},

'Chennai': {'Bangalore': 138},

'Pune': {'Mumbai': 75},

'Ahmedabad': {'Mumbai': 118, 'Delhi': 90},

'Jaipur': {'Delhi': 85}

}

# A\* search function

def astar(start, goal):

q = Q.PriorityQueue()

q.put((hn[start], start, [start]))

while not q.empty():

cost, city, path = q.get()

if city == goal:

print("Path:", " -> ".join(path), "| Total cost:", cost)

return

for neighbor, distance in gn[city].items():

new\_cost = cost - hn[city] + distance + hn[neighbor]

q.put((new\_cost, neighbor, path + [neighbor]))

# Run A\* search

astar('Mumbai','Delhi')

**PRACTICAL NO 2B**

graph = {

'A': {'B': 75, 'C': 118, 'D': 140}, 'B': {'A': 75, 'E': 71},

'C': {'A': 118, 'F': 111}, 'D': {'A': 140, 'E': 151, 'G': 99, 'H': 80},

'E': {'B': 71, 'D': 151}, 'F': {'C': 111, 'I': 70},

'G': {'D': 99, 'J': 211}, 'H': {'D': 80, 'K': 146, 'L': 97},

'I': {'F': 70, 'M': 75}, 'M': {'I': 75, 'K': 120},

'K': {'M': 120, 'H': 146, 'L': 138}, 'L': {'H': 97, 'K': 138, 'J': 101},

'J': {'G': 211, 'L': 101}

}

hn = {'A': 366, 'B': 374, 'C': 329, 'D': 253, 'E': 380, 'F': 244, 'G': 176,

'H': 193, 'I': 241, 'M': 242, 'K': 160, 'L': 100, 'J': 0}

def rbfs(curr, goal, path):

if curr == goal: return path

for loc in sorted(graph[curr], key=lambda l: graph[curr][l] + hn[l]):

res = rbfs(loc, goal, path + [loc])

if res: return res

return None

def search(start, goal):

res = rbfs(start, goal, [start])

if res:

cost = sum(graph[res[i]][res[i+1]] for i in range(len(res)-1))

print("Path:", res, "Cost:", cost)

else:

print("Path not found!")

search('A', 'J')

**PRACTICAL NO 3**

import pandas as pd

from sklearn.preprocessing import LabelEncoder

from sklearn.tree import DecisionTreeClassifier, plot\_tree, export\_graphviz

import graphviz

# Load and preprocess data

PlayTennis = pd.read\_csv('bar.csv').apply(LabelEncoder().fit\_transform)

# Display the processed dataset

print(PlayTennis)

# Train decision tree classifier

x, y = PlayTennis.drop('play', axis=1), PlayTennis['play']

clf = DecisionTreeClassifier(criterion='entropy').fit(x, y)

# Visualize decision tree

plot\_tree(clf)

dot\_data = export\_graphviz(clf, out\_file=None) # Export as dot data

graph = graphviz.Source(dot\_data) # Create a graphviz source object

graph.render("decision\_tree") # Save the visualization

graph.view() # Open the saved file

# Predict and compare predictions

print(clf.predict(x) == y)

**PRACTICAL NO 4**

import numpy as np

class NeuralNetwork:

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

np.random.seed()

self.weights = 2 \* np.random.random((3, 1)) - 1

def sigmoid(self, x):

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

def sigmoid\_derivative(self, x):

return x \* (1 - x)

def train(self, inputs, outputs, iterations):

for \_ in range(iterations):

output = self.think(inputs)

self.weights += np.dot(inputs.T, (outputs - output) \* self.sigmoid\_derivative(output))

def think(self, inputs):

return self.sigmoid(np.dot(inputs.astype(float), self.weights))

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

nn = NeuralNetwork()

print("Beginning Randomly Generated Weights:\n", nn.weights)

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

outputs = np.array([[0, 1, 1, 0]]).T

nn.train(inputs, outputs, 15000)

print("Ending Weights After Training:\n", nn.weights)

user\_input = np.array([int(input(f"Input {i+1}: ")) for i in range(3)])

print("Considering New Situation:", \*user\_input)

print("New Output data:\n", nn.think(user\_input))

**PRACTICAL NO 5**

import warnings

warnings.filterwarnings("ignore")

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split, GridSearchCV, cross\_val\_score

from sklearn.svm import SVC

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

import time

# Load dataset and display info

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

print(df.head(), df.shape, df.describe(), sep='\n')

# Prepare data

X, y = df.drop("Outcome", axis=1), df["Outcome"]

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

# Train SVC and evaluate

svc = SVC(kernel="linear").fit(X\_train, y\_train)

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

print(f"Confusion Matrix:\n{confusion\_matrix(y\_test, y\_pred)}")

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

print(classification\_report(y\_test, y\_pred))

# Cross-validation for accuracy

accuracies = cross\_val\_score(svc, X\_train, y\_train, cv=10)

print(f"Average Accuracy: {accuracies.mean()\*100:.2f}%")

print(f"Standard Deviation of Accuracies: {accuracies.std()\*100:.2f}%")

# Hyperparameter tuning

svm\_cv = GridSearchCV(SVC(kernel="linear"), {"C": np.arange(1, 20)}, cv=8)

start\_time = time.time()

svm\_cv.fit(X\_train, y\_train)

print(f"Best Score: {svm\_cv.best\_score\_}, Best Params: {svm\_cv.best\_params\_}")

print(f"Elapsed time: {time.time() - start\_time:.3f} seconds")

# Evaluate tuned model

y\_pred\_tuned = SVC(kernel="linear", C=svm\_cv.best\_params\_["C"]).fit(X\_train, y\_train).predict(X\_test)

print(f"Tuned Accuracy: {accuracy\_score(y\_test, y\_pred\_tuned)}")

**PRACTICAL NO 6**

import pandas as pd

from sklearn.ensemble import AdaBoostClassifier

from sklearn.model\_selection import cross\_val\_score

# Load dataset and prepare data

df = pd.read\_csv("https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv", names=['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class'])

X, y = df.iloc[:, :-1], df.iloc[:, -1]

# Evaluate model with SAMME algorithm

model = AdaBoostClassifier(n\_estimators=30, random\_state=7, algorithm='SAMME')

print("Mean Accuracy:", cross\_val\_score(model, X, y).mean())

**PRACTICAL NO 7**

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.preprocessing import LabelEncoder

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

# Load dataset and encode categorical variables

df = pd.read\_csv('C:/Users/nupur/sem 4 journals/AI/heart.csv')

le = LabelEncoder()

df[['Sex', 'ChestPainType', 'RestingECG', 'Diagnosis']] = df[['Sex', 'ChestPainType', 'RestingECG', 'Diagnosis']].apply(le.fit\_transform)

# Display data info and plot counts

print(df.head(11), df.info())

for col in ['Sex', 'ChestPainType', 'RestingECG', 'Diagnosis']:

sns.countplot(x=df[col])

plt.title(f"Count of '{col}'")

plt.show()

# Split data and train model

X, y = df.drop('Diagnosis', axis=1), df['Diagnosis']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=7)

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

# Evaluate model

print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

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

print(classification\_report(y\_test, y\_pred, zero\_division=1))

**PRACTICAL NO 8**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion\_matrix, classification\_report, roc\_curve, roc\_auc\_score

# Load dataset and split into features and target

df = pd.read\_csv('diabetes.csv')

print("Data Preview:\n", df.head(), "\n")

print("Data Shape:", df.shape)

print("Data Types:\n", df.dtypes)

# Split data into features and target

X, y = df.drop('Outcome', axis=1), df['Outcome']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4, random\_state=42)

# Train and evaluate k-NN for k from 1 to 9

train\_acc = [KNeighborsClassifier(n\_neighbors=k).fit(X\_train, y\_train).score(X\_train, y\_train) for k in range(1, 10)]

test\_acc = [KNeighborsClassifier(n\_neighbors=k).fit(X\_train, y\_train).score(X\_test, y\_test) for k in range(1, 10)]

# Plot accuracy

plt.plot(range(1, 10), train\_acc, label='Train Acc')

plt.plot(range(1, 10), test\_acc, label='Test Acc')

plt.title('k-NN Accuracy for Varying k'), plt.xlabel('k'), plt.ylabel('Accuracy'), plt.legend(), plt.show()

# Fit best k-NN model and evaluate

knn = KNeighborsClassifier(n\_neighbors=7).fit(X\_train, y\_train)

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

print(confusion\_matrix(y\_test, y\_pred), "\n", classification\_report(y\_test, y\_pred))

# ROC Curve and AUC Score

y\_proba = knn.predict\_proba(X\_test)[:, 1]

fpr, tpr, \_ = roc\_curve(y\_test, y\_proba)

plt.plot(fpr, tpr, label='k-NN (n\_neighbors=7)')

plt.plot([0, 1], [0, 1], 'k--')

plt.xlabel('FPR'), plt.ylabel('TPR'), plt.title('ROC Curve'), plt.legend(), plt.show()

print("ROC AUC Score:", roc\_auc\_score(y\_test, y\_proba))

# Hyperparameter tuning with GridSearchCV

best\_knn = GridSearchCV(KNeighborsClassifier(), {'n\_neighbors': np.arange(1, 50)}, cv=5).fit(X, y)

print(f"Best Score: {best\_knn.best\_score\_}, Best Params: {best\_knn.best\_params\_}")

**PRACTICAL NO 9**

import pandas as pd

from mlxtend.frequent\_patterns import apriori, association\_rules

from mlxtend.preprocessing import TransactionEncoder

# Sample transactions

transactions = [

['milk', 'bread', 'butter'],

['milk', 'bread'],

['milk', 'butter'],

['bread', 'butter'],

['milk', 'bread', 'butter']

]

# One-hot encode the transactions

te = TransactionEncoder()

te\_ary = te.fit(transactions).transform(transactions)

df = pd.DataFrame(te\_ary, columns=te.columns\_)

# Generate frequent itemsets

frequent\_itemsets = apriori(df, min\_support=0.6, use\_colnames=True)

# Generate association rules

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.7)

# Print the results

print("Frequent Itemsets:")

print(frequent\_itemsets)

print("\nAssociation Rules:")

print(rules[['antecedents', 'consequents', 'support', 'confidence']])