Practical No. 1: Breadth First Search & Iterative Depth First Search

Python Code for Breadth First Search algorithm

**BFS.py**

from collections import deque

def bfs(graph, start):

  visited = set()

  queue = deque([start])

  while queue:

    vertex = queue.popleft()

    if vertex not in visited:

      visited.add(vertex)

      print(vertex, end=" ")

      neighbors = graph[vertex]

      for neighbor in neighbors:

        if neighbor not in visited:

          queue.append(neighbor)

graph = {

  'A': ['B', 'C'],

  'B': ['A', 'D', 'E'],

  'C': ['A', 'F'],

  'D': ['B'],

  'E': ['B', 'F'],

  'F': ['C', 'E']

  }

start\_vertex = 'A'

bfs(graph, start\_vertex)

**IDFS.py**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def add\_edge(self, u, v):

        self.graph[u].append(v)

        self.graph[v].append(u)  # Assuming an undirected graph

    def iterative\_dfs(self, start, end):

        if start == end:

            return [start]

        visited = set()

        stack = [(start, [start])]

        while stack:

            current\_vertex, path = stack.pop()

            visited.add(current\_vertex)

            for neighbor in self.graph[current\_vertex]:

                if neighbor not in visited:

                    if neighbor == end:

                        return path + [neighbor]

                    stack.append((neighbor, path + [neighbor]))

        return None  # No path found

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

    g = Graph()

    g.add\_edge(1, 2)

    g.add\_edge(1, 3)

    g.add\_edge(2, 4)

    g.add\_edge(2, 5)

    g.add\_edge(3, 6)

    g.add\_edge(3, 7)

    g.add\_edge(4, 8)

    g.add\_edge(4, 9)

    g.add\_edge(5, 10)

    g.add\_edge(5, 11)

    g.add\_edge(6, 12)

    g.add\_edge(6, 13)

    g.add\_edge(7, 14)

    g.add\_edge(7, 15)

    start\_node = 1

    end\_node = 9

    shortest\_path = g.iterative\_dfs(start\_node, end\_node)

    if shortest\_path:

        print(f"Shortest path from {start\_node} to {end\_node}: {shortest\_path}")

    else:

        print(f"No path found from {start\_node} to {end\_node}")

**2. Python code for Recursive A\* Search algorithm**

**A\_star.py**

import heapq

# Define the map of Romania with distances between cities

romania\_map = {

 'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

 'Zerind': {'Arad': 75, 'Oradea': 71},

 'Timisoara': {'Arad': 118, 'Lugoj': 111},

 'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

 'Oradea': {'Zerind': 71, 'Sibiu': 151},

 'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

 'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

 'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

 'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

 'Drobeta': {'Mehadia': 75, 'Craiova': 120},

 'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

 'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

 'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},

 'Giurgiu': {'Bucharest': 90},

 'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

 'Hirsova': {'Urziceni': 98, 'Eforie': 86},

 'Eforie': {'Hirsova': 86},

 'Vaslui': {'Urziceni': 142, 'Iasi': 92},

 'Iasi': {'Vaslui': 92, 'Neamt': 87},

 'Neamt': {'Iasi': 87}

}

class Node:

  def \_\_init\_\_(self, city, cost, parent=None):

   self.city = city

   self.cost = cost

   self.parent = parent

  def \_\_lt\_\_(self, other):

   return self.cost < other.cost

def heuristic(node, goal):

  return 0 # No need for heuristic in this case

def astar\_search(graph, start, goal):

    open\_list = []

    closed\_set = set()

    heapq.heappush(open\_list, start)

    while open\_list:

     current\_node = heapq.heappop(open\_list)

     if current\_node.city == goal.city:

      path = []

      while current\_node:

       path.append(current\_node.city)

       current\_node = current\_node.parent

      return path[::-1] # Reverse the path to get it from start to goal

     closed\_set.add(current\_node.city)

     for neighbor, distance in graph[current\_node.city].items():

        if neighbor not in closed\_set:

          new\_cost = current\_node.cost + distance

          new\_node = Node(neighbor, new\_cost, current\_node)

          heapq.heappush(open\_list, new\_node)

    return None # No path found

start\_city = 'Arad'

goal\_city = 'Bucharest'

start\_node = Node(start\_city, 0)

goal\_node = Node(goal\_city, 0)

path = astar\_search(romania\_map, start\_node, goal\_node)

if path:

 print("Path found:", path)

else:

 print("No path found")

**Python code for Recursive Best-First Search algorithm**

**Recursive\_bfs.py**

from queue import PriorityQueue

class Node:

  def \_\_init\_\_(self, state, parent=None, f=float('inf')):

   self.state = state

   self.parent = parent

   self.f = f

def rbfs(start, goal):

  f\_limit = float('inf')

  stack = [(Node(start, f=0), f\_limit)]

  visited = set()

  while stack:

   (node, f) = stack.pop()

   visited.add(node.state)

   if node.state == goal:

    path = []

    cost = node.f

    while node is not None:

      path.append(node.state)

      node = node.parent

    return list(reversed(path)), cost

   successors = []

   for neighbor, cost in get\_neighbors(node.state):

    if neighbor not in visited:

      child = Node(neighbor, parent=node)

      child.f = max(child.parent.f, cost)

      successors.append(child)

   if len(successors) == 0:

    continue

   successors.sort(key=lambda x: x.f)

   best = successors[0]

   if best.f > f\_limit:

    return None, best.f

   alternative = successors[1].f if len(successors) > 1 else float('inf')

   stack.append((best, min(f\_limit, alternative)))

  return None, float('inf')

def get\_neighbors(state):

 successors = {

 1: [(2, 3), (3, 5)],

 2: [(1, 3), (4, 7)],

 3: [(1, 5), (5, 2)],

 4: [(2, 7), (6, 4)],

 5: [(3, 2), (7, 6)],

 6: [(4, 4), (8, 8)],

 7: [(5, 6), (8, 5)],

 8: [(6, 8), (7, 5)],

 }

 return successors.get(state, [])

if \_\_name\_\_ == '\_\_main\_\_':

 start\_state = 1

 goal\_state = 8

 path, cost = rbfs(start\_state, goal\_state)

 if path is not None:

  print(f"Optimal path from {start\_state} to {goal\_state}:")

  print(" -> ".join(map(str, path)))

  print(f"Total cost: {cost}")

 else:

  print("No path found.")

**Practical No.3: Decision Tree Learning**

**Python Code: Decision\_tree.py**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.tree import DecisionTreeClassifier, plot\_tree

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

from sklearn.metrics import accuracy\_score

data = pd.read\_csv('Iris2.csv')

X = data.drop('Species', axis=1)

y = data['Species']

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

clf = DecisionTreeClassifier()

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

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

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

plt.figure(figsize=(12, 8))

plot\_tree(clf, filled=True, feature\_names=X.columns, class\_names=y.unique().astype(str))

plt.title("Decision Tree Visualization")

plt.show()

**Practical No.4: Feedforward Backpropagation Neural Network**

**FNN.py**

import numpy as np

def sigmoid(x):

  return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

  return x \* (1 - x)

class NeuralNetwork:

  def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

    self.weights\_input\_hidden = np.random.uniform(-1, 1, (input\_size, hidden\_size))

    self.weights\_hidden\_output = np.random.uniform(-1, 1, (hidden\_size, output\_size))

  def forward(self, inputs):

    self.hidden\_input = np.dot(inputs, self.weights\_input\_hidden)

    self.hidden\_output = sigmoid(self.hidden\_input)

    self.output\_input = np.dot(self.hidden\_output, self.weights\_hidden\_output)

    self.predicted\_output = sigmoid(self.output\_input)

    return self.predicted\_output

  def backward(self, inputs, target, learning\_rate):

    error = target - self.predicted\_output

    delta\_output = error \* sigmoid\_derivative(self.predicted\_output)

    error\_hidden = delta\_output.dot(self.weights\_hidden\_output.T)

    delta\_hidden = error\_hidden \* sigmoid\_derivative(self.hidden\_output)

    self.weights\_hidden\_output += np.outer(self.hidden\_output, delta\_output) \* learning\_rate

    self.weights\_input\_hidden += np.outer(inputs, delta\_hidden) \* learning\_rate

  def train(self, training\_data, targets, epochs, learning\_rate):

    for epoch in range(epochs):

      for i in range(len(training\_data)):

        inputs = training\_data[i]

        target = targets[i]

        self.forward(inputs)

        self.backward(inputs, target, learning\_rate)

  def predict(self, inputs):

    return self.forward(inputs)

training\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

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

input\_size = 2

hidden\_size = 4

output\_size = 1

learning\_rate = 0.1

epochs = 10000

nn = NeuralNetwork(input\_size, hidden\_size, output\_size)

nn.train(training\_data, targets, epochs, learning\_rate)

for i in range(len(training\_data)):

 inputs = training\_data[i]

 prediction = nn.predict(inputs)

 print(f"Input: {inputs}, Predicted Output: {prediction}")

**Practical No. 5: Support Vector Machines(SVM)**

**Python Code: SVM.py**

import pandas as pd

from sklearn.svm import SVC

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

from sklearn.metrics import accuracy\_score

data = pd.read\_csv('Iris2.csv')

X = data.drop('Species', axis=1)

y = data['Species']

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

svm\_classifier = SVC(kernel='linear')

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

y\_pred = svm\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

**Practical No. 6:Adaboost Ensembale Learning.**

**Adaboost.py**

import pandas as pd

from sklearn import model\_selection

from sklearn.ensemble import AdaBoostClassifier

import warnings

warnings.filterwarnings('ignore')

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = pd.read\_csv(url, names=names)

array = dataframe.values

X = array[:, 0:8]

Y = array[:, 8]

seed = 7

num\_trees = 30

kfold = model\_selection.KFold(n\_splits=10, random\_state=seed, shuffle=True)

model = AdaBoostClassifier(n\_estimators=num\_trees, random\_state=seed)

results = model\_selection.cross\_val\_score(model, X, Y, cv=kfold)

print(results.mean())

**Practical No. 7: Naive Bayes Classifier**

**Naïve\_Bayes.py**

from sklearn.datasets import load\_iris

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

iris = load\_iris()

X = iris.data

y = iris.target

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

clf = GaussianNB()

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

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

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:",accuracy)

**Naive\_Bayes2.py**

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

import pandas as pd

dataset = pd.read\_csv('argfrc.csv')

X = dataset[['Argentina', 'France']].values

y = dataset['Result'].values

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

clf = GaussianNB()

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

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

accuracy = accuracy\_score(y\_test,

y\_pred)

print("Accuracy:", accuracy)

**Practical No. 8: K - Nearest Neighbors (K-NN)**

import pandas as pd

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Load dataset

df = pd.read\_csv("Iris2.csv")

# Prepare features and target variable

x = df.drop(["Id", "Species"], axis=1).values

y = df["Species"].values

# Split the dataset into training and testing sets

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

# Initialize KNeighborsClassifier with k = 3

k = 3

clf = KNeighborsClassifier(n\_neighbors=k)

# Train the model

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

# Predict on the test set

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

# Calculate classification accuracy

classification\_accuracy = accuracy\_score(y\_test, y\_pred)

# Print the classification accuracy

print(f"Classification Accuracy: {classification\_accuracy:.2f}")

**Practical No. 9: Association Rule Mining**

from mlxtend.frequent\_patterns import apriori

from mlxtend.frequent\_patterns import association\_rules

import pandas as pd

dataset = [

 ['milk','bread','nuts'],

 ['milk','bread'],

 ['milk','eggs','nuts'],

 ['milk','bread','eggs'],

 ['bread','nuts'],

 ]

df=pd.DataFrame(dataset)

df\_encoded = pd.get\_dummies(df,prefix= ",prefix\_sep=")

frequent\_itemsets= apriori(df\_encoded, min\_support = 0.5, use\_colnames=True)

print("Frequent itemsets:")

print(frequent\_itemsets)

rules=association\_rules(frequent\_itemsets,metric="lift",min\_threshold=1.0)

print("\nAssociation Rules:")

print(rules)

**Practical No. 10: Classification of E-mail using Tensorflow**

import tensorflow as tf

import numpy as np # Added import for NumPy

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

emails = [

"Buy cheap watches! Free shipping!",

"Meeting for lunch today?",

"Claim your prize! You've won $1,000,000!",

"Important meeting at 3 PM.",

]

labels = [1, 0, 1, 0]

max\_words = 1000

max\_len = 50

tokenizer = Tokenizer(num\_words=max\_words, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(emails)

sequences = tokenizer.texts\_to\_sequences(emails)

X\_padded = pad\_sequences(sequences, maxlen=max\_len, padding="post", truncating="post")

model = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim=max\_words, output\_dim=16, input\_length=max\_len),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(16, activation='relu'),

tf.keras.layers.Dense(1, activation='sigmoid')

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# nd 'labels'.

training\_data = np.array(X\_padded)

training\_labels = np.array(labels)

model.fit(training\_data, training\_labels, epochs=10)

file\_path = "Spam.txt"

with open(file\_path, "r", encoding="utf-8") as file:

  sample\_email\_text = file.read()

sequences\_sample = tokenizer.texts\_to\_sequences([sample\_email\_text])

sample\_email\_padded = pad\_sequences(sequences\_sample, maxlen=max\_len, padding="post",

truncating="post")

prediction = model.predict(sample\_email\_padded)

threshold = 0.5

if prediction > threshold:

  print(f"Sample Email ('{file\_path}'): SPAM")

else:

  print(f"Sample Email ('{file\_path}'): NOT SPAM")