**Practical No: 1**

# Aim:

1. **To implement the Breadth First Search algorithm to solve a given problem.**
2. **To implement the Iterative Depth First Search algorithm to solve the same problem.**
3. **Compare the performance and efficiency of both algorithms.**

**i) To implement the Breadth First Search algorithm to solve a given problem.**

The BFS algorithm works as follows:

1. Start by initializing a queue and a set to keep track of visited vertices.
2. En-queue the source vertex into the queue and mark it as visited.
3. Repeat the following steps until the queue becomes empty:
4. De-queue a vertex from the front of the queue.
5. Process the vertex (print it, store it, or perform any other desired operation).
6. En-queue all the unvisited neighbors of the vertex into the queue and mark them as visited.
7. The algorithm terminates when the queue becomes empty, indicating that all reachable vertices have been processed.

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)

            #Explore neighbours

            neighbours=graph[vertex]

            for neighbor in neighbours:

                if neighbor not in visited:

                    queue.append(neighbor)

#Example usage

graph={

    'A':['B','C'],

    'B':['A','D','E'],

    'C':['A','F'],

    'D':['B'],

    'E':['B','F'],

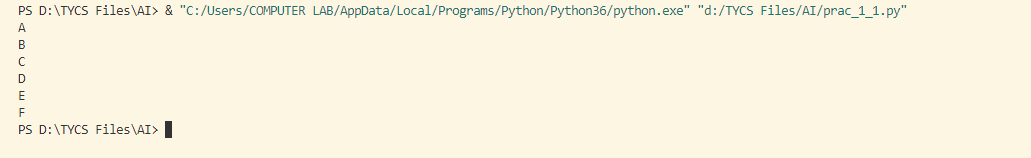
    'F':['C','E']

}

start\_vertex='A'

bfs(graph,start\_vertex)

**Output:**

****

**ii) To implement the Iterative Depth First Search algorithm to solve the same problem.**

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

#Example usage:

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)

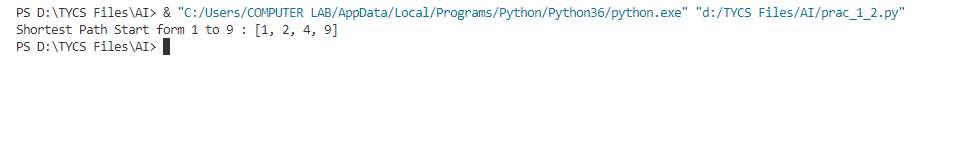
    if shortest\_path:

        print(f"Shortest Path Start form {start\_node} to {end\_node} : {shortest\_path}")

    else:

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

**Output:**

****

**Practical No: 2**

# Aim:

1. **Implement the A\* Search algorithm for solving a path finding problem.**
2. **Implement the Recursive Best-First Search algorithm for the same problem.**
3. **Compare the performance and effectiveness of both algorithms.**

**1) Implement the A\* Search algorithm for solving a path finding problem.**

The A\* algorithm works by maintaining two main values for each node: the cost to reach the node from the start node (known as g-value), and an estimate of the cost from the node to the goal node (known as h-value). It uses a priority queue, typically implemented as a min-heap, to prioritize the nodes for exploration based on their f-value, which is the sum of the g-value and h-value.

The A\* algorithm follows these steps:

1. Initialize the open list, closed list, and set the g-value of the start node to 0.
2. Calculate the h-value for each node in the graph or grid based on a heuristic function. The heuristic function estimates the cost from each node to the goal node. Common heuristic functions include Euclidean distance, Manhattan distance, or any other admissible and consistent heuristic.
3. Enqueue the start node to the open list with its f-value as the priority.
4. Repeat the following steps until the open list becomes empty or the goal node is reached:
5. Dequeue the node with the lowest f-value from the open list. This node becomes the current node.
6. If the current node is the goal node, the algorithm terminates, and the path has been found.
7. Add the current node to the closed list to mark it as visited.
8. Explore the neighboring nodes of the current node:
9. Calculate the tentative g-value for each neighbor by adding the cost to reach the neighbor from the current node to the g-value of the current node.
10. If the neighbor is not in the closed list or its tentative g-value is lower than its current g-value:
11. Update the g-value of the neighbor to the new lower value.
12. Calculate the f-value of the neighbor by adding its g-value and h-value.
13. If the neighbor is not in the open list, enqueue it with its f-value as the priority.
14. If the neighbor is already in the open list, update its priority if the new f-value is lower.
15. Set the parent of the neighbor to the current node.
16. If the open list becomes empty before reaching the goal node, there is no path available.
17. Once the goal node is reached, reconstruct the path by following the parent pointers from the goal node to the start node.

import heapq

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")

import heapq

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)

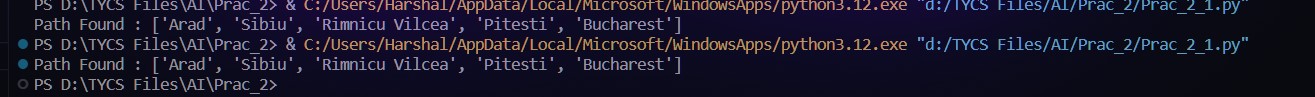
if path:

    print("Path Found :",path)

else:

    print("No Path Found")

**Output:**

****

**ii) To implement the Iterative Depth First Search algorithm to solve the same problem.**

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):

    # Define the successors for each state with their associated costs (simplified example).

    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.")

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):

    # Define the successors for each state with their associated costs (simplified example).

    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.")

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):

    # Define the successors for each state with their associated costs (simplified example).

    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.")

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):

    # Define the successors for each state with their associated costs (simplified example).

    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}:")

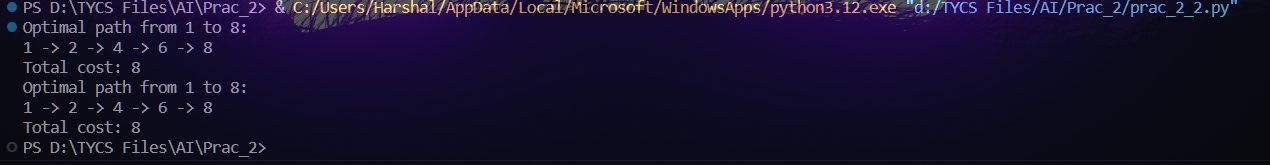
        print(" -> ".join(map(str, path)))

        print(f"Total cost: {cost}")

    else:

        print("No path found.")

**Output:**

****

**Practical No: 3**

# Aim:

**Decision Tree Learning**

**• Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.**

**• Evaluate the accuracy and effectiveness of the decision tree on test data.**

**• Visualize and interpret the generated decision tree.**

**Code:**# Import necessary libraries

import numpy as np

import pandas as pd  # Import Pandas for data loading

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

# Load your dataset from a local file (e.g., CSV)

# Replace 'Iris.csv' with the actual path to your dataset file

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

# Check the columns and first few rows

print("Columns in the dataset:", data.columns)

print("First few rows of the dataset:")

print(data.head())

# Assuming the target variable is in a column named 'species'

# Adjust 'species' to the actual target variable name based on your dataset

target\_variable = 'species'  # Change this based on your dataset

# Split features and target variable

X = data.drop(target\_variable, axis=1)

y = data[target\_variable]

# Split the dataset into a training set and a testing set

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

# Create a Decision Tree classifier

clf = DecisionTreeClassifier()

# Fit the classifier to the training data

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

# Make predictions on the test data

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

# Calculate the accuracy of the model

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

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

# Visualize and interpret the generated decision tree

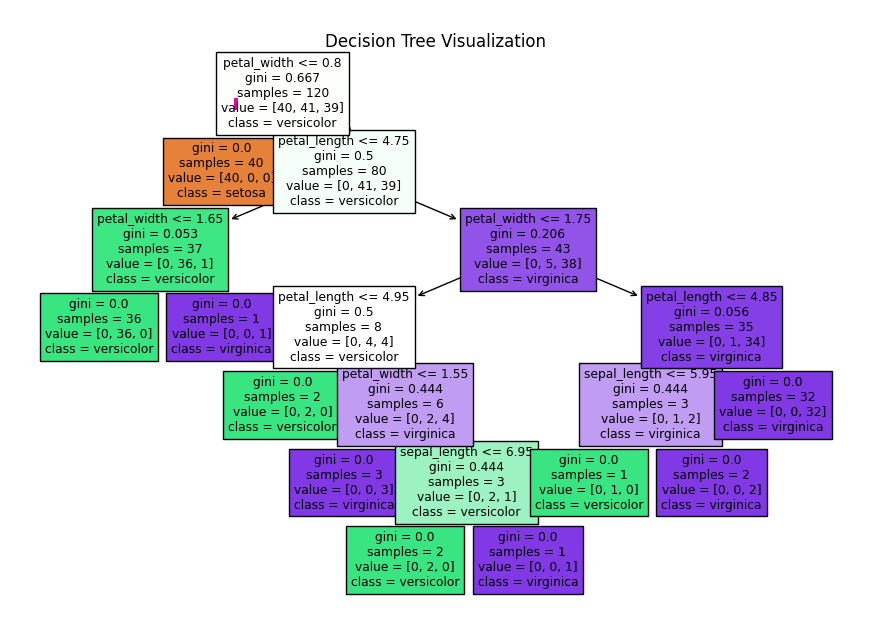
plt.figure(figsize=(12, 8))

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

plt.title("Decision Tree Visualization")

plt.show()

**Output:**



**Practical No: 4**

# Aim:

**Feed Forward Backpropagation Neural Network**

**• Implement the Feed Forward Backpropagation algorithm to train a neural network.**

**• Use a given dataset to train the neural network for a specific task.**

**• Evaluate the performance of the trained network on test data.**

**Code:**import numpy as np

# Define the sigmoid activation function and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Define the neural network class

class NeuralNetwork:

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

        # Initialize weights with random values

        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):

        # Forward propagation

        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):

        # Backpropagation

        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)

        # Update weights

        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)

# Define XOR dataset

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

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

# Create and train the neural network

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)

# Test the trained network

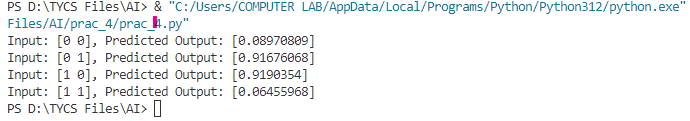
for i in range(len(training\_data)):

    inputs = training\_data[i]

    prediction = nn.predict(inputs)

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

**Output:**

****

**Practical No: 5**

# Aim:

**1) Implement the SVM algorithm for binary classification.**

**2) Train an SVM model using a given dataset and optimize its parameters.**

**3) Evaluate the performance of the SVM model on test data and analyze the results..**

**Code:**# Import necessary libraries

import pandas as pd

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

from sklearn.svm import SVC

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

# Load your dataset

# Replace 'your\_dataset.csv' with the actual path to your dataset file

data = pd.read\_csv('/content/iris.csv')

# Assuming the target variable is in a column named 'target'

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

y = data['Target']

# 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)

# Create an SVM classifier

svm\_classifier = SVC(kernel='linear', C=1.0)  # You can choose different kernels and adjust C

# Train the classifier

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

# Make predictions on the test data

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

# Evaluate the model

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

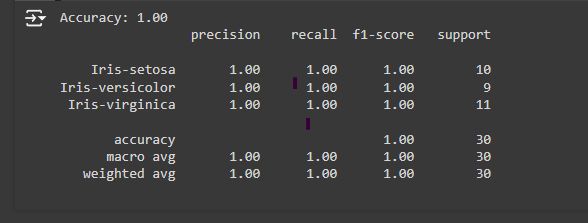
print(f"Accuracy: {accuracy:.2f}")

# Generate a classification report

report = classification\_report(y\_test, y\_pred)

print(report)

**Output:**

****

**Practical No: 6**

# Aim:

**Adaboost Ensemble Learning**

**• Implement the Adaboost algorithm to create an ensemble of weak classifiers.**

**• Train the ensemble model on a given dataset and evaluate its performance.**

**• Compare the results with individual weak classifiers.**

**Code:**from sklearn.datasets import load\_iris

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

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create individual weak classifier (Decision Tree)

weak\_classifier = DecisionTreeClassifier(max\_depth=1)

# Create AdaBoost classifier

adaboost\_classifier = AdaBoostClassifier(estimator=weak\_classifier, n\_estimators=50, algorithm='SAMME', random\_state=42)

# Train classifiers

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

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

# Make predictions

y\_pred\_weak = weak\_classifier.predict(X\_test)

y\_pred\_adaboost = adaboost\_classifier.predict(X\_test)

# Evaluate accuracy

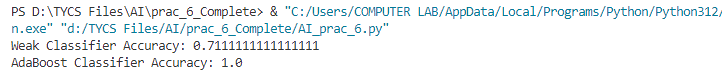
accuracy\_weak = accuracy\_score(y\_test, y\_pred\_weak)

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

print(f"Weak Classifier Accuracy: {accuracy\_weak}")

print(f"AdaBoost Classifier Accuracy: {accuracy\_adaboost}")

**Output:**

****

**Practical No: 7**

# Aim:

**Naive Bayes' Classifier**

**• Implement the Naive Bayes' algorithm for classification.**

**• Train a Naive Bayes' model using a given dataset and calculate class probabilities.**

**• Evaluate the accuracy of the model on test data and analyze the results.**

**Code: (Performed on Google Colab) https://colab.research.google.com/#scrollTo=xy9rbiAxTdof**

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

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# 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)

# Create a Naïve Bayes classifier (Gaussian Naïve Bayes for continuous features)

clf = GaussianNB()

# Train the classifier on the training data

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

# Make predictions on the test data

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

# Calculate and print the accuracy

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

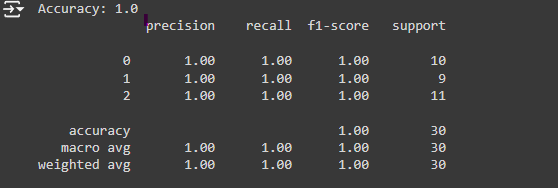
print("Accuracy:", accuracy)

# Generate a classification report

report = classification\_report(y\_test, y\_pred)

print(report)

**Output:**

****

**Practical No: 8**

# Aim:

**K-Nearest Neighbors (K-NN)**

**• Implement the K-NN algorithm for classification or regression.**

**• Apply the K-NN algorithm to a given dataset and predict the class or value for test data.**

**• Evaluate the accuracy or error of the predictions and analyze the results.**

**Code:**

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X, y = iris.data, iris.target

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create KNN classifier with k=5 (you can adjust k)

knn\_classifier = KNeighborsClassifier(n\_neighbors=5)

# Train the classifier

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

# Make predictions on the test data

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

# Evaluate accuracy

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

print(f"Accuracy: {accuracy}")

**Output:**

****

**Practical No: 9**

# Aim:

**Association Rule Mining**

**• Implement the Association Rule Mining algorithm (e.g., Apriori) to find frequent itemsets.**

**• Generate association rules from the frequent itemsets and calculate their support and confidence.**

**• Interpret and analyze the discovered association rules.**

**Code:**

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

from mlxtend.preprocessing import TransactionEncoder # Import TransactionEncoder

import pandas as pd

# Sample transaction data (replace with your own)

transactions = [

    ['milk', 'bread', 'eggs'],

    ['milk', 'bread', 'beer'],

    ['milk', 'diaper', 'beer', 'eggs'],

    ['milk', 'bread', 'eggs', 'beer'],

    ['bread', 'beer', 'diaper']

]

# Create a pandas DataFrame from the transaction data

dataset = pd.DataFrame(transactions)

# One-hot encode the transaction data

te = TransactionEncoder()

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

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

# Apply Apriori algorithm to find frequent itemsets

frequent\_itemsets = apriori(df, min\_support=0.2, use\_colnames=True)

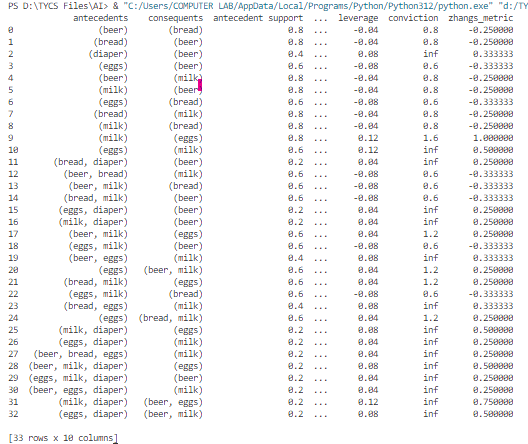
# Generate association rules from frequent itemsets

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.6)

# Print the discovered rules

print(rules)

**Output:**

****